SIBO 'C' Software Development Kit

OBJECT ORIENTED PROGRAMMING GUIDE

Version 2.01

August 23, 2000

CONTENTS

1 Introduction 1-1

Basic concepts 1-2
Classes 1-2
Object creation 1-2
Component objects 1-2
Object destruction 1-3
Categories 1-3
Category handles and category numbers 1-3
Message passing 1-4

Notation and conventions 1-5
Category numbers 1-5
Class names 1-5
Class numbers 1-5
Method names 1-5
Method function names 1-5
Messages and message numbers 1-5
Object handles 1-5
Method function prototypes 1-6

The basic component objects 1-7
The application manager 1-7
The window server object 1-8
Resources 1-8
The command manager 1-8
Client window 1-8
The engine 1-8
Menu bar 1-9
Dialogs 1-9

The required files 1-9
Category file 1-9
Source files 1-11
Method functions 1-11
Main 1-11
Resource externals file 1-12
Application Resource file 1-13
System resource file 1-13
Miscellaneous files 1-13
Icon 1-13
Add files list 1-13
Shell data file 1-13

2 Building an Object Oriented Application 2-1

An example application 2-2
The example source 2-2
Building the example application 2-5

3 Building a Dynamic Library 3-1

An example DYL 3-1
The example source 3-1
Building the example DYL 3-2
Using the example DYL 3-3

A DYL that supplies the ROOT class 3-4

Building DYLs into an application 3-5
DYL add-file lists 3-5
Accessing a built-in DYL 3-6

4 An HWIM Example - Hello World 4-1

The category file 4-3

The resource externals file 4-3

The resource file 4-4

The source code 4-5

Building the application 4-6

5 Commands and Command Menus 5-1

The command manager 5-1

Adding command options 5-2

Sharing method function code 5-5

Changing the text of an option 5-6

Disabling a menu option 5-8

Changing the number of options in a menu 5-8

Displaying a status window 5-9

Application-specific initialisation 5-10

Replacing a menu bar 5-10

Submenus 5-11

Shutdown messages 5-12

6 Windows 6-1

The WIN Class 6-2
Class definition 6-2
Property 6-3
Window flags 6-3

WIN methods 6-4
Create the window's window server data 6-4
Destroy 6-4
System-initiated redraw 6-4
Application-initiated redraw 6-5
Set visibility 6-5
Set window highlight 6-5
Sense start id for help 6-5
Calculate a window position 6-6
Set window position 6-6
Process a keypress 6-6

Deferred WIN methods 6-6
Initialise 6-7

Set property ... 6-7
Sense property ... 6-7
Draw to existing GC ... 6-7

The BWIN bordered window class ... 6-7
Class definition ... 6-8
Property ... 6-8

BWIN methods ... 6-8
Draw border ... 6-8
Update border ... 6-8

The LODGER class ... 6-9
Class definition ... 6-9
Property ... 6-10

LODGER methods ... 6-10
Initialise ... 6-10
Destroy ... 6-10
Set visibility ... 6-10
Create GC and draw ... 6-11
Check content is valid ... 6-11

Deferred LODGER methods ... 6-11
Sense required width ... 6-11
Update file name ... 6-11

The draw/redraw mechanism ... 6-11

Resizing a window ... 6-12

Window emphasis ... 6-12

7 Dialogs ... 7-1

The DLGCHAIN class ... 7-1

The DLGBOX class ... 7-2
Property ... 7-3
Dialog box flags ... 7-3
DLGBOX_ flags ... 7-4
DLGBOX_ITEM_ flags ... 7-5

DLGBOX methods ... 7-6
Consistency checks ... 7-6
Dynamic initialisation ... 7-7
Handle key input ... 7-7
Set item by index ... 7-7
Sense item by index ... 7-8
Sense item handle ... 7-8
Sense item index ... 7-8
Dim an item ... 7-8
Lock an item ... 7-8
Change the prompt for an item ... 7-9
Set item flags ... 7-9
Move focus to specified item ... 7-9
Add an item ... 7-10
Add an item by resource id ... 7-11
Replace an existing item ... 7-11
Specify minimum widths ... 7-11
Set size of dialog ... 7-12
Display 'dimmed' message ... 7-12
Sense start id for Help ... 7-13
Handle a keypress ... 7-13

Deferred DLGBOX methods ... 7-15
Item changed message ... 7-15
Focus changed message ... 7-15

Launch sub-dialog if required 7-15
Create non-system dialog item 7-15

Using dialog boxes 7-16
Default dialog behaviour 7-16
Dialogs and resource files 7-16
Launching a dialog 7-18
Simple dialogs 7-18
Dynamically initialised dialogs 7-19
Retrieving dialog results 7-20
Dialogs with and without 'WAIT' 7-21
Controlling the width of a dialog 7-21
Subdialogs 7-22

8 Dialog Controls 8-1

Text windows 8-2
Initialisation 8-4
Setting 8-4
Sensing 8-5

Choice lists 8-5
Initialisation 8-6
Setting 8-6
Sensing 8-6

Push buttons and action lists 8-7
Initialisation 8-8
Setting and sensing 8-9

Edit boxes 8-9
Initialisation 8-9
Setting 8-10
Sensing 8-10

LONG numeric editor 8-10
Initialisation 8-11
Setting 8-11
Sensing 8-11

Integer numeric editor 8-11
Initialisation 8-12
Setting 8-12
Sensing 8-12

WORD numeric editor 8-13
Initialisation 8-13
Setting 8-13
Sensing 8-13

Range numeric editor 8-14
Initialisation 8-14
Setting 8-14
Sensing 8-15

Floating point editor 8-15
Initialisation 8-15
Setting 8-16
Sensing 8-16

Date/time editor 8-16
Initialisation 8-16
Setting 8-17
Sensing 8-17

Latitude/Longitude editor 8-18
Initialisation 8-18
Setting 8-18

Sensing 8-19

File name editor 8-19
Initialisation 8-19
Setting 8-20
Sensing 8-20

File name choice list 8-20
Initialisation 8-21
Setting 8-21
Sensing 8-21

9 Active Objects 9-1

Active objects and asynchronous requests 9-1
Active object priorities 9-2
Application responsiveness 9-2
Background processing 9-2
Errors 9-3

A simple timer 9-3

10 Error Handling and Error Recovery 10-1

Errors during initialisation 10-1

General error recovery 10-2
The roll-back principle 10-2
Roll-back for component objects 10-3
Other resources in an object's property 10-3
Using the CLEANUP list 10-4

Interactions with system code 10-5

11 File-based Applications 11-1

Start-up initialisation 11-1

Opening and creating files 11-2
Switchfiles messages 11-3

Saving files 11-3
Application termination 11-3
Shutdown messages 11-3

12 Edit Windows 12-1

Introduction to EDWIN 12-1
Dialogs and edit windows contrasted 12-1
The NOTES example program 12-2
The “Hello World” program for edit windows 12-2
The EHELLO category file 12-3
Initialisation code in EHELLO 12-3
Other code in EHELLO 12-4

Simple use of EDWIN 12-5
Initialising an instance of EDWIN 12-5
The landlord of the edit window 12-6
The IN_EDWIN and IN_EDWIN_X data structs 12-6
The lg_set_id_pos method 12-7
Other edit window initialisation flags 12-7
A note on the CONTENTS field in the IN_EDWIN struct 12-8
Values of special characters in the text 12-9
A note on the MAXLEN field in the IN_EDWIN struct 12-9

The wn_sense method 12-9
The wn_set method 12-10
The wn_key method 12-10
The wn_emphasise method 12-11
The wn_draw method 12-11
Additional EDWIN methods 12-11
The ew_insert method 12-11
The ew_find method 12-12
The ew_replace method 12-12
The ew_replace_clip method 12-12
The ew_paste_clip method 12-13
The ew_evaluate method 12-13
The ew_set method 12-14
The ew_sense method 12-14
The concept of document offset 12-14
Allowed values of document offset 12-15
The EDWIN.CHANGE property 12-15
“Read-only” edit boxes and the ew_readonly method 12-15
The ew_leave method 12-16

Controlling the layout and formatting 12-17
An introduction to SCRLAY 12-17
SCRLAY structure definitions 12-17
Example: changing visibility of special characters 12-19
Default values of SCRLAY_STYLE in edit windows 12-20
Changing from the default layout style 12-20
An introduction to SCRIMG 12-20
SCRIMG structure definitions 12-21
Example: changing the width of the text cursor 12-22
Changing the font used by an editor 12-22
The ew_sense_size and ew_set_size methods 12-23
Changing the paragraph margins 12-24
Notifying SCRIMG of a change in style 12-24
Initialising the SCRIMG_WIN data structure 12-24

Direct interaction with document objects 12-25
Setting text directly into the document object 12-26
Dual variables at the EDWIN and SCRIMG levels 12-26
Adjusting the cursor position 12-27
Logical cursor movement and physical cursor movement 12-27
Notifying SCRIMG of a change in document content 12-28
Notifying SCRIMG of a local change in document content 12-29
When there is a change of content and a change in cursor position 12-30
The SCRLAY_DOC data structure 12-30
The five soft method numbers in SCRLAY_DOC 12-31
The SENSECHARS call-back 12-31
Structure of SCRLAY font width tables 12-33
The TOPARST call-back 12-33
The ENQPAGE call-back 12-33
The SENSEPDATA call-back 12-33
The SENSEPLABEL call-back 12-34

Some examples of edit-like windows 12-34
General comments on creating edit-like windows 12-35
The si_redraw method 12-35
The si_emphasize method 12-36
The si_pan method 12-36

13 Printing 13-1
Print preview 13-1

The basic model of WDR printing 13-1
Calculation of page breaks 13-2
Calculation of line breaks 13-3

Printer units 13-3
The difference between INDENT and RIGHT, and between DOWN and HEIGHT 13-3
Margins and page size 13-4
The PRINTER class and storage of the 'Print setup' dialog settings 13-4
Changing font or font style while printing 13-5
The text referenced in a print element 13-5
Limitations with the WDR_PRINT_KEEP flag 13-5
The need to specify font and style for each line 13-5
Use of WDR_PRINT_IDLE 13-6

Using LPRINTER for standard printing purposes 13-6
The syntax of the LPR_SENSE_TEXT callback 13-7
LPRINTER and word-wrap 13-7
Working out widths 13-7
Launching the print setup dialog suite 13-8

Examples of use of LPRINTER 13-8
Framework of the example applications 13-8
The 'Print details' dialog 13-9
Startup code and WS_DYN_INIT code 13-10
The LPRINTER initialisation code (first example) 13-11
The LPR_SENSE_TEXT method (first example) 13-12
Second example: additional initialisation code 13-12
The three states in printing a two-column display 13-13
More details about printing in columns with LPRINTER 13-15

Advanced uses of LPRINTER - and beyond 13-15
The LPR_READ method of LPRINTER 13-15
LPRINTER property introduced 13-16
The default word-wrapping algorithm 13-17
Calculating widths of text with variable font 13-18
Where printer font width tables come from 13-19
LPRINTER initialisation - phase one 13-19
A brief description of the PAGES active object class 13-20
More about the interface to and from PAGES 13-21
A brief description of the WDR class 13-22
Creating and destroying WDR objects 13-24

Using XPRINTER for print preview 13-25
The difference between XPRINTER and LPRINTER 13-25
Extended example of print and print preview using XPRINTER 13-26
The category file 13-27
Command manager 13-28
Print details dialog 13-29
Application initialisation 13-29
XPRINTER subclass initialisation 13-30
The XPRINTER LPR_SENSE_TEXT callback 13-31
Comments on the differences between XPRINTER and LPRINTER 13-32

WDR printing miscellany 13-33
WDR printing classes pictorial overview 13-33
The PDR class 13-34
Print preview without XPRINTER 13-34
Saving and restoring print context from file 13-35

14 Link Paste 14-1

The server side of link paste 14-1
Creating a LINKSV subclass instance 14-1
Declaring link paste server status 14-2
Initialising the SYSTEM component of w_am 14-3
The anatomy of a link paste transaction (server-side viewpoint) 14-4
Example LINKSV code 14-4
General remarks about link servers 14-5

Some standard link paste data formats 14-6
DF_LINK_TEXT and DF_LINK_PARAS contrasted 14-6
Word wrap and link paste 14-7
DF_LINK_TABTEXT 14-7
The hierarchy of text types 14-7

The client side of link paste 14-8
Determining whether there is suitable data available 14-8
The anatomy of a link paste transaction (client-side viewpoint) 14-8
Simple example of use of LINKCL 14-9

Special help with link pasting to and from edit windows 14-10
The ew_bring_in method of EDWIN 14-10
Simple example of calling EW_BRING_IN 14-11
The EWLINKSV class 14-11
The three text formats revisited 14-12

Native formats 14-13
Final comments 14-14

15 HWIM Resource Files 15-1

The application resource file 15-1
Resource file location 15-1
Loading an application resource 15-2
Resource Structures 15-2

The system resource file 15-2
Loading a system resource 15-2
Using system resources 15-3

Help resources 15-3
Using Help resources 15-4

16 Using the System Components 16-1

The application manager 16-1
Property 16-1
A command line example 16-1
Usable methods 16-2
Record a new filename 16-2
Load a resource 16-2
Load a resource to a buffer 16-3
Find application image file 16-3
Add a task 16-3
Wait for all activity to cease 16-3
Replaceable methods 16-3
Generate resource file name 16-4
Display notifier 16-4
Report an error 16-4

The window server active object 16-5
Property 16-5
Keyboard filters 16-6
Usable methods 16-7
Log a new client window 16-7
Run a dialog 16-7
Paragraph word wrap 16-8
Run help system 16-8
Get choice list resource text 16-8
Run the free-form dialling dialog 16-8
Alter the lock count 16-9
Set alternative menu bar 16-9
Reset the menu bar 16-9
Run a query dialog 16-9

Run an error dialog ... 16-10
Evaluate an expression ... 16-10
Set or get evaluator environment variable ... 16-10
Set or get dial environment variable ... 16-11
Run the set format dialog ... 16-11
Interface to wsAlert ... 16-12
Ensure print context data exists ... 16-12
Run print setup dialog ... 16-12
Run printer configuration dialog ... 16-12
Sense text for current printer device ... 16-13
Replaceable methods ... 16-13
Application-specific initialisation ... 16-13
Foreground message ... 16-13
Background message ... 16-13

17 HWIM utility functions ... 17-1

General utilities ... 17-2
Return TRUE ... 17-2
Return FALSE ... 17-2
Destroy an object ... 17-2
Make a window visible ... 17-2
Send command to command manager ... 17-2
Ensure path exists ... 17-3

Text management ... 17-3
Allocate cell and load resource ... 17-3
Load resource into buffer ... 17-3
Load choice list item into buffer ... 17-4
Generate error text ... 17-4
Generate formatted string ... 17-5
Generate formatted string, variable argument count ... 17-6
Append ellipsis to text ... 17-6
Set text mode ... 17-6
Set font ... 17-7
Set text style ... 17-7
Get normal text width for buffer ... 17-7
Get normal text width for string ... 17-8
Get bold text width for buffer ... 17-8

User notification ... 17-8
Display an information message ... 17-8
Display an error information message ... 17-9
Display a busy message ... 17-9
Make a beep ... 17-9

Run a dialog ... 17-10
Launch a dialog ... 17-10
Run a confirm dialog ... 17-10
Run a two-line confirm dialog ... 17-11
Run an error dialog ... 17-11

Dialog box utilities ... 17-12
Set an item ... 17-12
Set a text item ... 17-12
Set text in edit window ... 17-12
Set text in prompt window ... 17-13
Set choice in choice list ... 17-13
Set On/Off choice list to On ... 17-13
Set value of numeric editor ... 17-14
Set latitude/longitude editor ... 17-14
Set punctuation editor ... 17-15
Set floating point editor ... 17-15
Set date editor ... 17-16

Set range editor....17-16
Set dialog title....17-17
Sense an item....17-17
Sense edit window....17-17
Sense a choice list....17-18
Sense a numeric editor....17-18
Sense a range editor....17-19
Sense a floating point editor....17-19
Sense a latitude/longitude editor....17-19
Sense a punctuation editor....17-20
Sense a date editor....17-20
Dim/undim an item....17-20
Lock/unlock an item....17-21
Change focus....17-21
Set floating point editor from twips value....17-21
Sense twips value from floating point editor....17-22

18 Application Design....18-1

Basic design....18-1
The user interface....18-2
The engine....18-3

The Record application....18-3
Specification....18-3
Top-level view....18-3
Playing....18-4
Recording....18-4
Running for the first time....18-4
Menu....18-4
Design....18-5
The client window....18-6
The engine....18-7
Dialogs....18-8
The application manager....18-9

Appendix A - Category Files....A-1

Category file content....A-2
Class definition....A-3

Sub-category files....A-4
Using sub-category files....A-5

Category translation....A-6
The .ext external reference file....A-7
The .c C language category source file....A-8
The .g C language include file....A-9
The .asm assembly language category source file....A-10
The .ing assembly language include file....A-10
The .lis category listing file....A-10
The .c skeleton method function source file....A-11

Appendix B - Method Function Source Files....B-1
Method function parameters....B-1
Calling conventions and function order....B-1

Appendix C - Mechanisms....C-1
Classes....C-1
Class descriptor....C-1
Object creation....C-2

Categories C-3
Category handles and category numbers C-3
Dynamic linkage C-4
Referencing by category handle C-5
Message passing C-5
Calling conventions for method functions C-6
Method parameters C-6

CHAPTER 1

INTRODUCTION

This manual provides practical information on the use of Psion's Object Oriented Programming (OOP) system for the Series 3 and Series 3a machines. It illustrates how to use Object Oriented techniques to write applications, using the HWIM (Hand-held WIMP) dynamic class library, which is built into the ROM of Series 3 and Series 3a machines. Such applications can be written to at least the level of functionality obtainable by using standard C programming and the Hwif library, as described in the *Programming in Hwif* manual. In many respects the use of Object Oriented techniques allows applications to be written to a standard that is well above what can be achieved with the Hwif library.

The content of this manual assumes no particular prior knowledge of Object Oriented programming techniques. However, it would be useful to have read a textbook on the fundamental principles of OOP and to be familiar with basic concepts and the terminology in common use. There are many topics that are covered briefly in an early chapter and then described more fully at a later stage. The manual should therefore be read in its entirety for a full understanding, although many of the details may be skipped on first reading.

This manual assumes that the reader is familiar with the basics of producing applications for the Series 3 range of machines, as described in the *Series 3/3a Programming Guide*. It also assumes some familiarity with the use of windows and pull-down menus in the application's user interface as described, for example, in the *Programming in Hwif* manual. This form of user interface will also be familiar to those who have written OPL applications for the Series 3 or Series 3a.

Applications written using OOP and the HWIM library have a number of advantages over those written in OPL and those written in C, using the Hwif library. Some of the more significant advantages are:

- By default an HWIM application maintains its screen's appearance by using a dynamic redrawing technique. OPL and Hwif allow only the use of a backed-up bitmap, which consumes more memory and slows down drawing to the screen.
- A number of classes such as dialog boxes and dialog box controls already exist, with default behaviour. As such, they can be used directly, or subclassed as required for more specialised behaviour. Such classes permit quite sophisticated applications to be built fairly quickly.
- The contents of dialog boxes and menus can be changed dynamically, allowing a more sophisticated dialog with the user. Also, consistency and dependency checks can be performed before allowing the user to exit from a dialog. Both of these are difficult, if not impossible, under Hwif.
- Program activity may continue while a menu or a dialog is being displayed.
- All resources are placed in resource files. This encourages the creation of language independent applications.
- OOP permits software to be re-used; standard classes can be developed for one application and can be re-used in another, thus avoiding the need to "re-invent" software.
- The object paradigm is well suited to the design process, particularly for interactive event-driven programs with graphical interfaces. Good design permits quicker modification and allows maintainance to be done more easily, more quickly, more cheaply and with greater reliability.

Basic concepts

The material in this section provides a general overview of many of the basic mechanisms of Psion's OOP system. Further details on many of these topics will be found in *Appendix C*.

An object may be regarded as a combination of a number of items of data (referred to in this manual as the object's *property*) and a number of functions (the object's *methods*) that manipulate that data. An object usually represents some particular aspect of the problem under consideration. In general, the name of an object is noun-like and the names of its methods are verb-like. As an example, one of the most commonly used objects is a *window*, that is, an object that represents a rectangular area on a computer screen in which data may be displayed. In this case, the property includes the dimensions of the rectangle and one of the object's methods would be *resize*, to alter the window's dimensions.

An object is characterised by its *class definition*, specifying the property and the methods that are supported. Each object is said to be an *instance* of its class.

Classes

A *class* specifies the data (property) and behaviour (methods) of a particular type of object and is defined by an entry in a *category file*. The category file entry lists the property items and the method functions, in a way that is described later in this chapter and, in more detail, in *Appendix A*.

In general, a class is derived from another, more general class, known as its *superclass*. In such a case, one or more of the class methods (and items of property) may be supplied by the superclass. The class is said to be a *subclass* of its superclass. The subclassing process may be repeated to construct arbitrarily long subclass chains. In practice, however, such chains tend to be fairly short. On grounds of maintainability and comprehensibility it is inadvisable to construct long chains.

Although each subclass *inherits* methods and property from its superclass, a subclass can add its own methods and replace (that is, redefine) inherited methods to provide more specialised behaviour.

Object creation

An *instance* of a class is created by calling `p_new`, `f_new`, `f_newlibh`, `p_newlibh`, `f_newsend` or `f_newlibhsend`. These functions return a handle for the instance, and this handle must always be used to identify a particular instance. The handle is, in fact, a pointer to an allocated heap cell that contains the *property* (if any) of that class. This property includes any property inherited from superclasses.

All the functions (`p_new` etc) that create an object initialise the property with zeros.

Property may only be accessed via the object's handle. Within the method function code, this handle is conventionally given the name `self`. Note that, in contrast to some other object oriented systems such as C++, any code to which an object's handle is available has access to all the property of that object, including the property of all its superclasses. For those who are familiar with C++, the idea of private and protected data and functions does not exist. In terms of access to property and methods, Psion's Object Oriented Programming system is similar to Object Pascal.

Although not enforced by the programming system, the intention is that, by default, property should be considered as private to a particular class unless there are valid reasons for it to be externally accessed. Property of the classes supplied in the object libraries should always be considered to be private unless it is explicitly stated otherwise.

Component objects

An object's property may contain the handles of other objects, usually for the purpose of sending messages to them. Such a handle may have been passed to the object as the parameter to one of its methods and may be stored purely for convenience. A roughly equivalent result could have been obtained by storing the handle in a global variable.

A more significant case, however, is where the object 'owns' a subsidiary object that is an integral part of the owning object. Such a subsidiary object will normally have been created by the owning object, which is usually the only object to have any knowledge of its existence, and must be destroyed when the owning object is itself destroyed. In such a case the subsidiary object is said to be a *component* of the owning object. A class definition in a category file can mark one or more items of property as being the handles of component objects.

Object destruction

An object is normally destroyed by sending it a `DESTROY` message (sending a message is described later).

A root class (i.e. a class that has no superclass) normally contains a single method. This implements the default destroy method and is inherited by all other objects.

The default destroy method is designed to destroy the object *and all its components* (and the components of components and so on). In addition to avoiding the need to duplicate component-destroying code, this feature is one of the cornerstones of efficient recovery from error conditions. The standard root class (imaginatively named `ROOT`) is supplied by the OLIB class library and the default destroy method function, `root_destroy`, is provided by the PLIB library.

Some object classes may create resources that are not objects (e.g. an I/O channel, which needs to be closed) or may wish to destroy objects in a specific order. Such classes generally replace the destroy method to clean up the resources introduced by the class. In addition to its class-specific action, the method must "supersend" the destroy message to the superclass in order to continue the destruction process. Every object is expected to execute `root_destroy` at some stage in its destroy method.

Objects in existence at the termination of an application do not need to be explicitly destroyed. The EPOC operating system ensures that all resources used by a process are released when the process terminates.

Categories

A *category* is a group of one or more classes packaged into a *load module* which, when loaded, occupies a single code segment. Category code segments are shared - there is only one copy of a particular category in memory, however many processes are executing it.

There are two main groups of categories:

- *Image categories* are used to implement programs. The name of a code segment that contains an image category has the extension *.$sc*.
- *Dynamic library categories* (DYLs) contain classes that are referenced from image categories and other DYLs. The name of a code segment containing a DYL has the extension *.dyl*.

A straightforward small- to medium-sized application, such as those described in this manual, typically consists of a single image category that references the built-in ROM DYLs.

Programmers may, however, develop their own DYLs for one of the following reasons:

- A larger application can choose to be organised into multiple categories to limit its working set by selectively loading transient subsystem categories into memory (analogous to overlays in single-tasking operating systems).
- A large application may use DYLs simply to overcome the 64K code segment limit.
- An application may wish to develop an open-ended set of "polymorphic" DYLs to implement, for example, a set of different printer drivers.
- To develop a general-purpose DYL which supplements the system object libraries.
- To provide the common functionality for a product that consists of a suite of application programs.

Dynamic libraries have the following advantages over normal (static) libraries:

- only one copy of the code is present in memory however many processes are using it
- the DYL code does not detract from the 64K segment limit of the application that is using it
- provided you don't change the interface to the DYL (or at least make it upward compatible), you don't need to relink the applications that use the DYL when you build a new DYL

Category handles and category numbers

A reference to a category may be made either by a category handle or by category number. The reference may be to:

- the *local* category, that is, to the category containing the code that makes the reference

- an *external* category, that is, any category other than the local category.

A category *handle* uniquely identifies a category code segment, but is only known at run time, after the categories have been dynamically linked (for example, by means of a call to `p_linklib`).

A category code segment may also be identified from within a category by a category *number*, which is known at compile time. A category number is not unique, in that two categories will, in general, use different category numbers to refer to the same external category.

Because of this fact, a category number should not be passed as a parameter to an external method (for example, to create a component of variable class). When there is a requirement to pass a category as a parameter, the category handle rather than the category number should be used. After dynamic linking, the category handle may be obtained from the category number by calling `p_getlibh`.

A category number is mainly used to create an instance of an object class using `p_new`, `f_new` or `f_newsend`. These functions automatically convert the passed category number to the corresponding category handle.

The most common use of a category handle is to create an instance of an external class using `p_newlibh`, `f_newlibh` or `f_newlibhsend`.

Message passing

In OOP terminology, sending a message to an object means calling a method function of the class or superclass of which that object is an instance, the method function being identified by its *method number*.

The most common way of sending a message is to use `p_send` or, more efficiently, one of the `p_sendn` variants. All of these functions must be supplied with the handle of the object instance (as returned by, say, `p_new` or `p_newlibh`) and the method number as their first two parameters. Up to three additional parameters may be supplied.

The `p_send` function locates the appropriate method function by scanning up the superclass chain, starting with the class of which the object is an instance, selecting the first matching method function.

If no suitable method is located in the superclass chain, the sending function panics with panic number 48. The send will also panic (with panic number 55) if any class that is scanned does not have a valid structure. This catches, amongst other things, the sending of a message to an object that has already been destroyed.

If successfully located, the method function is passed the object handle and the optional parameters (the method number passed to `p_send` is suppressed).

Although `p_send` is the most commonly used message-sending function, the following functions may also be used:

`p_supersend`	this is used within a method function to send a message of the same method number to the same object, but to be handled by a superclass method. It works like `p_send` except that the search for a method starts at the immediate superclass of the class associated with the method containing the call to `p_supersend`. It is typically used within a subclass method that adds further processing (before, after or around the call to `p_supersend`) to the method being replaced.
`p_entersend`	this works like a `p_send` that has been called with a `p_enter` - but more efficiently
`p_exactsend`	this can send a message to a method of a specific class in an object's class tree. The search for the method starts at the specified class. It is frequently used within a method function to send a message to the same object, but to be handled by a superclass method once removed - in effect a super-supersend.

A method function is normally declared with the `METHOD_CALL` calling convention. Other calling conventions may be needed in some circumstances, as described in the *Calling conventions for method functions* section of *Appendix C*.

Notation and conventions

Category numbers

The local and external category numbers are represented in code by means of generated symbolic constants. The symbolic name for a category number is generated, in upper case, from the names of the local and externally referenced categories, separated by an underscore, and a `CAT_` prefix.

The symbolic name indicates in an explicit manner the category from which the reference is being made and the category that is being accessed. For example, from a MYCAT category the (external) category number of the OLIB category is represented by `CAT_MYCAT_OLIB`. The local category is, in this case, represented by `CAT_MYCAT_MYCAT`.

Class names

A reference to a class name in the text of this manual is given in upper case. The HWIM command manager class, for example, whose class definition starts with the line:

```
CLASS comman root
```

is referred to in the text as `COMMAN`.

Class numbers

Each class within a category has an associated class number, used by code that creates an instance of the class (such as a call to `p_new`). A class number is represented by a generated symbolic constant. The symbolic constant name is generated, in upper case, by prefixing the class name with `C_`. Thus the `COMMAN` class number is represented by `C_COMMAN`.

Method names

A method name, as declared in a class definition, by convention is normally given a two or three letter prefix that indicates the class in which it is declared. The prefix is separated by an underscore from the remainder of the name. For example, `com_init` is a method of the `COMMAN` class. The prefix will, in general, be different from the prefix used for methods of a superclass.

The method name is used when building the symbolic constant for the associated method number.

Method function names

A method function *must* be given a name constructed from the class name, followed by an underscore and the method name. For example, the method function associated with the `com_init` method of the `COMMAN` class must have the name `comman_com_init`.

Messages and message numbers (method numbers)

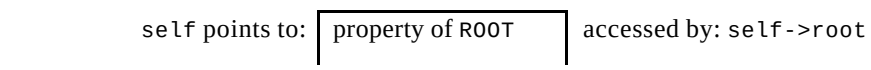
Each method is identified by a method number, represented by a generated symbolic constant. The symbol is generated, in upper case, by prefixing `O_` to the method name. For example, the `com_init` method has a method number represented by the symbol `O_COM_INIT`.

The message itself is referred to in the text simply by an uppercased method name. For example, the `com_init` method is invoked by the sending of a `COM_INIT` message.

Object handles

An object handle identifies a particular object (instance of a particular class). It is, in fact, a pointer to the memory cell that results from a call to one of the object-creating functions (`p_new`, for example). The cell contains a C struct, whose typedef appears in the appropriate *.g* file generated from the category file by CTRAN. The type name of the struct is generated by prefixing `PR_` to the class name. Thus the handle of an object of class `WIN` is a pointer to a `PR_WIN` struct. Within the method functions of a class the handle of an instance of that class is conventionally given the name `self`.

The memory cell pointed to by the object handle contains the object's property, including the property defined for all its superclasses. Take, for example, the `HWIMMAN` class. This subclasses `APPMAN` which, in turn, subclasses `ROOT`. The handle of an `HWIMMAN` object points to a `PR_HWIMMAN` struct that is composed as follows:



property of `APPMAN`	accessed by: `self->appman`
property of `HWIMMAN`	accessed by: `self->hwimman`

If a class supplies no additional property then the corresponding element of the struct does not exist.

The object handle may optionally be declared as a pointer to any superclass. Thus the pointer in the above example may be declared as a pointer to either `PR_ROOT` (which gives access to only the `ROOT` component) or `PR_APPMAN` (with access to `ROOT` and `APPMAN` property). It may also declared as a `VOID *` in which case, obviously, there is no access to any of the object's property.

Method function prototypes

The description of each method in the documentation contains a function prototype that specifies the nature of any return value and the parameters with which the method is called. By convention, the listed parameters always exclude the object handle that is passed as the first parameter to every method function. It also does not show the method function number, which is passed to the message-sending functions such as `p_send`, but is removed by the message-sending mechanism and is never passed to a method function.

For example, a `wn_emphasise` method for the class `WIN`, described in the documentation by the prototype:

```
VOID wn_emphasise(UINT flag);
```

would be invoked by, for example:

```
p_send3(hand,O_WN_EMPHASISE,TRUE);
```

where `hand` is the handle of an object of the `WIN` class and `O_WIN_EMPHASISE` is the method number.

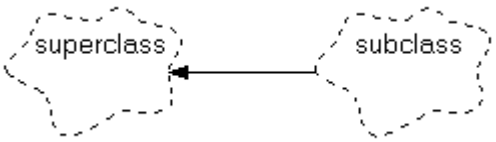
This corresponds to a method function declared in C source code as:

```
METHOD VOID win_wn_emphasise(PR_WIN *self,UINT flag)
    {
    ...
    }
```

Class diagrams

The class diagrams in this manual broadly follow the notation as used in *Object Oriented Design with Applications*, by Grady Booch (The Benjamin/Cummings Publishing Company Inc, 1991). A class is represented as shown below, where the class name is written inside the symbol. An underlined name represents an imported class, that is, a class whose class definition is in an external category.

Relationships between classes may be that one class subclasses another, or that a class uses another. The superclass/subclass relationship is represented by an arrow joining the two classes, as illustrated in the following diagram.

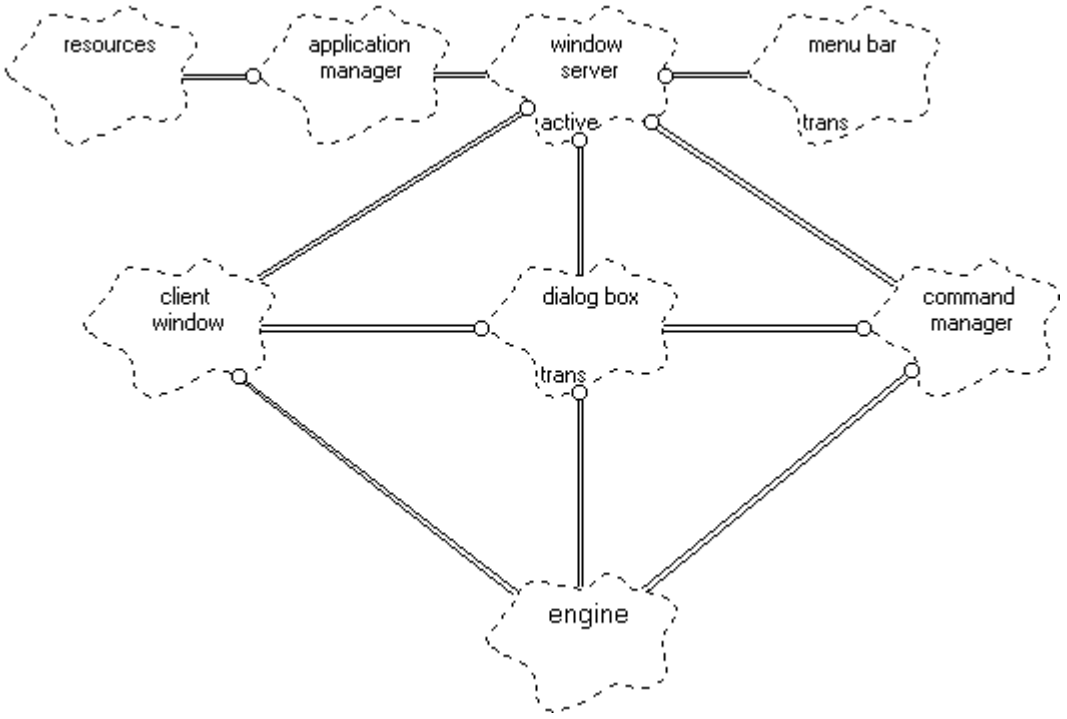


A 'using' relationship is illustrated in the following diagram, where class A uses class B. The using relationship may mean that class B is a component of class A, or simply that class A sends messages to class B.

In some cases, where the two classes may be considered co-equals, the using relationship may be shown without the circle that indicates the direction of the relationship.

The basic component objects

The standard classes that are present in a normal HWIM application are as illustrated in the following class diagram.



There are five main static objects (by *static* we mean an object that exists for the lifetime of the application) provided by HWIM; the application manager, the application's resources, the window server object, the command manager and the client window. A sixth static component that is usually present in non-trivial applications is the engine. The menu bar and, optionally, one or more dialog boxes are transient objects, being created when they are required and destroyed on completion of their function.

The application manager and the window server object together form the central core of the application. From the application programmer's point of view they may be considered as a single entity that provides the application's main event-handling loop and a range of system services. The separation of this functionality between two objects represents a division of labour; the window server object deals with the user interface and the application manager handles those aspects that are independent of the user interface. (Although it is beyond the scope of this manual, it is worth pointing out that an application with no user interface can be constructed around the application manager alone.)

The application manager

The application manager provides the framework for an application, including its main event-scheduling loop. It performs all standard start-up and initialisation. As part of this initialisation it creates a window server active object (and hence a command manager - see later) and opens the system resource file and the application's own resource file. In addition, the application manager supplies methods for manipulating other standard system components and adding further optional components.

The HWIM library supplies the HWIMMAN application manager class, which is a subclass of the OLIB APPMAN class (see the *OLIB Reference* manual). An instance of this class is normally created and initialised from the application's main(). It is rarely subclassed by an application, the main exception being that of a multi-lingual application, which will need to modify the mechanism that loads the application resource file.

An application may subclass HWIMMAN by replacing existing methods. In the interests of future compatibility, application-specific subclasses should not add methods or property. An application that adds methods or property to the application manager is not guaranteed to run on future versions of machines in the Series 3 range.

The window server object

The window server is an active object that acts as the source of those events (keypresses, redraws etc) that are sent to the application by the window server process. On receipt of such an event the window server object directs it to the appropriate object, normally either a window (the client window or a dialog box) or the objects involved in the command execution mechanism.

An instance of a window server object is automatically created during the initialisation of the application manager. As part of its standard initialisation the window server object creates and initialises a command manager.

The HWIM library supplies the WSERV window server object class which is a subclass of the OLIB ACTIVE class. HWIM applications will almost invariably subclass WSERV , replacing its ws_dyn_init method, to provide application-specific initialisation. Part of this initialisation will be the creation and initialisation of a client window.

An application is free to subclass WSERV by replacing existing methods. In the interests of future compatibility, application-specific subclasses should not add methods or property. An application that adds methods or property to the window server object class is not guaranteed to run on future versions of machines in the Series 3 range.

Resources

All standard HWIM applications are assumed to have access to two resource files - the system resource file (in the ROM) and an application resource file. These are opened automatically during initialisation of the application manager, which also provides methods to access individual resources. There is rarely any need for an application programmer to subclass the resource file class.

The application resource file must contain the resources that provide the accelerator keypresses and the text of the application's menu bar and pull-down menus. In addition to these essential items, it may also contain resources for any application dialogs and other application-specific text.

The command manager

The command manager supplies the functionality to execute the command options that may be selected from the application's pull-down menus. A command manager instance is automatically created during the initialisation of the window server active object.

The HWIM library supplies the COMMAN command manager class, which provides the basic skeleton for a command manager. Although a very simple application could make direct use of an instance of the COMMAN class, HWIM applications will normally subclass COMMAN, replacing one or more of the supplied methods and adding application-specific methods and property.

Client window

All standard HWIM applications are assumed to have a main window, designated as the client window to provide the principal view of the application's data. This window will receive messages from the window server active object in response to window server events (such as keypresses). The client window must be explicitly created and initialised by application-specific code, normally from within the window server object's ws_dyn_init method.

The functionality of the client window varies widely from application to application and much of an application's code will be associated, directly or indirectly, with the client window. For this reason, the HWIM library provides very general window classes that will normally be extensively subclassed in most applications.

The engine

The engine is an application-specific class that is normally created at the same time as the client window. A typical engine will subclass ROOT. Engines are further discussed in the *Application Design* chapter.

Menu bar

An instance of the application's menu bar class is automatically created by the window server object whenever the menu bar must be displayed (for example, when the Menu key is pressed) and is destroyed when the menu bar disappears.

The menu bar class contains the mechanism to convert a menu selection into the appropriate command manager message and is unlikely to be subclassed in any application.

Dialogs

A dialog is usually created by application-specific code in a command manager method that is called in response to the selection of a command menu option. Such a dialog may simply use the supplied `DLGBOX` class or may be an application-specific subclass. The use of dialogs is discussed in some detail in the *Dialogs* and *Dialog Controls* chapters.

The required files

To create an object oriented application the writer's task consists of constructing a number of files which will be input to the build process. A number of tasks must be performed which can be broadly defined as follows:

- building the required classes by defining their property and methods; further, deciding whether any of the required classes can be subclassed from existing classes, thereby re-using existing software
- providing the functionality for a number of method functions that will be called by system code
- writing a short `main()` that creates and initialises an application manager, supplying it with one or more of a set of options
- building the resource file(s) and optionally, the Icon file, the Add files list and the Shell data file.

The following sections describe the structure and content of the files required to create an object oriented application in more detail and broadly correspond to the tasks outlined above.

Category file

The category file defines the application-specific classes - methods, property and associated defined constants and structs. As part of the process of building an application, the category file is translated with the aid of the CTRAN tool. The output from the translation process is a C source file (which is also compiled, ready for linking into the application) one or more generated include files, each with a *.g* extension and an external file with a *.ext* extension. Other files may optionally be generated.

The external file contains information about this category which will be needed when translating any other category which makes an external reference to this one.

The content of a category file is best explained in conjunction with the following short example. A more complete explanation is contained in *Appendix A*. Here we shall concentrate on the basic content of a class definition.

The file header contains the category name, external category references (the order of which determines the external category number sequence) and a number of included header files:

```
A demonstration cat file
IMAGE demo

! External reference to OLIB library
EXTERNAL olib

INCLUDE p_std.h
INCLUDE p_object.h
INCLUDE varray.g          required, in this case, for knowledge of VAFLAT
```

This is followed by one or more class definitions, each of which follows the general model illustrated below:

```
CLASS dummy root           the class name and its superclass
Dummy class definition,
as an illustration only
    {                      Methods follow...
    REPLACE destroy        free buffer and supersend
    ADD dm_init            create VAFLAT component and allocate buffer
    DEFER dm_sub           defined by a subclass...
    CONSTANTS              auxiliary symbolic constants
        {
        ! for the buffer
        DUMMY_BUF_SIZE 128 allocated buffer size
        ! for the VAFLAT component
        DUMMY_GRAN     16
        }
    TYPES                  contains auxiliary structs
        {
        typedef struct     /* comments here are exceptional */
            {
            TEXT *buf;     pointer to allocated buffer
            UWORD len;
            } DUMMY_BUF;
        }
    PROPERTY 1
        {
        PR_VAFLAT *array;  the component VAFLAT instance
        DUMMY_BUF buffer;
        }
    }
```

The CLASS keyword introduces a class definition. It is followed by the name of the class and then the name of the parent superclass. The above example defines the class DUMMY which is a direct subclass of the ROOT class. The layout of a class definition is significant; apart from leading whitespace, which is ignored, it must follow the pattern that is illustrated above - and in the class definitions given elsewhere.

The class definition of each subclass lists its additional methods and any additional property. It may also, as in the above example, include the definitions of auxiliary structures and constants used by that class. There are many further examples of class definitions throughout this and other manuals (in the *OLIB Reference* manual, for example).

The class definition may include any number of method declarations,[1] introduced by the ADD, REPLACE or DEFER keywords. Each of these is followed by a method name. The method declarations may be followed by one of each of the CONSTANTS, TYPES and PROPERTY keywords.

The method declaration keywords have the following meanings:

ADD	declare a method in addition to the methods provided by the superclass. The name must be unique in relation to all other methods in this category, or any externally referenced categories. Although not compulsory, the name conventionally starts with a short prefix related to the name of the class in which it is introduced.
REPLACE	declare a method whose functionality is to replace that of a method supplied by a superclass. The name must be that of an existing method in the superclass inheritance tree.
DEFER	declare an additional method as for ADD, except that the functionality of the method is not defined by the current class and is expected to be provided by a subclass (using REPLACE). A class containing one or more DEFERred methods is known as an *abstract* class and, in general, no instances of such a class will ever be created.[2]

It is recommended that each method name be followed by a concise descriptive comment.

[1] Subject to a maximum of 255 methods, including those inherited from superclasses.

[2] There is no formal requirement for all DEFERred methods to be REPLACEd and it is acceptable to create an instance of such a class provided that it is known that no DEFERred method will ever be called. Window subclasses, for example, do not need to REPLACE all DEFERred methods of the HWIM WIN superclass.

The CONSTANTS keyword introduces a list of symbolic constant definitions, each consisting of the symbol name (conventionally in upper case) followed by the numeric value. The value may be an expression involving symbolic constants defined earlier, either in the category file itself, or in any included file. The expression itself must not contain any whitespace.

The TYPES keyword introduces a list of C language typedef struct definitions, whose layout should follow that given in the example category file. (Many further examples may be found in the class definitions shown for each class in, say, the *OLIB Reference* manual.)

The PROPERTY keyword introduces a list of data element declarations to be included in the struct that defines the class property. The form of this struct is described in the *Category Translation* chapter of the *Object Oriented Programming Reference* manual.

This keyword may optionally be followed by a literal number (expressions may not be used) that specifies how many component items listed in the property are to be sent an automatic DESTROY message when an instance of the class is destroyed.

This assumes that, for a value ncomp, the first ncomp items in the additional property for the class are either NULL or handles (pointers to instances) of component objects (as defined earlier in this chapter).

In the above example, DUMMY's component VAFLAT instance will be automatically destroyed when DUMMY receives a DESTROY message.

Source files

Method functions

A method function must be supplied for each added or replaced method in each application-specific class. The method functions may be supported by auxiliary (or utility) functions - that is, normal C functions called from within the method functions. These functions may be in a separate file or in the same file as the method functions.

The method functions themselves may be organised into C files in any suitable way. Normally, all the method functions for a particular class will be grouped into one file, but this is not a requirement. It may be convenient to group together all the methods of a related set of objects, and a simple application may have all its method functions in a single source file.

Where a file contains a mixture of method functions and auxiliary functions, it is conventional to put all the auxiliary functions at the top of the file, followed by the method functions. One advantage of this scheme is that it minimises the number of compiler directives needed to establish the correct calling conventions for the different function types. This may be extended to include other function calling conventions so that, in general, a source file will have the following form:

```
<includes>

<'normal' functions>

#pragma ENTER_CALL

<functions called via p_enter>

#pragma CDECL

<method functions that are callable via p_enter>

#pragma METHOD_CALL

<method functions>
```

Failing to declare the correct calling convention for a function will cause unpredictable run-time errors when the function is called.

Main

The main() of an object oriented application that uses the HWIM library takes the following form:

```
#include <hwimman.g>

GLDEF_C VOID main(VOID)
    {
    IN_HWIMMAN app;
    IN_WSERV ws;
    VOID *handle;

    p_linklib(0);
    app.flags=FLG_APPMAN_RSCFILE|FLG_APPMAN_SRSCFILE|FLG_APPMAN_CLEAN;
    app.wserv_cat=p_getlibh(CAT_MYAPP_MYAPP);
    app.wserv_class=C_MYAPPWS;
    ws.com_cat=p_getlibh(CAT_MYAPP_HWIM);
    ws.com_class=C_COMMAN;
    handle=p_new(CAT_MYAPP_HWIM,C_HWIMMAN);
    p_send4(handle,O_AM_INIT,&app,&ws);
    }
```

The call to `p_linklib` dynamically links the application category with the DYLs (HWIM, OLIB etc.) in the ROM. If the application uses one or more application-specific categories it may be necessary, depending on how they are used, to load and link them at this point. A failure to link the appropriate categories can cause a range of object-related run-time errors.

The `flags` field of the `IN_HWIMMAN` struct specifies a range of options that are used during initialisation of the (`HWIMMAN`) application manager. The possible range of flags is described in the *OLIB Reference* manual. The three flags specified in the above code, specifying that the application uses the system resource file, an application resource file and a `CLEANUP` object, are mandatory for all HWIM applications.

By default an application will be built to run on the Series 3 and will run in compatibility mode on the Series 3a. `OR`'ing `FLG_APPMAN_FULLSCREEN` into the `app.flags` field specifies that the application should not run in compatibility mode on the Series 3a.

The `wserv_cat` and `wserv_class` fields of the `IN_HWIMMAN` struct must be set to the category and class numbers of the application's window server object. As indicated in the example, all applications will use an application-specific subclass of the `WSERV` class that is supplied by HWIM.

Similarly, the `com_cat` and `com_class` fields of the `IN_WSERV` struct must be set to the category and class numbers of the application's command manager that is created during the initialisation of the window server object. The example specifies the use of `COMMAN` itself: normally an application will use an application-specific subclass of `COMMAN`.

The final action is to create and initialise the application's command manager (typically, as in this case, the `HWIMMAN` class supplied by the HWIM category). The application manager locates and opens the resource files, creates and initialises both the window server object and command manager and starts the application's main event-processing loop. All further interaction with application-specific code is via calls from system code to a range of method functions. The first such call is to the window server object's `ws_dyn_init` method, which is assumed to perform all necessary application-specific initialisation.

Note that the application will never return from the sending of the `AM_INIT` message to the application manager.

Resource externals file

The resource externals file, with a *.re* extension, is used to extract from the application's header files those defined constants that are needed during the compilation of the application resource file. It is used to generate a *.rg* file that is `#included` in the application resource file, thus avoiding the need to `#include` a large number of separate header files.

In larger applications, this process can reduce the time taken to perform a resource compilation and reduces the risk of out of memory situations occurring by effectively passing only those definitions required for successful resource compilation. It is also useful if the resource file is to be translated. The (usually small) *.rg* file can be supplied to the translator together with the resource file, rather than having to supply a (usually large) number of other header files. The translator can then easily make a test compilation of the translated resource file, with consequent savings in time and effort.

Application Resource file

Application resource files contain the text strings used in the application's command menus, dialogs, text messages and so on. In object oriented programming, such strings must be kept separate from code.

In general, resource files are useful for the following main reasons:

- having data in a resource file rather than in the code reduces the size of the code segment
- resource files make it easier to write language independent applications

In essence, any resource item within a resource file can be identified by a unique number. This number is usually assigned to a symbolic constant that is published in a header file generated by the resource compiler. Including this header file in a source file allows code to reference the resources.

The structure of resource files is described in much greater detail in the *Resource Files* chapter in the *Additional System Information* document.

Further information can also be found in the *HWIM Resource Files* chapter in this manual.

System resource file

The system resoure file is built into the ROM and is functionally similar to application resource files. However it contains *common* resources, including many system dialogs, the text for standard information, error messages and basic help.

The general structure of the system resource file is similar to that of application resource files and is described in the *Resource Files* chapter in the *Additional System Information*.

Again, further information can be found in the *HWIM Resource Files* chapter in this manual.

Miscellaneous files

Icon

An application can be represented by an Icon. Although not mandatory, the system screen will refuse to install the application unless it contains one. In this situation, the application can still be run from RunImg but an empty icon boundary will be displayed.

An Icon is normally placed in a *.pic* file and can be produced in a variety of ways:

- using the *Iconed* demonstration application that can be built using the HWIF part of the SDK, or the Series 3a *Iconeda* application that is installed into the *\sibosdk\s3atool* directory.
- using the window server tool *wspcx.exe* on the *.pcx* output of a PC program such as *Windows PaintBrush*

The format of *.pic* files is given in the *Bitmaps* section in the *Window Server Reference* manual.

For further information on Icons, see the *Series 3/3a Programming Guide*.

Add files list

An add files list is a text file with extension *.afl* which contains from one to four filenames. In essence, as part of the process of building an application, the files referred to in this list are combined with the application's *.img* file to produce a larger *.img* file.

The list can refer to a *.pic* file for an Icon, a *.rsc* file for an application's resource file and so on.

For further information on add files, see the chapter *Building an Application* in the *General Programming* manual.

Shell data file

The Shell data file is a file which can be included in the add files list (and therefore embedded into the application). It specifies information required by the System Screen application (also known as the *Shell*).

For example, it tells the *Shell* the expected extensions of any files to be edited and the default directory of these files. This information is specified at compile time.

For further information on the Shell data file, see the *Communicating with the System Screen* chapter in the *Series3 Programming Guide*.

CHAPTER 2

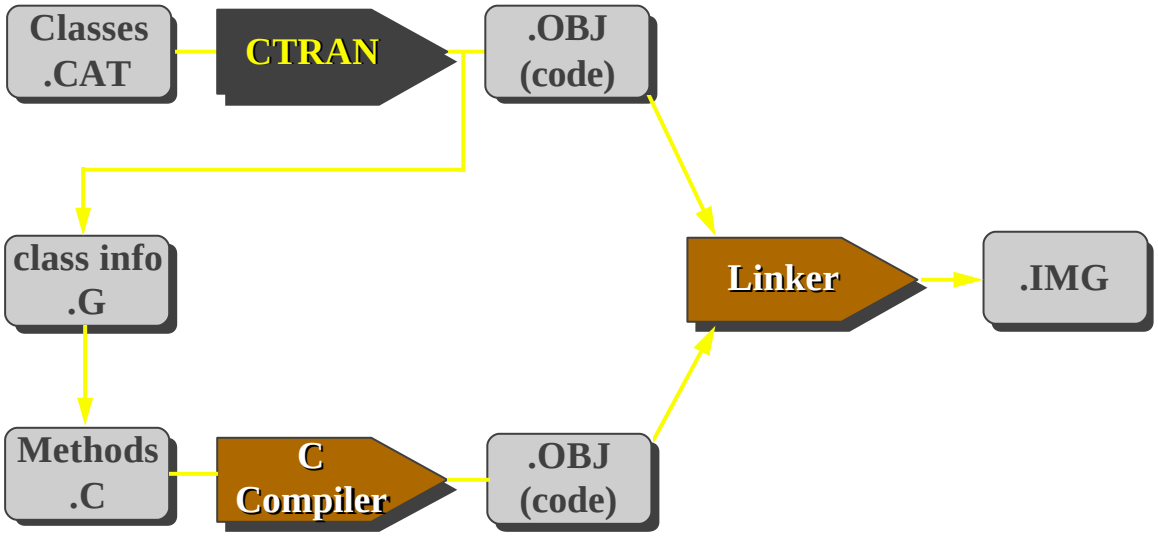
BUILDING AN OBJECT ORIENTED APPLICATION

The process of building an application that contains object oriented code is very similar to that used for building non-object oriented multi-file programs. The basic mechanisms of compiling and linking the various modules are as described in the *Building an Application* chapter of the *General Programming Manual*.

Perhaps the most obvious difference is that, in addition to the file(s) containing source code, an object oriented program also requires a *category file*, described in the *Category Files* chapter of the *Object Oriented Programming Reference* manual.

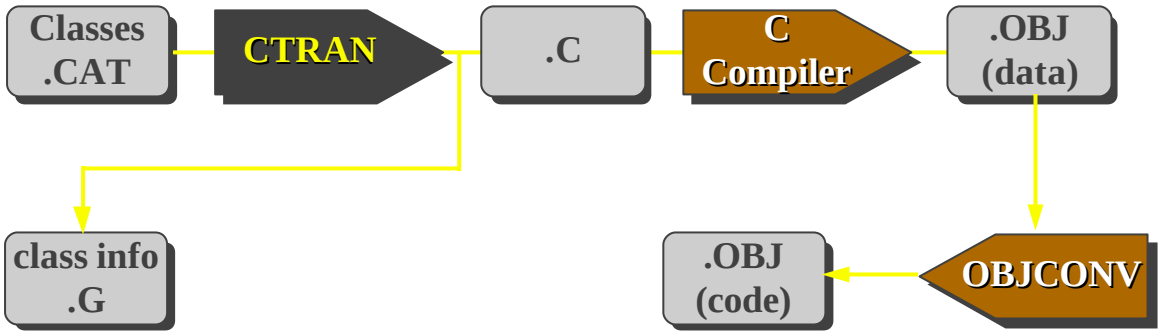
The application must also supply a method function for each additional or replacement method declared in the category file. For clarity, it is generally preferable to use a separate file for the source code of the method functions of each subclass. This has the added advantage that it also tends to reduce the amount of data in the included files, particularly if the category file is separated into a number of sub-category files (see *Appendix A - Category Files*). It is, however, perfectly acceptable to combine the method functions of two or more subclasses in a single file, particularly if they are closely related, or if they share some common functionality.

The process of generating a *.img* file is illustrated in the following diagram.



The category file is translated, producing an object file and one or more *.g* files. The *.g* files are included, as required, into the *.c* files containing the method functions. These files are compiled, to produce object files, in the same way as for any other *.c* file. The *.img* file is formed by linking these object files and the object file that results from category translation.

The processing of the category file is shown in more detail in the following diagram. The CTRAN category translator tool actually generates a *.c* file that contains, in source form, the data for the class descriptors of each class in the category. This file has to be compiled to produce the class descriptors in object form. Since class descriptors have to be in the code space of a process, the object file has to be further processed. This processing is performed by the OBJCONV tool, which modifies all data in the object file so that it will be loaded into the process code space when the application runs.



The entire process, from category file to converted object file, is performed by the supplied *ct.bat* batch file. You only need to be aware of the underlying mechanisms because of the .c file that is generated in this process. This file has the same name as the *.cat* file so that, for example, the category file *mycat.cat* will generate the file *mycat.c*. You should therefore avoid creating a separate C source file (a method source file, for example) with the same name as the application's category file, otherwise it will be overwritten during the category translation process.

An object oriented application (*.app* file) is constructed from a *.img* file by adding an icon, a shell data file and a resource file, in the same way as is described in the *Series 3 Programming Guide*.

An example application

This example application (the source of which, on installation, is copied into a *\sibosdk\oopdemo* directory) prints specified directory listings to the screen. It is based on the (non-object oriented) *p_prndir* example mentioned in the *Building an Application* chapter of the *General Programming Manual*. It uses object oriented techniques to store the file names in a variable array object. This adds value by allowing the names to be ordered so that the list can be displayed in alphabetical order, with all directory file names appearing first.

To avoid obscuring the main principles of building an object oriented application, this example makes minimal use of the object libraries built into the Series 3 and Series 3a. It does not, for example, use any of the user interface mechanisms provided by the HWIM library and is therefore not a typical example of an object oriented application. Its user interface uses the console device as used in many of the straight C examples (that is, using neither the HWIM or the HWIF libraries) given in, say, the *PLIB Reference* manual. The example described in the chapter *An HWIM Application - Hello World* makes use of the user interface objects and thus forms a better model for a real application.

The example source

The source for this application consists of three files:

`prndir.cat`	the `DIRLIST` object class definition
`dirlist.c`	the `DIRLIST` method functions
`dirmain.c`	`main()` and auxiliary functions

The category file, *prndir.cat*, is as follows:

```
IMAGE prndir
EXTERNAL olib
INCLUDE varray.g
INCLUDE p_file.h

CLASS dirlist vaxvar
Directory list
    {
    REPLACE va_test         sort by various criteria
    TYPES
        {
        typedef struct
            {
            P_INFO info;
            TEXT name[P_FNAMESIZE];
            } DIRLIST_ITEM;
        }
    }
```

The method function file, *dirlist.c*, contains only one method function, the replacement for the `va_test` method:

```
/*
DIRLIST
*/

#include <plib.h>
#include <prndir.g>

#pragma METHOD_CALL

METHOD INT dirlist_va_test(PR_DIRLIST *self,RC_VAXVAR *prec1,RC_VAXVAR *prec2)
/*
Firstly perform a 2 way test on file or dir name records.
Order dir names before file names.
Otherwise order names alphabetically.
Return 0 if equal, <0 if *prec1 is before *prec2, >0 if after.
*/
    {
    FAST DIRLIST_ITEM *p1,*p2;
    FAST INT ret;

    ret=0;
    p1=(DIRLIST_ITEM *)prec1->buf;
    p2=(DIRLIST_ITEM *)prec2->buf;
    if ((p1->info.status&P_FADIR)^(p2->info.status&P_FADIR))
        ret=(p1->info.status&P_FADIR)?-1:+1;
    else /* both records either dir or file names */
        ret=p_scmp(&p1->name[0],&p2->name[0]);
    if (self->varoot.key.desc)
        ret=(-ret); /* reverse result if required */
    return(ret);
    }
```

The file *dirmain.c*, listed below, uses the console device for obtaining keyboard input and displaying its output. Provided a valid directory name is typed in, the code builds, in an instance of the `DIRLIST` variable array class, a corresponding directory listing. The file names are stored and displayed in alphabetical order. Press `Enter` at the input prompt to exit the program.

```
/*
DIRMAIN
*/
#include <plib.h>
#include <prndir.g>

LOCAL_D VOID *dcb=NULL;
LOCAL_D VOID *hand;

LOCAL_C VOID error(TEXT *msg, INT errno)
    {
    TEXT bb[E_MAX_ERROR_TEXT_SIZE];

    p_close(dcb);
    dcb=NULL;
    p_errs(&bb[0],errno);
    p_printf("%s: %s",msg,&bb[0]);
    }

LOCAL_C VOID panic(TEXT *msg, INT errno)
    {
    error(msg,errno);
    p_leave(0);
    }
```

```
LOCAL_C VOID PrintDirLine(TEXT *name, P_INFO *pinfo)
    {
    P_DAYSEC ds;
    P_DATE dt;
    TEXT *p,b[40];

    p=&b[0];
    if (pinfo->status&P_FAVOLUME)
        p=p_scpy(p,"Vol,");
    if (pinfo->status&P_FADIR)
        p=p_scpy(p,"Dir,");
    if (pinfo->status&P_FAMOD)
        p=p_scpy(p,"Mod,");
    if (!(pinfo->status&P_FAWRITE))
        p=p_scpy(p,"Read,");
    if (pinfo->status&P_FASYSTEM)
        p=p_scpy(p,"Sys,");
    if (pinfo->status&P_FAHIDDEN)
        p=p_scpy(p,"Hid,");
    if (*(p-1)==',')
        *--p=0;
    p_sttods(&pinfo->modst,&ds);
    p_dstodt(&ds,&dt);
    p_printf("%- 12s %7lu %02u-%02u-%02u %02u:%02u %s",
            name,pinfo-
>size,dt.day+1,dt.month+1,dt.year,dt.hour,dt.minute,&b[0]);
    }

LOCAL_C VOID MakeDirList(TEXT *dir)
    {
    INT ret;
    DIRLIST_ITEM d;
    RC_VAXVAR rec;

    p_send2(hand,O_VA_RESET);
    if ((ret=p_open(&dcb,dir,P_FDIR))!=0)
        panic("Failed to open directory file",ret);
    while (!(ret=p_iow(dcb,P_FREAD,&d.name[0],&d.info)))
        {
        rec.buf=(UBYTE *)&d;
        rec.len=sizeof(d.info)+p_slen(&d.name[0])+1;
        p_send4(hand,O_VA_INSERTISQ,&rec,&i);
        }
    p_close(dcb);
    dcb=NULL;
    if (ret!=E_FILE_EOF)
        panic("Failed to read directory",ret);
    }

GLDEF_C INT CDECL DoDirLists(VOID)
    {
    UWORD i,count;
    DIRLIST_ITEM *pd;
    TEXT name[P_FNAMESIZE];

    hand=f_newsend(CAT_PRNDIR_PRNDIR,C_DIRLIST,O_VA_INIT,16);
    while (p_getl(">",&name[0],P_FNAMESIZE))
        {
        MakeDirList(&name[0]);
        if (!(count=p_send2(hand,O_VA_COUNT)))
            p_printf("No files found");
        else
            {
            for (i=0;i<count;i++)
                {
                pd=(DIRLIST_ITEM *)p_send3(hand,O_VA_PBUF,i);
                PrintDirLine(&pd->name[0],&pd->info);
                }
            }
        }
    p_send2(hand,O_DESTROY);
    return(0);
    }
```

```
GLDEF_C INT main(VOID)
    {
    INT err;

    p_linklib(0); /* Link to OLIB */
    if ((err=p_enter((VOID *)DoDirLists))!=0)
        error("Called p_leave",err);
    return(0);
    }
```

Note that a standard console application is not suitable for running on an EPOC emulator. See the code in the *Using the example DYL* secton of the *Building a Dynamic Library* chapter for how to adapt console code so that it is suitable to run on an emulator.

Building the example application

In order to illustrate the various stages in building an object oriented application, the description of the building of the example application makes use of a number of separate batch files. An alternative would be to make more use of the TopSpeed development environment and this approach is used in the building of the Hello World example, described in the chapter *An HWIM Application - Hello World*.

The first step in building the application is to translate the category file, *prndir.cat*, using the tool *ctran.exe*. This generates a number of files, including a *prndir.g* include file and a *prndir.c* C language source file. The generated *.c* file must be compiled and the resulting *.obj* file must be converted so that its class descriptor data is located in the code segment, using *ecobj.exe*. The entire process is conveniently performed by means of the supplied batch file, *ct.bat* (in *\sibosdk\sys*) whose content is:

```
@echo off
ctran %1 -e..\include -x..\include -g..\include -c -l -s -v
if errorlevel 1 goto end
call cc %1
ecobj %1
:end
```

The meaning of the various flags that can be passed to *ctran.exe* are explained in the *Category Translation* chapter of the *Object Oriented Programming Reference* manual.

Note that the above batch file is set up assuming that the source code is in a *\sibosdk\oopdemo* directory and that all include files are assumed to be found in a *..\include* directory (which is also the destination for include files generated by *ctran*). You must ensure that the current TopSpeed redirection file, *ts.red*, is set up to correspond with the location of all include files. In particular, the redirection file must include a line such as:

```
*.G   = .; ..\INCLUDE;
```

There is no need to include the location of generated *.ext* files in *ts.red*, since these are only referenced by *ctran*.

The remaining *.c* source files must be compiled as normal, for example, by using the same *cc.bat* file as is suitable for being called from *ct.bat*:

```
@echo off
call checkvid
tsc %1.c /fpunnamed /%jpivid%
```

This assumes the presence of an *unnamed.pr* project file of a similar form to that used when compiling non-object oriented programs, for example:

```
#system epoc img
#set epocinit=iplib
#model small jpi
#compile %main
#link %main
```

Finally the application must be linked. A suitable *lf.bat* link batch file is:

```
@echo off
call checkvid
tsc %1.pr /l /%jpivid%
```

which is called as:

```
lf prndir
```

assuming the presence of a *prndir.pr* file containing:

```
#system epoc img
#set epocinit=iplib
#model small jpi
#pragma link(olib.lib)
#pragma link(prndir)
#pragma link(dirlist)
#pragma link(dirmain)
#link prndir
```

This generates a *prndir.img* file which may be copied to a SIBO machine and executed as any other *.img* file. Note that, during the link process, two warning messages are always displayed, warning of duplicate tables. These messages should be ignored.

Note that *an object oriented application must be linked with the PLIB library* since none of the object oriented mechanisms are supported by CLIB.

The entire build process may be summarised in, say, a *bldpdir.bat* batch file (not supplied) as follows:

```
call ct prndir
call cc dirlist
call cc dirmain
lf prndir
```

CHAPTER 3

BUILDING A DYNAMIC LIBRARY

A dynamic library (DYL) is built from a category file and one or more method function source files in a similar way to the building of an application.

A DYL differs from an application in the following ways:

- it has no specific entry point and thus does not require a `main` function
- it may not contain any static data
- its category file should start with a LIBRARY statement
- it is linked with a different startup module

A DYL must be built with the PLIB (rather than CLIB) library, although you may, as usual, use a mix of PLIB and CLIB function calls. You should not use true floating point arithmetic in a DYL, but you may use the PLIB functions that avoid the 8087 emulator, described in the *Floating Point* chapter of the *PLIB Reference* manual.

An example DYL

This dynamic library (the source of which, on installation, is copied into a *\sibosdk\oopdemo* directory) contains one class, providing a method to sort an array of integers, using the quicksort service provided by PLIB.

The example source

The source for this application consists of two files:

sort.cat	the `ISORT` object class definition
isort.c	the `ISORT` method function

The category file, *sort.cat*, is as follows:

```
LIBRARY sort

EXTERNAL olib

INCLUDE olib.g

CLASS isort root
    {
    ADD sort            Sort a given list of integers
    }
```

The method function file, *isort.c*, contains only one method function:

```
/*
ISORT.C
*/

#include <sort.g>

#define IdataBuf ((INT *)DataBuf)

LOCAL_C INT OrdFunc(INT i,INT j,VOID *DataBuf)
/*
Order function used in qsort
*/
    {
    return(IdataBuf[i]-IdataBuf[j]);
    }

LOCAL_C VOID ExchFunc(INT i,INT j,VOID *DataBuf)
/*
Exchange function used in qsort
*/
    {
    INT k;

    k=IdataBuf[i];
    IdataBuf[i]=IdataBuf[j];
    IdataBuf[j]=k;
    }

#pragma METHOD_CALL

METHOD VOID isort_sort(PR_ROOT *self,INT *start,INT num)
    {
    p_qsort(num,OrdFunc,ExchFunc,start);
    }
```

Building the example DYL

As in the previous chapter, in order to illustrate the various stages, a number of separate batch files are used. Again, an alternative would be to use the more integrated approach as is used in the building of the Hello World example, described in the chapter *An HWIM Application - Hello World*.

The first step in building the DYL is to translate the category file, *sort.cat*, using the tool *ctran.exe*. This generates a number of files, including a *sort.g* include file and a *sort.c* C language source file.

As in the case of building an application, the generated *.c* file must be compiled and the resulting *.obj* file must be converted so that its class descriptor data is located in the code segment. The entire process is again conveniently performed by means of the same *ct.bat* batch file as is used for application category files. See the *Building an Object Oriented Application* chapter for further details.

The remaining *.c* source files (in this case, only *isort.c*) must be compiled as normal, using the same *cc.bat* file as is suitable for compiling application source files.

Finally the DYL must be linked. As for building an application, the link is controlled by a project file, in this case *sort.pr*:

```
#system epoc dyl
#set epocinit=iplib
#model small jpi
#pragma link(olib.lib)
#pragma link(sort)
#pragma link(isort)
#link sort
```

The significant difference between a project file for a DYL and that for an object oriented application is that the file type is declared as `dyl`, rather than `img`. Again, it is essential that the PLIB library be used.

You may use the same *lf.bat* link batch file as for linking applications, but you may wish to use the following *lfc.bat* variant:

```
@echo off
if exist %1.dyl del %1.dyl
call checkvid
tsc %1.pr /l /%jpivid%
```

This ensures that any failure in the link process does not result in an older version of the DYL being left. It is called as:

```
lfc sort
```

The entire build process may be summarised in a *dylbld.bat* batch file, as follows:

```
call ct sort
call cc isort
lfc sort
```

Using the example DYL

The following code, in *runsort.c*, illustrates a simple application that uses *sort.dyl* to sort the contents of an array of ten integers.

```
/*
RUNSORT.C
*/

#include <plib.h>
#include <sort.g>

GLREF_D VOID *DatCommandPtr;

GLDEF_D P_RECT _DefScreenRect;

LOCAL_D INT array[] = {10,1,5,7,9,3,6,8,4,2};

#pragma save,ENTER_CALL

LOCAL_C INT RunSort(HANDLE dyl)
    {
    VOID *sort;

    sort=f_newlibh(dyl,C_ISORT);
    p_send4(sort,O_SORT,&array[0],10);
    p_send2(sort,O_DESTROY);
    return(0);
    }

#pragma restore

GLDEF_C INT main(VOID)
    {
    INT err,i;
    HANDLE dyl;
    TEXT buf[P_FNAMESIZE];

    p_fparse("sort.dyl",DatCommandPtr,&buf[0],NULL);
    err=p_loadlib(&buf[0],&dyl,TRUE);
    if (!err)
        {
        _DefScreenRect.tl.x=0; /* set console window size */
        _DefScreenRect.tl.y=0;
        _DefScreenRect.br.x=40; /* 40 columns */
        _DefScreenRect.br.y=8;  /* 8 rows */

        err=p_enter2(RunSort,dyl);
        p_unloadlib(dyl);
        for (i=0;i<10;i++)
            p_printf("%d",array[i]);
        }
    return(err);
    }
```

In this example the DYL is loaded and linked by the call to `p_loadlib`. Since the DYL name is parsed with `DatCommandPtr`, *sort.dyl* will be expected to be found in the directory from which runsort is executed.

Reporting is via an automatically opened console window, whose size is set to be suitable for the Series 3 screen. Note that the code to set the console window size, using `_DefScreenRect`, is only necessary when you want to override the default size, or when the application is to be run on an emulator. If you include this code in an application, ignore the spurious warning of duplication that is given by the linker.

A DYL that supplies the ROOT class

Since OLIB contains classes that provide many basic services, most DYLs will either reference or subclass one or more OLIB classes. They will therefore need to declare an external reference to OLIB, as in the previous example.

In that example, however, the external reference is necessary only because the `ISORT` class subclasses the `ROOT` class provided by the OLIB dynamic library. In such a case it may be more reasonable for a DYL to define its own `ROOT` class and be independent of OLIB. The following sample code provides the same integer sorting functionality as the previous example, but without requiring an external reference to OLIB.

The category file, *rsort.cat*, is as follows:

```
LIBRARY rsort

INCLUDE p_std.h
INCLUDE p_object.h

CLASS root
    {
    ADD destroy
    ADD sort            Sort a given list of integers
    PROPERTY
        {
        P_OBJECT pc;    Class link
        }
    }
```

The class definition of `ROOT` duplicates that of the OLIB `ROOT` class with, in this case, the addition of the `sort` method.

The method function source file, *root.c* is almost identical with that of *isort.c*, the only significant difference being that the method function is renamed to `root_sort`.

```
/*
ROOT.C
*/

#include <rsort.g>

#define IdataBuf ((INT *)DataBuf)

LOCAL_C INT OrdFunc(INT i,INT j,VOID *DataBuf)
/*
Order function used in qsort
*/
    {
    return(IdataBuf[i]-IdataBuf[j]);
    }

LOCAL_C VOID ExchFunc(INT i,INT j,VOID *DataBuf)
/*
Exchange function used in qsort
*/
    {
    INT k;

    k=IdataBuf[i];
    IdataBuf[i]=IdataBuf[j];
    IdataBuf[j]=k;
    }

#pragma METHOD_CALL

METHOD VOID root_sort(PR_ROOT *self,INT *start,INT num)
    {
    p_qsort(num,OrdFunc,ExchFunc,start);
    }
```

Note that there is no need to provide the `root_destroy` method function, since this is supplied by the PLIB library.

Building the DYL follows exactly as in the previous example, using the link project file, *rsort.pr*:

```
#system epoc dyl
#set epocinit=iplib
#model small jpi
#pragma link(olib.lib)
#pragma link(rsort)
#pragma link(root)
#link rsort
```

Building DYLs into an application

One or more DYLs may be combined into a *.img* or a *.app* file. The technique is similar to the add-file technology that can combine a resource file, an icon and a shell data file with a *.img* file. A significant difference is that whereas add-files are limited to a maximum of four add-files, there is no limit on the number of DYLs that may be added.

The main advantage of building DYLs into an application is that there is then no danger of the various files becoming separated, or of an essential DYL being accidentally deleted. There can never be any confusion over the location of a DYL and if the application is present, its DYLs must also be present.

DYL add-file lists

A DYL add-file list is a text file with a *.dfl* extension, containing a list of the names of the DYLs that are to be combined with a *.img* file. For example, the Series 3a Spreadsheet has a *.dfl* file with the content:

```
shfl.dyl
shgf.dyl
shgp.dyl
shdb.dyl
shta.dyl
shbr.dyl
shpr.dyl
shrg.dyl
shvw.dyl
shso.dyl
```

to build ten DYLs into the Spreadsheet application.

When any *.pr* project file is invoked that leads to the building of a *.img* file, a check is made for the existence of a *.dfl* file with the same name as the application. If this file exists, the DYLs it lists are automatically built into the *.img* file, in the order in which they are listed.

Accessing a built-in DYL

A DYL that is built into an application is accessed by use of the functions `p_openlib` and `p_loadfilelib`., as in the following example. The required DYL is specified in a call to `p_loadfilelib` by an index, counting from zero, where the numbering order is determined by the order in which the DYLs are listed in the *.dfl* file.

The example code, *dylsort.c*, is a variant of *runsort.c*, described earlier. It has exactly the same action as the earlier example, but *sort.dyl* is built into the resulting *.img* file, rather than being a separate file.

```
/*
DYLSORT.C
*/

#include <plib.h>
#include <sort.g>

GLREF_D VOID *DatCommandPtr;

GLDEF_D P_RECT _DefScreenRect;

LOCAL_D INT array[] = {10,1,5,7,9,3,6,8,4,2};

#pragma save,ENTER_CALL
```

```
LOCAL_C INT RunSort(HANDLE dyl)
    {
    VOID *sort;

    sort=f_newlibh(dyl,C_ISORT);
    p_send4(sort,O_SORT,&array[0],10);
    p_send2(sort,O_DESTROY);
    return(0);
    }

#pragma restore

GLDEF_C INT main(VOID)
    {
    INT err,i;
    HANDLE dyl;
    VOID *dcb;

    p_openlib(&dcb,DatCommandPtr); /* open application .img file for DYL
access */
    err=p_loadfilelib(dcb,0,&dyl,TRUE); /* load the first (and only) DYL */
    if (!err)
        {
        _DefScreenRect.tl.x=0;
        _DefScreenRect.tl.y=0;
        _DefScreenRect.br.x=40;
        _DefScreenRect.br.y=8;

        p_printf("Sorting...");
        err=p_enter2(RunSort,dyl);
        p_unloadlib(dyl);
        for (i=0;i<10;i+=2)
            p_printf("%4d %4d",array[i],array[i+1]);
        p_getch();
        }
    return(err);
    }
```

The only difference in the code between this example and *runsort.c* is in the first two lines of `main()`. In contrast with the earlier example, there is no need to parse the application's full file specification (pointed to by `DatCommandPtr`) with the DYL file name, since the file containing the DYL *is* the application file itself.

The DYL add-file list is *dylsort.dfl*, which contains the single line:

```
sort.dyl
```

The application can be built in a similar way to runsort, that is, by:

```
cc dylsort
lf dylsort
```

Running *edump.exe*, by typing:

```
edump dylsort
```

produces the following output, showing the presence of the built-in *sort.dyl*.

```
EDump V4.30F (09/11/93) Copyright (C) Psion PLC 1989-92
LOC::E:\SIBOSDK\OOPDEMO\DYLSORT.IMG IMAGE file data
Image version      = 200F
Code Segment       = 0330 (bytes)
Initial IP         = 0000
Stack              = 1000 (bytes)
Data               = 0070 (bytes)
Heap               = 0800 (bytes)
Data Segment       = 1870 (bytes)
Initialized data   = 0050 (bytes)
Code checksum      = 765A
Data checksum      = 06B7
Code Version       = 100F
Priority           = 0080
Header size        = 0040 (bytes)
Dyl count          = 0001
Dyl table offset   = 000005E0
Dyl 00       SORT.DYL offset = 000003C0
Image file size    = 000005F2 (bytes)
```

CHAPTER 4

AN HWIM EXAMPLE - HELLO WORLD

This chapter describes a minimal HWIM application. The application displays a bordered window containing the text "Hello world" and has a menu bar that offers a single Exit option. It is written so that it will run on either the Series 3 or the Series 3a (in compatibility mode).

Note that this application is not intended to exercise the full potential of OOP. Instead, it gives a "feel" for the construction of an OOP application and, while its broad structure will be described, a full understanding may not be apparent until later chapters in this manual have been read. The example code does, however, provide the basic framework of all HWIM applications and may be used as a starting point for the construction of more complex applications.

The source of this application is supplied in the *\sibosdk\oopdemo* directory that can be installed from the C SDK disks. The main source files for the application are listed below, and are described in more detail in the following sections.

Category file, *hello.cat*

```
IMAGE hello

EXTERNAL olib
EXTERNAL hwim

INCLUDE hwimman.g

CLASS   hellows   wserv
window server active object
    {
    REPLACE ws_dyn_init
    }

CLASS   hellobw   bwin
a simple bordered window
    {
    REPLACE wn_init
    REPLACE wn_draw
    }
```

Resource file, *hello.rss*

```
/*
    HELLO.RSS

English resource file for Hello World application
*/

#include <hwim.rh>
#include <hello.rg>


RESOURCE WSERV_INFO hello_accs
    {
    menbar_id=hello_menbar;
    first_com=O_COM_EXIT;
    accel={'x'};    /* Exit */
    }
```

```
RESOURCE MENU_BAR hello_menbar
    {
    items=
        {
        MENU_BAR_ITEM
            {
            menu_id=special_menu;
            mb_item="Special";
            }
        };
    }

RESOURCE MENU special_menu
    {
    items =
        {
        MENU_ITEM
            {
            com_id=O_COM_EXIT;
            mn_item="Exit";
            }
        };
    }
```

Source code, *o_hello.c*

```
/*
O_HELLO.C
*/

#include <hwimman.g>
#include <hello.g>

GLREF_D WSERV_SPEC *wserv_channel;

GLDEF_C VOID main(VOID)
    {
    IN_HWIMMAN app;
    IN_WSERV ws;

    p_linklib(0);
    app.flags=FLG_APPMAN_RSCFILE|FLG_APPMAN_SRSCFILE|FLG_APPMAN_CLEAN;
    app.wserv_cat=p_getlibh(CAT_HELLO_HELLO);
    ws.com_cat=p_getlibh(CAT_HELLO_HWIM);
    app.wserv_class=C_HELLOWS;
    ws.com_class=C_COMMAN;
    p_send4(p_new(CAT_HELLO_HWIM,C_HWIMMAN),O_AM_INIT,&app,&ws);
    }

#pragma METHOD_CALL

METHOD VOID hellows_ws_dyn_init(PR_HELLOWS *self)
    {
    self->wserv.cli=f_new(CAT_HELLO_HELLO,C_HELLOBW);
    p_send2(self->wserv.cli,O_WN_INIT);
    }

METHOD VOID hellobw_wn_init(PR_HELLOBW *self)
    {
    W_WINDATA wd;

    wd.extent.tl.x=0;
    wd.extent.tl.y=0;
    wd.extent.width=wserv_channel->conn.info.pixels.x;
    wd.extent.height=wserv_channel->conn.info.pixels.y;
    p_send5(self,O_WN_CONNECT,NULL,W_WIN_EXTENT,&wd);
    p_send3(self,O_WN_VISIBLE,WV_INITVIS);
    p_send3(self,O_WN_EMPHASISE,TRUE);
    }
```

```
METHOD VOID hellobw_wn_draw(PR_HELLOBW *self)
    {
    p_supersend2(self,O_WN_DRAW);
    gPrintText(50,50,"Hello World",11);
    }
```

The category file

The application's category file is as follows:

```
IMAGE hello

EXTERNAL olib
EXTERNAL hwim

INCLUDE hwimman.g

CLASS   hellows   wserv
window server active object
    {
    REPLACE ws_dyn_init
    }

CLASS   hellobw   bwin
a simple bordered window
    {
    REPLACE wn_init
    REPLACE wn_draw
    }
```

The `IMAGE` statement identifies the file as being one that will create a category for an application (*.img* or *.app*) file. Note that the external references to the OLIB and HWIM DYLs are mandatory for all HWIM applications.

The category file defines two application-specific classes, `HELLOWS` and `HELLOBW`. These are subclasses of the HWIM `WSERV` window server active object and `BWIN` bordered window classes respectively. These two subclasses add no property or new methods; they just replace methods that are defined in a superclass.

The `BWIN` class is described in more detail in the *Windows* chapter.

The resource externals file

The resource file, described in the next section, contains a reference to the `O_COM_EXIT` method number. This symbol is defined in the *hwimman.g* include file (itself generated by the translation of the category containing the `hwimman` class).

By building the resource externals file, with file name *hello.re*, containing the following lines:

```
#include <hwimman.g>

_O_COM_EXIT
```

The file *hello.rg* is generated by the *re.bat* batch file by typing:

```
re hello
```

and contains the single line:

```
#define O_COM_EXIT 7
```

This process has extracted the definition of the symbol `O_COM_EXIT` from the *hwimman.g* include file, so that the generated *.rg* file can be `#included` in the application resource file instead of the larger *hwimman.g*.

The resource file

The resource file, *hello.rss*, is one of the simplest possible HWIM application resource files:

```
/*
    HELLO.RSS

English resource file for Hello World application
*/

#include <hwim.rh>
#include <hello.rg>


RESOURCE WSERV_INFO hello_accs
    {
    menbar_id=hello_menbar;
    first_com=O_COM_EXIT;
    accel={'x'};    /* Exit */
    }

RESOURCE MENU_BAR hello_menbar
    {
    items=
        {
        MENU_BAR_ITEM
            {
            menu_id=special_menu;
            mb_item="Special";
            }
        };
    }

RESOURCE MENU special_menu
    {
    items =
        {
        MENU_ITEM
            {
            com_id=O_COM_EXIT;
            mn_item="Exit";
            }
        };
    }
```

In addition to *hello.rg*, it includes the HWIM resource header file, *hwim.rh*, that defines the standard resource structures used by HWIM applications (for example, the `WSERV_INFO` resource structure). This resource file contains three resources that must be present in all HWIM application resource files, being used by the HWIM command menu mechanism.

The first resource must always be a `WSERV_INFO` resource structure. Its reference name (in this case, `hello_accs`) is normally not relevant. The `menbar_id` element must refer to a following `MENU_BAR` resource structure that defines the content of the application's menu bar, and `first_com` is set to the method number of a method of the application's command manager (usually, as in this case, `O_COM_EXIT`). The `accel` element defines one or more accelerator keypresses. Each accelerator may be used to invoke a command menu option by executing a command manager method function. In this example there is only one accelerator, Psion-X, which executes the command manager's exit method (with method number `O_COM_EXIT`).

The `MENU_BAR` resource defines a menu bar containing a single menu, with menu name "Special", and an associated pull-down menu defined in the `MENU` resource structure referenced by `special_menu`. This pull-down menu contains only the single menu option "Exit".

The source code

The code is sufficiently brief that there is no advantage in writing it as a number of separate C modules. All the code is in the file *o_hello.c* (so named to distinguish it from the *hello.c* that is generated during the translation of *hello.cat*).

```
/*
O_HELLO.C
*/

#include <hwimman.g>
#include <hello.g>

GLREF_D WSERV_SPEC *wserv_channel;

GLDEF_C VOID main(VOID)
    {
    IN_HWIMMAN app;
    IN_WSERV ws;

    p_linklib(0);
    app.flags=FLG_APPMAN_RSCFILE|FLG_APPMAN_SRSCFILE|FLG_APPMAN_CLEAN;
    app.wserv_cat=p_getlibh(CAT_HELLO_HELLO);
    ws.com_cat=p_getlibh(CAT_HELLO_HWIM);
    app.wserv_class=C_HELLOWS;
    ws.com_class=C_COMMAN;
    p_send4(p_new(CAT_HELLO_HWIM,C_HWIMMAN),O_AM_INIT,&app,&ws);
    }

#pragma METHOD_CALL

/* APPLICATION-SPECIFIC WINDOW SERVER OBJECT */

METHOD VOID hellows_ws_dyn_init(PR_HELLOWS *self)
    {
    self->wserv.cli=f_new(CAT_HELLO_HELLO,C_HELLOBW);
    p_send2(self->wserv.cli,O_WN_INIT);
    }

/* BORDERED CLIENT WINDOW */

METHOD VOID hellobw_wn_init(PR_HELLOBW *self)
    {
    W_WINDATA wd;

    wd.extent.tl.x=0;
    wd.extent.tl.y=0;
    wd.extent.width=wserv_channel->conn.info.pixels.x;
    wd.extent.height=wserv_channel->conn.info.pixels.y;
    p_send5(self,O_WN_CONNECT,NULL,W_WIN_EXTENT,&wd);
    p_send3(self,O_WN_VISIBLE,WV_INITVIS);
    p_send3(self,O_WN_EMPHASISE,TRUE);
    }

METHOD VOID hellobw_wn_draw(PR_HELLOBW *self)
    {
    p_supersend2(self,O_WN_DRAW);
    gPrintText(50,50,"Hello World",11);
    }
```

Main

The `main()` function is basically as specified in the *Introduction* chapter. Note that, since the window server active object is subclassed by the application, it is indicated to be in the local category (category number `CAT_HELLO_HELLO`). In contrast, the application uses the basic command manager supplied by HWIM and thus in the HWIM external category (category number `CAT_HELLO_HWIM`).

The method functions are preceded by:

```
#pragma METHOD_CALL
```

to ensure the correct calling convention.

Window server object

The `ws_dyn_init` method supplied for the `HELLOWS` class simply creates an instance of the `HELLOBW` bordered window subclass and sends it a `WN_INIT` message. The window is set to be the application's client window by writing its handle to the window server active object's `wserv.cli` property.

Client window

Initialisation of the bordered client window sets the window's size to the screen dimensions, as read from the `WSERV_SPEC` struct pointed to by the magic static `wserv_channel`. Most application-specific window classes will supply a `wn_init` method that, at some stage, sends a `WN_CONNECT` message. Normally, as in the present case, the client window is a top-level window (it has no parent) as indicated by the `NULL` parameter to the `WN_CONNECT` message.

The window is made visible and set to the emphasised state by sending `WN_VISIBLE` and `WN_EMPHASISE` messages. These messages are processed by superclass method functions (described in the *Windows* chapter of this manual).

The `wn_draw` method is not explicitly called by application code, but will be called directly from system code whenever the window must be drawn (such as when it first becomes visible). See also the *The draw/redraw mechanism* section of the *Windows* chapter.

After supersending the `WN_DRAW` message (handled by the `BWIN` superclass, to draw the window's border) the text "Hello World" is drawn, using the window server function `gPrintText`.

Building the application

Unlike the simple examples in earlier chapters, the "Hello World" example provides a good basic model for most object oriented applications written for the Series 3 and Series 3a machines. The mechanism supplied for building this example, described below, is suitable for use with any such application. It uses the TopSpeed *Make mode* to invoke the project system, in order to minimise the amount of compilation and linking required to make sure that the *.img* file is up-to-date.

The *.pr* file for the "Hello World" application is *hello.pr* and is listed below.

```
#system epoc img
#set epocinit=iplib
#model small jpi

#abort on

#set version=0x100F
#set priority=0x80
#set heapsize=0x80

#compile %main.cat

#if %remake #or (%main.rg #older %main.re) #or (%main.rg #older %main.g) #then
  #run "re %main" no_window no_abort
#endif

#if (%main.rsg #older %main.rss) #or (%main.rsg #older %main.rg) #then
  #run "rs %main" no_window no_abort
#endif

#if %main.rzc #older %main.rsg #then
  #run "rch %main" no_window no_abort
#endif

#compile o_hello.c

#if %main.shd #older %main.ms #then
  #run "makeshd %main" no_window no_abort
  #file delete %main.img
#endif

#if (%main.img #older %main.afl) #or (%main.img #older %main.pic) #then
  #file delete %main.img
#endif

#pragma link(olib.lib)
#pragma link(hwim.lib)

#link %main
```

The TopSpeed project system takes account of the dependencies in the application's C source files but, as can be seen in this example, the *.pr* file must explicitly take into account the dependencies of all other files that are needed to build the application.

The *hello.pr* project file is suitable for use as a model for building any object oriented application using the HWIM library. In the vast majority of cases the main change will be to replace the line:

```
#compile %o_hello.c
```

with one or more `#compile` statements to compile the various C source files that contain the application's code.

In this example the variables `version`, `priority` and `heapsize` are set (as hexadecimal numbers) to their default values - they will be given these values automatically if they are not set in the *.pr* file. Only in exceptional circumstances will you need to alter the application's priority, but you should set both the version number and the heap size as a matter of course. See the *General System Services* chapter of the *PLIB Reference* manual for a description of the format of a version number. The heap size is measured in paragraphs (16-byte units) and should normally be set to a value that is not less than the amount of the heap used when the application is started up (one way of determining this is to use the Spy application, described in the *Series 3/3a Programming Guide*). There is more information about the use of these three variables in the *Building an Application* chapter of the *General Programming Manual*.

The version of *tsprj.txt* supplied with the SDK contains a compiler entry to handle the translation of category files. It is therefore important to copy this file into your *\ts\sys* directory and run *tscfg.exe*, as described in the *Installation* chapter of the *General Programming Manual*, even if you are upgrading from an earlier version of the SDK. This entry, which is listed below, takes into account the creation dates of the category file and the *.g* file that is generated from it when determining if the category file needs translating.

```
#declare_compiler cat=
'#split %%src
 #set make=%%remake
 #if %%make #or %%name.g #older %%name.cat #then
  #run "ctran %%name -c -s -v -l -e..\include\ -x..\include\ -g..\include\"
no_window no_abort
  #rundll TSC %%name.c %%name.obj
  #run "ecobj %%name.obj" no_window no_abort
 #endif
 #pragma link(%%name.obj)'
```

If you include application-specific header files in the application's category file you will need to add an explicit check in your *.pr* file to ensure that the application is rebuilt properly. If, for example, your category file contains the line:

```
INCLUDE myheader.h
```

you will need to insert the lines:

```
#if myheader.h #older %main.cat #then
  #file delete %main.g
#endif
```

immediately before the line:

```
#compile %main.cat
```

Apart from the `#pragma link` statements to include the standard libraries (OLIB and HWIM) the remainder of the *.pr* file takes care of the dependencies of the application on the remaining component files in a fairly straightforward way.

If you develop an application that contains built-in DYLs you will need to add further checks on the creation dates of the *.dfl* DYL add-file list and the DYLs that it lists. A suitable check for the *.dfl* file would be:

```
#if %main.img #older %main.dfl #then
  #file delete %main.img
#endif
```

and a check for *mydyl.dyl* is:

```
#if %main.img #older mydyl.dyl #then
  #file delete %main.img
#endif
```

Any such lines should be inserted immediately before the first `#pragma link` statement.

The "Hello World" application is built by typing:

```
make hello
```

This makes use of a *make.bat* batch file, whose contents is as follows:

```
@echo off
call checkvid
if not exist %1.pr goto error
tscx /m %1 /%jpivid%
tscx /m %1 /%jpivid%
goto end
:error
echo Project file %1.pr does not exist
:end
```

Note that this batch file uses `tscx`, rather than `tsc`, so that full use may be made of the extended memory of your PC. If you do not have extended memory, you should replace each occurrence of `tscx` with `tsc`.

The batch file uses two passes of the TopSpeed project system. This is necessary because the project system does not take into account dependencies that change dynamically as a result of the 'compilation' of the Psion-specific source files. This will not normally result in any unnecessary repetitions of compilation or linking.

The result of typing `make hello` is the creation of *hello.img*. The final step in producing the "Hello World" application is (optionally) to rename *hello.img* as *hello.app*.

Variants

You may like to experiment with making the following variations to the supplied example code:

- To make the application take advantage of the larger screen size on the Series 3a, `OR` the flag `FLG_APPMAN_FULLSCREEN` into the flag values assigned to `app.flags` in `main()`.
- In the `HELLOWS` `ws_dyn_init` method, the lines:

  ```
  self->wserv.cli=f_new(CAT_HELLO_HELLO,C_HELLOBW);
  p_send2(self->wserv.cli,O_WN_INIT);
  ```

  could be replaced by the more compact:

  ```
  self->wserv.cli=f_newsend(CAT_HELLO_HELLO,C_HELLOBW,O_WN_INIT);
  ```

- In the `HELLOBW` `wn_init` method, replace

  ```
  p_send3(self,O_WN_VISIBLE,WV_INITVIS);
  ```

  with the equivalent, but more compact:

  ```
  hInitVis(self);
  ```

- Modify the bordered window's `wn_draw` method to centre the text in the window, in a way that works for both the Series 3 and Series 3a, by reading the screen dimensions from `wserv_channel->conn.info.pixels`.
- Put the "Hello World" text in the resource file and load it from there (for guidance, see the examples in the *Commands and Command Menus* chapter).

CHAPTER 5

COMMANDS AND COMMAND MENUS

The command manager

The command manager is created and initialised by system code during the start-up process, immediately before the window server active object receives a `WS_DYN_INIT` message. If the creation of the command manager fails the application process will terminate immediately. If the command manager is created successfully, its handle is accessible via the handle of the window server active object, held in the magic static `w_ws`. This static should be declared as[1]:

```
GLREF_D PR_WSERV *w_ws;
```

and the command manager's handle is then `w_ws->wserv.com`. See also the utility function `hWservComSend` that is used by system code to send messages to the command manager.

The HWIM `COMMAN` class definition (whose associated generated header file is *comman.g*) is as follows:

```
CLASS   comman  root
Superclass of all command managers
    {
    ADD com_init=p_dummy        User's own initialisation
    ADD com_statwin=p_dummy     Toggle permanent status window
    ADD com_accl_check=p_true   Called whenever an accelerator is matched
    ADD com_menu=p_dummy        Called whenever a menu is pulled down
    ADD com_mode_change=p_dummy W_KEY_MODE received
    ADD com_file_change=p_dummy The core code for open or new
    ADD com_exit                Message sent here on exit accelerator
    CONSTANTS
        {
        O_COM_SYS_LAST          O_COM_EXIT
        }
    }
```

The supplied methods of the command manager are called by system code. As is apparent from the class definition, the supplied method functions do very little. Although there is no need to replace any of these methods, they are intended to be replaced, as necessary, in an application-specific subclass. The supplied `com_exit` method, for example, simply calls `p_exit(0)`. While this is sufficient for a simple application, the method will usually be replaced - particularly if the application is file-based. The uses of most of these methods, and the circumstances under which the corresponding messages are sent by system code are described later in this chapter. The `com_file_change` and `com_exit` methods are described in the *File Based Applications* chapter.

[1]Depending on the application, it may be more appropriate to declare `w_ws` as a pointer to an instance of an application-specific subclass of `WSERV`.

Adding command options

Any HWIM application that has a more complex set of commands than that of the "Hello World" example applicaton will need to supply more extensive menu bar and pull-down menu resources. It will also need to subclass the `COMMAN` command manager class.

The basic mechanism for extending the number of command options involves the following steps:

- Extend the menu bar and pull-down menu resources in the application resource file to include a menu item for each command.
- Subclass the command manager to add a new method for each additional command and supply a method function for each method.[2] The methods must appear consecutively in the command manager's class definition and normally must immediately follow the last (`com_exit`) method of the `COMMAN` superclass.
- Include an accelerator in the first resource in the application resource file for each additional command. There must be an accelerator for every command option that appears in the menus, and the accelerators must appear in the same order as the method declarations in the command manager class. Note, however, that there need be no relation between the order of the accelerators and the order in which the commands appear in the command menus. The Exit option, for example, conventionally appears as the last item in the last menu, although it is normally first in the list of accelerators.

The sending of the appropriate message to the command manager when a command menu option is selected, either by means of an accelerator keypress or via the pull-down menus, is handled by system code in the window server active object and requires no additional application-specific code. The method functions that correspond to the menu options are not expected to return a value and should be declared as `VOID` functions. In general these functions do not take parameters, but may optionally make use of a single `INT` parameter, containing the function's method number, this being passed in system-generated messages (see the later *Sharing method function code* section).

Example

The following example extends the "Hello World" example to provide two additional command options in a separate pull-down menu. The additional commands switch the display to one of two alternative messages. The source code is supplied in the files *hello2.rss*, *hello2.cat* and *o_hello2.c*.

The resource file is as follows:

```
/*
    HELLO2.RSS

English resource file

*/

#include <hwim.rh>
#include <hello2.rg>

RESOURCE WSERV_INFO hello_accs
    {
    menbar_id=hello_menbar;
    first_com=O_COM_EXIT;
    accel={'x',    /* Exit */
           'h',    /* Hello */
           'b'};   /* Bye */
    }
```

[2]Note that, if appropriate, several commands may share the code of a single method function. This alternative is described later.

```
RESOURCE MENU_BAR hello_menbar
    {
    items=
        {
        MENU_BAR_ITEM
            {
            menu_id=message_menu;
            mb_item="Message";
            },
        MENU_BAR_ITEM
            {
            menu_id=special_menu;
            mb_item="Special";
            }
        };
    }

RESOURCE MENU message_menu
    {
    items =
        {
        MENU_ITEM
            {
            com_id=O_HCM_HELLO;
            mn_item="Say hello";
            },
        MENU_ITEM
            {
            com_id=O_HCM_BYE;
            mn_item="Say goodbye";
            }
        };
    }

RESOURCE MENU special_menu
    {
    items =
        {
        MENU_ITEM
            {
            com_id=O_COM_EXIT;
            mn_item="Exit";
            }
        };
    }

RESOURCE STRING hello_message {str="Hello World";}

RESOURCE STRING bye_message {str="Goodbye";}
```

The list of accelerators contains three items, and the menu bar has two pull-down menus; a 'Message' menu and a 'Special' menu. The 'Message' menu contains the two options 'Say hello' and 'Say goodbye', associated with the command manager methods hcm_hello and hcm_bye respectively. The two text messages to be displayed are included in the resource file, rather than appearing as data in the source code.

The category file defines a `HELLO2` category (the `IMAGE` statement is `IMAGE hello2`) that contains an additional class definition for the `HELLOCM` class - a subclass of the HWIM `COMMAN` class:

```
CLASS hellocm comman
command manager
    {
    ADD hcm_hello
    ADD hcm_bye
    }
```

Note that the order of declaration of the methods must agree with the order of the associated accelerators in the resource file.

The corresponding command manager method functions are:

```
METHOD VOID hellocm_hcm_hello(PR_HELLOCM *self)
    {
    p_send3(w_ws->wserv.cli,O_WN_SET,FALSE);
    }

METHOD VOID hellocm_hcm_bye(PR_HELLOCM *self)
    {
    p_send3(w_ws->wserv.cli,O_WN_SET,TRUE);
    }
```

Both of these methods send a `WN_SET` message to the client window, whose handle is available as `w_ws->wserv.cli` (similar to accessing the command manager itself, as described earlier). Note that these actions correspond to a design decision that the client window should itself be responsible for recording which message to display.

The client window class thus needs to replace the `wn_set` method and specify an item of property to record the message state, as follows:

```
CLASS   hellobw   bwin
bordered client window
    {
    REPLACE wn_init
    REPLACE wn_draw
    REPLACE wn_set
    PROPERTY
        {
        UWORD isbye;
        }
    }
```

The client window `wn_draw` method uses the state of its `isbye` property to determine which of the two text resources to display:

```
METHOD VOID hellobw_wn_draw(PR_HELLOBW *self)
    {
    UINT resid;
    TEXT buf[40];

    resid=self->hellobw.isbye ? BYE_MESSAGE:HELLO_MESSAGE;
    p_send4(w_am,O_AM_LOAD_RES_BUF,resid,&buf[0]); /* load the resource */
    p_supersend2(self,O_WN_DRAW);                  /* draw the border */
    gPrintText(50,50,&buf[0],p_slen(&buf[0]));
    }
```

Note that the line:

```
p_send4(w_am,O_AM_LOAD_RES_BUF,resid,&buf[0]);
```

could be replaced by the following use of the `hLoadResBuf` utility function:

```
hLoadResBuf(resid,&buf[0]);
```

The `wn_set` method sets or clears this property and causes the window to be drawn with the appropriate message:

```
METHOD VOID hellobw_wn_set(PR_HELLOBW *self,UINT isbye)
    {
    self->hellobw.isbye=isbye;
    p_send2(self,O_WN_DODRAW);
    }
```

Note that the application does not explicitly write to the isbye property on initialisation of the window. It takes advantage of the fact that property is zero filled when an object is created.

Essentially the same effect could be achieved by replacing the line:

```
p_send2(self,O_WN_DODRAW);
```

with the window server call:

```
wInvalidateWin(self->win.id);
```

This would result in the client window receiving, at some future time, a WN_REDRAW message. This technique might prove useful if two or more things could change that require the window to be redrawn. If all such changes invalidate the window, there is less likelihood that each change will cause the window to be redrawn.

In this example the difference between the two alternatives is not noticeable. Invalidating the window causes inter-process messages to be sent between the application and the window server, whereas the sending of the WN_DODRAW message does not and will, in general, result in the window being updated more responsively. Which of these two techniques to use depends to a large extent on the requirements of a particular application.

The code of main() must also be modified slightly to take into account the use of an application-specific command manager:

```
GLDEF_C VOID main(VOID)
    {
    IN_HWIMMAN app;
    IN_WSERV ws;

#ifndef EPOC
    GLREF_D P_DEVICE p_file;
    GLREF_D P_DEVICE p_serial;
    p_inst(&p_file,&p_serial,NULL_D);
#endif
    p_linklib(0);
    app.flags=FLG_APPMAN_RSCFILE|FLG_APPMAN_CLEAN|FLG_APPMAN_SRSCFILE;
    app.wserv_cat=p_getlibh(CAT_HELLO2_HELLO2);
    ws.com_cat=p_getlibh(CAT_HELLO2_HELLO2);
    app.wserv_class=C_HELLOWS;
    ws.com_class=C_HELLOCM;
    p_send4(p_new(CAT_HELLO2_HWIM,C_HWIMMAN),O_AM_INIT,&app,&ws);
    }
```

As before, add the flag FLG_APPMAN_FULLSCREEN to app.flags to take advantage of the larger screen on the Series 3a.

Sharing method function code

All messages sent by system code to the command manager as the result of selecting a menu option or using an accelerator (that is, the com_exit method and any following methods added by a subclass) are sent by a call to the utility function hWservComSend. The code for this utility function is effectively as follows:

```
VOID hWservComSend(INT comid);
    {
    p_send3(w_ws->wserv.com,comid,comid);
    }
```

Note that the parameter comid, which is the appropriate method number, is passed as an additional parameter and is thus available to the method function. This can be used to advantage when two or more methods have very similar method functions, as do the two command manager methods introduced in the previous example. The command manager class definition could have been written as:

```
CLASS hellocm comman
command manager
    {
    ADD hcm_hello
    ADD hcm_bye=hellocm_hcm_hello
    }
```

so that both methods share the same method function. The code could then be written as:

```
METHOD VOID hellocm_hcm_hello(PR_HELLOCM *self, INT comid)
    {
    if (comid==O_HCM_HELLO)
        p_send3(w_ws->wserv.cli,O_WN_SET,FALSE);
    else
        p_send3(w_ws->wserv.cli,O_WN_SET,TRUE);
    }
```

Clearly, there is no great advantage in this particular case, since the two methods are so simple. This is aggravated by the fact that the replacement code is written for clarity rather than compactness. Nevertheless, the technique can frequently be used to advantage in real applications.

Changing the text of an option

The options to set the message text in the previous example could be replaced by a single option that toggles between the two messages. To be effective, the text of the option should also change, depending on which message is currently visible.

Each time a pull-down menu is displayed, its content is reloaded from the resource file. The command manager's `com_menu` method is called immediately before a pull-down menu is displayed, but after the menu's text has been loaded.A command manager subclass may therefore replace this method to change the text of one or more items.

Continuing from the previous (hello2) example, we can use a single method instead of the two methods `com_hello` and `com_bye`. The class definition of `HELLOCM` thus becomes:

```
CLASS hellocm comman
command manager
    {
    REPLACE com_menu
    ADD hcm_hello
    }
```

The resource file needs to be changed to remove an accelerator and the menu item for the 'Say goodbye' option. An additional string resource is required to specify the alternative text for the option. The relevant resources become:

```
RESOURCE WSERV_INFO hello_accs
    {
    menbar_id=hello_menbar;
    first_com=O_COM_EXIT;
    accel={'x',    /* Exit */
           'h'};   /* Hello/Bye */
    }

RESOURCE MENU message_menu
    {
    items =
        {
        MENU_ITEM
            {
            com_id=O_HCM_HELLO;
            mn_item="Say goodbye";
            }
        };
    }

RESOURCE STRING say_hello {str="Say hello";}
```

All other resources are unchanged.

In all cases where menu text is replaced, the text that appears in the `MENU_ITEM` resource *must be the longest text of all the possible alternatives*. Thus, in this case, it is important that the text in the option corresponding to the message number `O_HCM_HELLO` is "Say goodbye" and that the replacement resource is the shorter "Say hello". The reason is that the menu resource is loaded into memory as a sequence of items of fixed length, determined by the size of the items in the resource. It is therefore possible to overwrite an element with a shorter replacement, but an attempt to replace it with a longer element will overwrite part of the following item.

A suitable `com_menu` method function is as follows:

```
#include <varray.g>

METHOD VOID hellocm_com_menu(PR_HELLOCM *self, INT menu_num, PR_VAROOT *array)
    {
    UBYTE *p;

    if (menu_num==0)
        {
        if (p_send2(w_ws->wserv.cli,O_WN_SENSE))
            {
            p=(UBYTE *)p_send3(array,O_VA_PREC,0);
            hLoadResBuf(SAY_HELLO,p+2); /* pointer skips the byte count and
method number */
            }
        }
    }
```

The method is passed the menu number (counting from zero for the leftmost menu in the menu bar) of the pull-down menu that is about to appear and the handle of an OLIB variable array containing the data for each of the options in that menu. Each element of this array consists of a leading length byte that indicates the (fixed) length of a following `MENU_ITEM` struct, defined in *pulldown.g* as:

```
typedef struct
    {
    UBYTE com_id;     /* command manager method number */
    TEXT mn_txt[1];   /* option text, as a zero terminated string */
    } MENU_ITEM;
```

The `com_menu` method tests if the relevant menu (in this case, the first menu, with menu number zero) is about to appear. If so, it further tests whether the text needs to be changed, indicated by the client window's `isbye` property being set (that is, the window is displaying "Goodbye" and so the menu option should be 'Say hello'). The additional client window `wn_sense` method is described below.

Provided both conditions are met, the address of the appropriate array element (in this case the first - and only - element, whose index is zero) is found by sending the array object a `VA_PREC` message. Finally, the replacement zero terminated text string is loaded from the resource file to overwrite the original text that starts at a two byte offset within the array element.

An additional client window method, `wn_sense`, is needed so that the command manager can determine the client window's state. The client window class definition must become:

```
CLASS   hellobw   bwin
bordered client window
    {
    REPLACE wn_init
    REPLACE wn_draw
    REPLACE wn_set
    REPLACE wn_sense
    PROPERTY
        {
        UWORD isbye;
        }
    }
```

and a suitable method function could be:

```
METHOD UINT hellobw_wn_sense(PR_HELLOBW *self)
    {
    return(self->hellobw.isbye);
    }
```

The `com_menu` method may be used to replace the text of any number of menu options in any number of menus. Any one call to the method will, of course, only replace the text of items in the one pull-down menu indicated by the `menu_num` parameter.

Disabling a menu option

Many applications may, from time to time, enter a state in which one or more command options do not represent valid operations. A common example is for a 'Copy' option that copies a highlighted section of data to a clipboard. This option is clearly inappropriate if none of the data is highlighted.

The recommended way of handling such a situation is to display an information message that explains why the option is not valid. In the case of copying, for example, the built-in applications display a "Nothing to copy" information message.

One way of implementing this is to add a validity check at the start of each of the relevant command manager methods. This is adequate if only one command is affected, but if it applies to a number of options this can lead to much duplicated code. A more efficient solution may be to subclass the `com_accl_check` method.

Whenever a menu option is selected, either from a pull-down menu or by an accelerator keypress, the command manager first receives a `COM_ACCL_CHECK` message, passing the method number of the message that corresponds to the selected command option. Only if the `com_accl_check` method function returns a `TRUE` value is the command manager sent (via `hWservComSend`) the message that executes the selected command option.

Suppose an application has a `com_copy` method in its `MYCOM` command manager, used to copy a highlighted region. A suitable `com_accl_check` method function would be of the form:

```
METHOD INT mycom_com_accl_check(PR_MYCOM *self, INT comid)
    {
    if (comid==O_COM_COPY)
        (
        if (!RangeHighlighted())
            {
            hInfoPrint(-NOTHING_TO_COPY);
            return(FALSE);
            }
        }
    return(TRUE);
    }
```

where the `RangeHighlighted` function is assumed to return `TRUE` only if a range is highlighted (possibly detecting this case by sending some form of SENSE message to the appropriate object). Note that `hInfoPrint` is passed a negative resource id, indicating that the resource with id `NOTHING_TO_COPY` is located in the system resource file (see the *Resource Files* chapter).

In the case of Copy it may be more efficient to perform the test within the `com_copy` method function, as follows:

```
METHOD VOID mycom_com_copy(PR_MYCOM *self)
    {
    if (!CopyRangeTo Clip())
        hInfoPrint(-NOTHING_TO_COPY);
    }
```

where `CopyRangeToClip` is assumed to return `TRUE` only if a highlighted range has been copied to a clipboard.

Which of these two methods to use will depend on the exact circumstances in a particular application.

Changing the number of options in a menu

In some circumstances one or more commands options may be disabled for the greater part of the time. An example of this is the Spell check option of the built-in word processor. This option is not available unless the Spell-checker application has been purchased and installed on the machine. It would be inappropriate to handle such an option in the way described in the previous section. A better way is to display the option in a command menu only if the option may validly be selected. This involves subclassing both the `com_menu` and `com_accl_check` methods.

The content of a pull-down menu is loaded from the application's resource file into an allocated memory cell whose size is determined by the size of the resource. It is therefore not an easy task to add items dynamically, once the resource has been loaded. It is, however, quite simple to remove items.

The recommended technique for varying the number of options in a menu is thus to include an entry for each such option in the appropriate `MENU` resource, and provide it with an accelerator as normal. The item may then be removed, if necessary, from the pull-down menu by replacing the `com_menu` method. Selection of the option by its accelerator (which does not involve the pull-down menu) must also be disabled, if necessary, by a suitable replacement `com_accl_check` method.

Suppose that an application's 'Optional' command option, with a Psion-O accelerator, is the third item in the fifth menu of an application and is executed by a `com_optional` method in a `MYCOM` subclass of the command manager. The `WSERV_INFO` resource and the appropriate `MENU` resource would contain the following:

```
RESOURCE WSERV_INFO my_accs
    {
    ...
    accel={...
           'o',    /* Optional */
           ...};

    }

RESOURCE MENU fifth_menu
    {
    items =
        {
        ...
        MENU_ITEM
            {
            com_id=O_COM_OPTIONAL;
            mn_item="Optional";
            },
        ...
        };
    }
```

The `com_menu` and `com_accl_check` methods would need to be of the form:

```
METHOD VOID mycom_com_menu(PR_MYCOM *self, INT menu_num, PR_VAROOT *array)
    {
     if (menu_num==4) /* in fifth menu */
        {
        if (!OptionalCommandIsValid())
             p_send3(array,O_VA_DELETE,2); /* delete third item */
        }
    }

METHOD INT mycom_com_accl_check(PR_MYCOM *self, INT comid)
    {
    if (comid==O_COM_OPTIONAL)
        (
        if (!OptionalCommandIsValid())
            return(FALSE);
        }
    return(TRUE);
    }
```

Displaying a status window

The command manager is sent a `COM_STATWIN` message when the application receives a Control-Menu keypress. The supplied `com_statwin` method does nothing, so the default action of an application is to ignore this keypress. An application that wishes to respond to the Control-Menu keypress by altering the state of its status window should subclass this method.

A Series 3 application may record (generally in an element of property) the current state of vis ibility of its status window and toggle its visibility by appropriate calls to either `wsEnable` or `wsDisable`. Before

changing the status window it should adjust the size of its client window display to fill the space that will not be occupied by the status window.

A Series 3a application has the option of switching between a large, small or no status window. The following code illustrates how the `com_statwin` method may be used to cycle around the three possible states. Note that this code assumes that the client window has a `wn_change_width` method that adjusts the width of the window, given the (signed) amount by which the width needs to be changed. Possible code for such a method is given in the *Windows* chapter of this manual.

```
METHOD VOID wpcman_com_statwin(PR_WPCMAN *self)
    {
    INT winType,delta;
    P_EXTENT oldExtent;
    P_EXTENT newExtent;

    winType=wInquireStatusWindow(-1,&oldExtent);
    if (winType==W_STATUS_WINDOW_OFF)
        winType=W_STATUS_WINDOW_BIG;
    else if (winType==W_STATUS_WINDOW_SMALL)
        winType=W_STATUS_WINDOW_OFF;
    else
        winType=W_STATUS_WINDOW_SMALL;
    wInquireStatusWindow(winType,&newExtent);
    delta=(oldExtent.width-newExtent.width);
    p_send3(w_ws->wserv.cli,O_WN_CHANGE_WIDTH,delta);
    wStatusWindow(winType);
    }
```

Application-specific initialisation

The command manager's `com_init` method is intended to be replaced in applications that need some form of application-specific command manager initialisation. Many applications will not need to replace this method. Application-specific initialisation may be necessary, for example, to create and initialise component objects used in the execution of one or more command options.

Note that the command manager is created and initialised (by being sent a `COM_INIT` message) immediately before the window server object is sent a `WS_DYN_INIT` message. This is important because it means that at the time the `com_init` method function is executed, the client window does not yet exist (since an application creates and initialises its client window in the `ws_dyn_init` method). The `com_init` method may not therefore make any direct reference to the client window or any components that it may create.

Replacing a menu bar

An application may, in some circumstances, wish to make a permanent or temporary replacement of its menu bar. An example is an application that can switch between, say, a graphic and a text display and requires a different set of command menu options for each view. Alternatively, an application may operate under a number of aliases (see the *Aliasing applications* section of the *Communicating with the System Screen* chapter of the *Series 3 Programming Guide*) and require a different set of command menu options for each alias.

The initial menu bar of an application is set up by system code, during the initialisation of the window server object, immediately before the creation and initialisation of the command manager. The data for the initial menu bar is read from the `WSERV_INFO` resource that must be the first item of the application's resource file. This struct specifies the resource id of the `MENU_BAR` resource that contains the menu bar text, the accelerators for each option and the command manager method number of the method to be associated with the first of these accelerators. The menu bar resource, in turn, contains the resource ids of the `MENU` resources that contain the pull-down menus associated with the menu bar items.

For each alternative menu, the application resource file must contain a separate `WSERV_INFO` resource, its associated `MENU_BAR` resource and any additional `MENU` resources (two or more menu bars may share a single `MENU` resource that is common to them). These additional resources may appear at any position, and in any order, in the application's resource file.

To change the menu bar, an application should send the window server active object a WS_SET_MENUBAR message, passing the resource id of the new WSERV_INFO resource. If, for example, an application has an alternate menu defined in its resource file as:

```
RESOURCE WSERV_INFO alternate_accs
    {
    ....
    }
```

it would change its menu bar by means of code of the form:

```
VOID *pOldMenBar;

pOldMenBar = p_send3(w_ws,O_WS_SET_MENBAR,ALTERNATE_ACCS);
```

The method returns a pointer to an allocated memory cell that contains the data for the original menu bar. If this is not to be restored the application should free this memory by calling, for example:

```
p_free(pOldMenBar);
```

If, however, the application is intending to restore the original menu at some future time it should preserve this pointer. The original menu bar is restored by the message:

```
p_send3(w_ws,O_WS_RESET_MENUBAR,pOldMenBar);
```

Neither of these two messages cause the new or the restored menu bar to be displayed. The appropriate menu will be made visible as normal when the Menu key is next pressed.

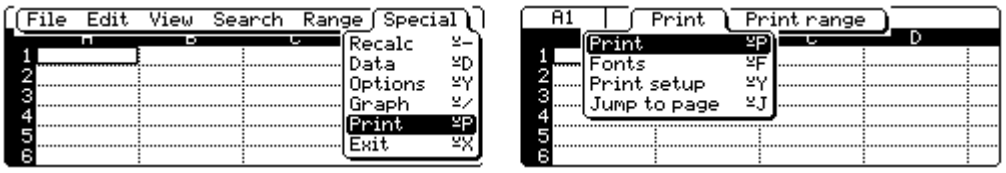
A permanent replacement of a menu bar, say for an aliased application, may be made at any time during application-specific initialisation. Suitable locations for the code ar either the command manager's com_init method or the window server object's ws_dyn_init method.

Note that any alias information text that was passed to the application in its start-up command line is pointed to by a text pointer in the HWIMMAN application manager's property, accessed through the w_am magic static by w_am->hwimman.aliasinfo, which must be declared as:

```
GLREF_D PR_HWIMMAN *w_am
```

Submenus

An application may make a temporary replacement of its menu bar to implement a submenu by sending the window server object a WS_DO_SUBMENU message. An example of this, taken from the Series 3 Spreadsheet application, is illustrated below.



The method will normally be called from the command manager method corresponding to a main menu option. As for the ws_set_menbar method, ws_do_submenu requires the resource id of a WSERV_INFO resource. Note that, as illustrated above, commands in a submenu may have names and/or accelerators that duplicate those appearing in the main menu. One use of submenus is therefore to provide a means of exceeding an application's normal limit on the number of its commands.

A typical call would be of the form:

```
p_send3(w_ws,O_WS_DO_SUBMENU,SUBMENU_ACCS);
```

and causes the submenu to become visible, ready for selecting one of its commands. Note that, unlike ws_set_menubar, this method does not return a pointer. The current main menu is restored automatically when the submenu menu bar ceases to be visible.

Shutdown messages

The System Screen may send an application a *Shutdown* message at any time, unless the application has explicitly elected not to receive such messages. It may do this by adding 4000 to its type number in its shell

data file (see *Application type numbers* in the *Communicating With the System Screen* chapter of the *Series 3 Programming Guide*).

An HWIM application receives a *Shutdown* request from the System Screen in the form of a normal `COM_EXIT` message to its command manager. It should handle this in exactly the same way as when the user explicitly selects the application's Exit option (the application actually has no means of distinguishing between the two cases). The default `com_exit` method supplied by the `COMMAN` class is quite adequate for an application that is not file-based. See the *File-based Applications* chapter for the required behaviour of `com_exit` when the application maintains an open file.

An application may indicate that it is temporarily unable to accept a *Shutdown* message by setting the `DatLocked` reserved static to be non-zero. This will generally only be relevant for a file-based application: a typical case is while an application is performing an extended operation that must run to completion, such as saving a file. The application must ensure that it clears `DatLocked` again, as soon as possible.

CHAPTER 6

WINDOWS

An HWIM window object represents a rectangular region on the screen, in which data may be displayed. The concept of a window is discussed in the *Introduction* and *Windows* chapters of the *Window Server Reference* manual. The following description of the basic HWIM window classes assumes a reasonable familiarity with the content of these chapters. In addition, drawing to a window requires a knowledge of the *Graphics Output* chapter of the *Window Server Reference* manual.

In general, an HWIM application uses a number of different windows. Some windows may exist for the entire lifetime of the application, for example, an application's main display window. Others may exist for a short time, such as the windows used to display a menu bar, a pull-down menu or a dialog box.

An HWIM application is expected to have one particular window, designated the *client window*, that (generally) exists for the lifetime of the application and provides the main view of the application's data. This window should be created as part of the application's initialisation, in the `WSERV` window server active object's `ws_dyn_init` method, with its handle being written to the `wserv.cli` property element. See, for example, the creation of a bordered client window in the "Hello World" example application.

All HWIM windows subclass the `WIN` class, which is thus the ultimate superclass of all windows. The window classes supplied by HWIM are abstract classes and will normally need to be subclassed in order to create useful window objects.

In contrast with the mechanisms supplied for the handling of command menus and dialogs, the HWIM library provides relatively few 'solutions' for the display of data in a window (but see the *Edit Windows* chapter). For example, an application can normally simply use the supplied dialog box class or, at most, replace one or two methods. It will, however, usually have to subclass the window classes to provide a variety of application-specific mechanisms. This chapter therefore describes the supplied window classes in more detail than is supplied for, say, the dialog box class.

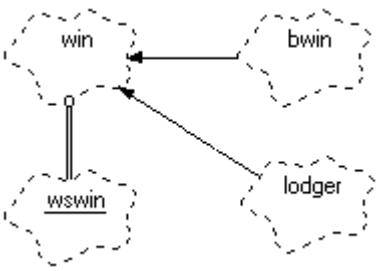
The supplied classes, listed below, are described in the following sections of this chapter. The descriptions of the methods of each of these classes are followed by explanations of a few key mechanisms.

`WIN` the ultimate window superclass.

`BWIN` a subclass of `WIN` that draws a standard border around the window.

`LODGER` a pseudo-window that subclasses `WIN`. Such a window is assumed to occupy a rectangular region within another window. The enclosing window (referred to as the *landlord*) will normally delegate all processing for that rectangle to the lodger.

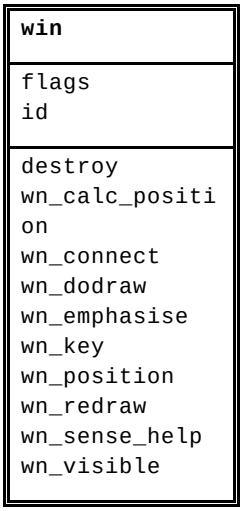
The relationships between these classes is illustrated in the following class diagram.



In addition to the methods and property in the application's code and data segments, a window has an associated data structure in the window server's resources, created by a call to `wCreateWindow` when an instance of `WIN` is itself created. The window server provides a set of functions that operate on such data structures, each data structure being uniquely identified by a window id returned by `wCreateWindow`.

Since a uniquely identifiable data structure with a set of functions that operate on it is effectively an object, the window server resources associated with a window can be considered as a component object. The existence of the window server resources is thus indicated in the above class diagram by the 'using' relationship between `WIN` and a notional `WSWIN` (Window Server WINdow) class.

The WIN class



The `WIN` abstract class subclasses `ROOT` to form the superclass for all windows in HWIM applications.

Class definition

Defined in sub-category file *hwim.cl* (generated header file *hwim.g*).

```
CLASS   win     root
The window object superclass
    {
    REPLACE destroy         Close server window then supersend destroy
    ADD wn_redraw           Called in reponse to WM_REDRAW
    ADD wn_dodraw           Same effect as redraw but called by application
    ADD wn_connect          Connect the window to window server data (wswin)
    ADD wn_position         Calculate position and re-position
    ADD wn_calc_position    Calculate position
    ADD wn_key=p_false      For processing WM_KEY from server
    ADD wn_visible          Alters visibility state of window
    ADD wn_emphasise        Window is highlighted in some way
    ADD wn_sense_help       Give start id for help
    DEFER wn_set            Set some window object property fields
    DEFER wn_sense          Sense some window object property fields
    DEFER wn_draw           Draw to existing graphics context - usually
subclassed
    DEFER wn_init           Specific initialisation for a type of window
```

```
CONSTANTS
    {
    CONTROL_HEIGHT          10      pixel height of a standard control
    PR_BWIN_CUSHION         0x1     matches W_BORD_CUSHION
    PR_BWIN_CORNER_4        0x2     matches W_BORD_CORNER_4
    PR_BWIN_SHADOW_1        0x4     matches W_BORD_SHADOW_S
    PR_BWIN_SHADOW_2        0x8     matches W_BORD_SHADOW_D
    PR_WIN_EMPHASISED       0x10    matches W_BORD_SHADOW_ON
    PR_BWIN_OPEN            0x20    matches W_BORD_OPEN
    PR_BWIN_CORNER_1        0x40    matches W_BORD_CORNER_1
    ! Gap of four bits, reserved for the arrow bits
    PR_WIN_INITIALISED      0x800    Window completely initialised
    PR_WIN_FORCE_RIGHT      0x1000
    PR_WIN_FORCE_LEFT       0x2000
    PR_WIN_FORCE_BOTTOM     0x4000
    PR_WIN_FORCE_TOP        0x8000
    PR_WIN_FORCE_FLAGS      0xf000
    IN_WIN_EMPHASISED       (PR_WIN_EMPHASISED)

    WV_INVISIBLE            0
    WV_VISIBLE              1
    WV_INITINVIS            2
    WV_INITVIS              3
    WN_KEY_NO_CHANGE        0
    WN_KEY_CHANGED          3
    WN_KEY_CHANGED_DEFER    7
    WN_KEY_CANCELLED        (-1)
    WN_KEY_ABSORB_ON        (-2)
    ERROR_RID_OFFSET        512
    }

PROPERTY
    {
    UWORD id;           window server window id
    UWORD flags;        PR_WIN EMPHASISED etc.
    }
}
```

Property

`win.id`	the window id, returned from a call to `wCreateWindow`, and used by the window server to identify the window
`win.flags`	a collection of flags recording the state of the window, as described below

Window flags

The content of `win.flags` may be any legal combination of the following flags:

`PR_WIN_EMPHASISED`	`TRUE` if the window is to be drawn in a highlighted state, the flag is set or cleared by the `wn_emphasise` method
`PR_WIN_INITIALISED`	`TRUE` if some significant aspect of the window initialisation is successfully completed, normally implying some change in action during subsequent destruction of the window. This is currently used by only the `DLGBOX` subclass
`PR_WIN_FORCE_RIGHT`	the window is to be positioned at the right hand edge of the screen. This flag is mutually exclusive with `PR_WIN_FORCE_LEFT`
`PR_WIN_FORCE_LEFT`	the window is to be positioned at the left hand edge of the screen. This flag is mutually exclusive with `PR_WIN_FORCE_RIGHT`
`PR_WIN_FORCE_BOTTOM`	the window is to be positioned at the bottom edge of the screen. This flag is mutually exclusive with `PR_WIN_FORCE_TOP`
`PR_WIN_FORCE_TOP`	the window is to be positioned at the top edge of the screen. This flag is mutually exclusive with `PR_WIN_FORCE_BOTTOM`

Additional flag bits are used by the `BWIN` subclass.

WIN methods

WN_CONNECT Create the window's window server data

```
VOID wn_connect(PR_WIN *par, UINT fields, W_WINDATA *windata);
```

Create the window server data structure for the window by calling the window server function `wCreateWindow`, writing the returned id to `win.id`, as indicated by the following code:

```
METHOD VOID win_wn_connect(PR_WIN *self,PR_WIN *par,UINT fields,W_WINDATA
*wdata)
    {
    self->win.id=wCreateWindow((par?par->win.id:0),fields,wdata,(UWORD)self);
    }
```

The parameters `windata` and `fields` are as described for `wCreateWindow` in the *Windows* chapter of the *Window Server Reference* manual. The value of `par` should be `NULL` if the window is to be a top-level window, otherwise it should be the handle of the parent window in the window tree. The value of `self` is passed to `wCreateWindow` for use by the window server as a handle to identify the window, for example, when the window must be sent a redraw message. The window server uses `self` in much the same way that window method functions use `win.id`.

In general, HWIM windows are created with the `W_WIN_BACK_BITMAP` bit in `windata->flags` cleared, so that they will be redrawn by the window server `WM_REDRAW` mechanism, which causes the window to be sent a `WN_REDRAW` message.

Assuming that the window server function `wDisableLeaves` has not been called, any failure will result in `p_leave` being called.

This method must be executed during the initialisation of any window.

Application code will frequently replace this method to customise the window's features and/or to create child windows.

DESTROY Destroy

```
VOID destroy(VOID);
```

Destroy the window, including its window server data structure if this has previously been successfully created. An application will not normally send this message to its client window (or to any permanent component of the client window).

The method ensures that no window server event can be directed to the window after its destruction sequence is initiated, and calls `wCloseWindowTree`, passing `win.id`, to free the window's window server data.

Finally, the method supersends the `DESTROY` message, to destroy the window's client-side resources (including automated destruction of any component objects).

WN_REDRAW System-initiated redraw

```
VOID wn_redraw(P_RECT *prect);
```

This method is intended to be called by system code to redraw the window, following receipt by `WSERV` of a window server event of type `WM_REDRAW`. An application will not normally replace this method, nor will it explicitly send a `WN_REDRAW` message to any of its windows.

The operation of the method is as illustrated in the following code:

```
VOID win_wn_redraw(PR_WIN *self, P_RECT *prect)
    {
    wBeginRedrawGC0(self->win.id);
    p_send2(self,O_WN_DRAW);
    wEndRedraw();
    }
```

Note that the whole window is redrawn, the rectangle coordinates pointed to by `prect` being ignored. A window that selectively redraws only the area specified by `prect` should replace this method.

WN_DODRAW Application-initiated redraw

```
VOID wn_dodraw(VOID);
```

This method is intended to be called by the application itself, for example, following a change in the data being displayed. An application will not normally subclass this method.

The operation is illustrated in the following code:

```
VOID win_wn_dodraw(PR_WIN *self)
    {
    wValidateWin(self->win.id);
    gCreateTempGC0(self->win.id);
    p_send2(self,O_WN_DRAW);
    gFreeTempGC();
    wFlush();
    }
```

The use of this method is equivalent to making a call to the window server function `wInvalidateWin`, except that the redrawing is performed immediately, without having to wait for a redraw event to arrive from the window server.

WN_VISIBLE Set visibility

```
INT wn_visible(UINT flag);
```

Set the visibility of the window, and any descendant windows, according to the value of `flag`. An application will not normally subclass this method.

During initialisation of a window, possible values of flag are:

`WV_INITVIS`	which calls `wInitialiseWindowTree(win.id)`
`WN_INITINVIS`	which calls `wMakeInvisible(win.id)` and then `wInitialiseWindowTree(win.id)`.

For an existing initialised window, this method may be called with flag set to either `WV_VISIBLE` or `WV_INVISIBLE` which respectively call `wMakeVisible(win.id)` or `wMakeInvisible(win.id)`.

This method will normally be called during the initialisation of a window. See also the HWIM utility function `hInitVis`.

WN_EMPHASISE Set window highlight

```
VOID wn_emphasise(UINT flag);
```

Set or clear the `PR_WIN_EMPHASISED` bit in `win.flags`, depending on whether flag is `TRUE` or `FALSE`. Then call `wInvalidateWin(win.id)` so that the window will eventually receive a `WN_REDRAW` message.

Windows generally draw themselves differently in some way, according to whether or not they are emphasised.

This method is not suitable for use in quality applications. It is expected that a subclass will always replace this method to provide specific and more effective redraw logic (see, for example the `BWIN` subclass).

WN_SENSE_HELP Sense start id for help

```
INT wn_sense_help(VOID);
```

Sense the resource id for the application's current Help index.

Returns the value of `w_ws->wserv.help_index_id` or, if this value is zero, the (negative) system resource id `-SYS_HELP_ON_HELP`.

This method can be subclassed to facilitate context-sensitive Help.

WN_CALC_POSITION Calculate a window position

```
VOID wn_calc_position(INT flags, P_EXTENT *pext);
```

Calculate, and write to `pext->tl.x` and `pext->tl.y`, the required screen coordinates of the top left corner of the window of width `pext->width` and height `pext->height` to place it on the screen in the position specified by `flags`. The calculation assumes that the window includes a blank 'cushion', one pixel wide, on all four sides.

The value of flags may contain any `OR`ed combination of:

- either `PR_WIN_FORCE_RIGHT` or `PR_WIN_FORCE_LEFT`, with obvious meanings. In either case the window is positioned so that the single pixel cushion at the side of the window that touches the edge of the screen is not visible. If neither flag is present the window is centred horizontally
- either `PR_WIN_FORCE_TOP` or `PR_WIN_FORCE_BOTTOM`, again with obvious meanings. In either case the window is positioned so that the single pixel cushion at the side of the window that touches the edge of the screen is not visible. If neither flag is present the window is centred vertically

An application will not normally need to either replace or make explicit calls to this method.

WN_POSITION Set window position

```
VOID wn_position(INT flags);
```

Position the window according to the value of flags, which may contain any `OR`ed combination of:

- either `PR_WIN_FORCE_RIGHT` or `PR_WIN_FORCE_LEFT`, with obvious meanings. In either case the window is positioned so that the single pixel cushion at the side of the window that touches the edge of the screen is not visible. If neither flag is present the window is centred horizontally
- either `PR_WIN_FORCE_TOP` or `PR_WIN_FORCE_BOTTOM`, again with obvious meanings. In either case the window is positioned so that the single pixel cushion at the side of the window that touches the edge of the screen is not visible. If neither flag is present the window is centred vertically

The method sends a `WN_CALC_POSITION` message to determine the required position before moving the window by means of a call to `wSetWindow`.

If used, this method will normally be called during the initialisation of a window, but must not be called before the window has processed a `WN_CONNECT` message.

An application will not normally need to replace this method

WN_KEY Process a keypress

```
INT wn_key(INT keycode, INT modifiers);
```

Process a keypress, where keycode is the code of the key pressed and modifiers contains a set of flags indicatinf which modifier keys (SHIFT, CTRL etc.) were held down when the key was pressed. The possible values of keycode and modifiers are described in the *Events* chapter of the *Window Server Reference* manual.

Depending on the type of keypress and the current state of the application, the `WN_KEY` message may be sent to a window (normally the client window or a dialog) by system code.

The supplied method simply returns zero (`WN_KEY_NO_CHANGE`).

A subclass will, in general, replace this method. In many cases the `wn_key` method of a window may delegate the processing by sending `WN_KEY` messages to one or more other windows.

Deferred WIN methods

A window will generally replace one or more of the methods described below. Note that there is no requirement for any particular subclass to replace all these methods.

WN_INIT — Initialise

```
VOID wn_init(...);
```

Provide class-specific initialisation. For most windows this will include a call of the form:

```
p_send5(self,O_WN_CONNECT,parent,ws_flags,&ws_data);
```

The number and types of parameters to the `wn_init` method depend on the class. However, once specified for a particular class, further subclasses will normally follow the same parameter structure.

Note that a simple window may not need to supply this method, since all essential initialisation may be performed by the `wn_connect` and `wn_visible` methods.

WN_SET — Set property

```
VOID wn_set(...);
```

Set one or more property fields.

The number and types of parameters to the `wn_set` method depend on the class. However, once specified for a particular class, further subclasses will normally follow the same parameter structure.

WN_SENSE — Sense property

```
VOID wn_sense(...);
```

Sense one or more property fields.

The number and types of parameters to the `wn_sense` method depend on the class. However, once specified for a particular class, further subclasses will normally follow the same parameter structure.

WN_DRAW — Draw to existing GC

```
VOID wn_draw(VOID);
```

Supply class-specific drawing for the whole area of the window, called from the `wn_redraw` and `wn_dodraw` methods. Most application subclasses will need to supply this method.

The method assumes that an appropriate graphics context exists, so the caller is responsible for supplying a graphics context. Note, however that this method is called from the `wn_redraw` and `wn_dodraw` methods, both of which create a basic graphics context around the sending of a `WN_DRAW` message.

The BWIN bordered window class

win	bwin

flags id	
destroy wn_calc_positi on wn_connect wn_dodraw ~~wn_emphasise~~ wn_key wn_position wn_redraw wn_sense_help wn_visible	wn_draw wn_emphasise

The BWIN abstract class is the superclass for all bordered windows.

Class definition

Defined in sub-category file *bwin.cl* (generated header file *bwin.g*).

```
CLASS   bwin      win
All windows that have a border use this
    {
    REPLACE wn_draw             Redraw border
    REPLACE wn_emphasise        Update border
    CONSTANTS
        {
        IN_BWIN_CORNER_4        PR_BWIN_CORNER_4
        IN_BWIN_SHADOW_1        PR_BWIN_SHADOW_1
        IN_BWIN_SHADOW_2        PR_BWIN_SHADOW_2
        IN_BWIN_CUSHION         PR_BWIN_CUSHION
        IN_BWIN_OPEN            PR_BWIN_OPEN
        BWIN_CUSHION_X          1
        BWIN_CUSHION_Y          1
        BWIN_SHADOW_1_HEIGHT    1
        BWIN_SHADOW_1_WIDTH     1
        BWIN_SHADOW_2_HEIGHT    2
        BWIN_SHADOW_2_WIDTH     2
        }
    }
```

Property

None.

BWIN methods

WN_DRAW — Draw border

```
VOID wn_draw(VOID);
```

Draw the window's border by calling

```
gBorder(self->win.flags);
```

The significant flag bits are PR_BWIN_CUSHION to PR_BWIN_CORNER_1 inclusive, together with the following four bits (0x80 to 0x400 inclusive) used to indicate the corner arrows. These bits are equivalent to the window server flags W_BORD_CUSHION to W_BORD_CORNER_1 and W_BORD_TOP_ON to W_BORD_BOT_OFF, whose effects are explained in the description of gBorder and gBorderRect in the *Graphics Output* chapter of the *Window Server Reference* manual.

An application-specific subclass will normally replace this method to provide drawing of the window's content, including the line:

```
p_supersend2(self,O_WN_DRAW);
```

to draw the border.

WN_EMPHASISE — Update border

```
VOID wn_emphasise(UINT flag);
```

If `flag` is `FALSE`, clear the `PR_WIN_EMPHASISED` flag (equivalent to the window server `W_BORD_SHADOW_ON` flag) in `win.flags`, otherwise set it.

Then redraw the border as indicated in the following code:

```
gCreateTempGC0(self->win.id);
gBorder(self->win.flags);
gFreeTempGC;
```

The shadow of a shadowed bordered window is only visible when the window is emphasised.

The LODGER class

win	lodger
flags id	landlord offset width
~~destroy~~ wn_calc_position wn_connect wn_dodraw wn_emphasise wn_key wn_position wn_redraw wn_sense_help ~~wn_visible~~	destroy lg_draw lg_self_check lg_set_id_pos wn_init wn_visible

The `LODGER` class defines a window that does not have its own independent window server data structure, and therefore does not have an independent window server id. A *lodger window* is defined to be any window that subclasses `LODGER`.

A lodger window occupies a rectangular region within another window, referred to as the lodger window's *landlord*, and shares the window id of the landlord window. In other respects a lodger window has broadly similar behaviour to a normal window, supporting a similar set of drawing and key processing messages. A lodger window can be considered to take over the responsibility for drawing the content of a rectangular region of the landlord window.

A significant difference between the two types of window is that drawing in a lodger window is not clipped by the boundaries of the lodger window itself. Drawing will only be clipped by the landlord window rectangle; as a consequence a lodger window has a duty never to draw outside its bounding rectangle.

The most common single use for a lodger window is as a component control within a dialog box.

Since a lodger window does not have an independent window server data structure, it is more efficient in terms of memory usage. This can be significant in the case of a complex compound window (a dialog box, for example, can easily need ten or more component sub-windows). An additional advantage is that such a compound window will scroll more smoothly - the scroll mechanism operates on only the single 'real' window and does not require messages to be sent to any of the component lodger windows (except for those that have freshly exposed regions to draw).

Class definition

Defined in sub-category file *lodger.cl* (generated header file *lodger.g*).

```
CLASS lodger    win
Lodger window
    {
    REPLACE wn_init              Set up landlord
    REPLACE destroy=root_destroy
    REPLACE wn_visible
    ADD lg_set_id_pos            Set window id after dynamic initialisation
    ADD lg_draw                  Temp GC and call wn_draw
    ADD lg_self_check=p_true     Returns TRUE if contents are legal
    DEFER lg_sense_width         For dialogs to set their sizes
    DEFER lg_update              For fnselwn and fnedit

    CONSTANTS
        {
        LG_CHECK_OK             1     Lodger self checked o.k., no change
        LG_CHECK_FAILED         0     Lodger failed self check, no change
        LG_CHECK_FAILED_CHANGED (-1)  Lodger failed self check, changed
        LG_CHECK_OK_CHANGED     (-2)  Lodger self checked ok., changed
        }

    PROPERTY
        {
        P_POINT offset;       offset into landlord window
        UWORD width;          width available to control
        PR_WIN *landlord;     owner window
        }
    }
```

Property

`lodger.offset`	the pixel coordinates of the top left corner of the lodger window with respect to the top left corner of the landlord window
`lodger.width`	the width, in pixels, of the area available to the lodger window
`lodger.landlord`	the handle of the window's landlord window, assumed to be (a subclass of) `WIN`. The lodger window has access to the window server id of the 'real' window (the landlord) via `lodger.landlord->win.id`

A lodger window's property does not record the height of the lodger window. This is suitable when a lodger window is a component of a dialog box since, in this case, it always has a fixed height of `CONTROL_HEIGHT` pixels.

LODGER methods

WN_INIT — Initialise

```
VOID wn_init(UINT *par, PR_WIN *landlord);
```

Initialise the lodger window by copying the value of `landlord`, which should be the handle of the owning landlord window, into `lodger.landlord`.

The method ignores the value of `par`, which is specified for use by subclassers.

DESTROY — Destroy

```
VOID destroy(VOID);
```

Destroy only the client-side resources of the lodger window, together with automatic destruction of any components.

Calls the `ROOT` `destroy` method directly, to avoid calling the `destroy` method at the `WIN` level, which might free the landlord's window server data structure.

WN_VISIBLE — Set visibility

```
VOID wn_visible(UINT flag);
```

Make the lodger window invisible if `flag` is `FALSE`, otherwise make the lodger window visible. (Note that, in contrast to the `WIN` superclass, only two flag values are supported.)

If the lodger window is being made visible, the method copies the landlord window's window server id into `win.id` and sends an `LG_DRAW` message to draw the lodger window's contents.

If the lodger window is being made invisible, `win.id` is set to zero and the method creates a temporary graphics context, calls `gClrRect` to clear the area in the landlord window that is occupied by the lodger window and then frees the temporary graphics context.

A non-zero value of `win.id` thus indicates that the window is visible and this test is used, for example, by the `lg_draw` method. Setting `win.id` to be a copy of the value in `lodger.landlord->win.id` is an implementation decision that simplifies access to the window server id of the 'real' window.

LG_SET_ID_POS — Set id, position and width

```
VOID lg_set_id_pos(INT id, P_POINT *pos, UINT width);
```

Set `win.id` to the value of `id`, copy `*pos` into `lodger.offset` and set `lodger.width` to `width`.

A typical call to this method will be from a dialog box after all the items have been loaded and the dialog box has been set to the required width to display all its items.

LG_DRAW — Create GC and draw

```
VOID lg_draw(VOID);
```

If `win.id` is non-zero, indicating that the window is in the visible state, draw the window content:

```
gCreateTempGC0(self->win.id);
p_send2(self,O_WN_DRAW);
gFreeTempGC();
```

Otherwise do nothing.

LG_SELF_CHECK — Check content is valid

```
INT lg_self_check(INT item, INT can_defer);
```

Perform a check that any property contains valid data. A numeric editor may, for example, check that its current value is within its allowed upper and lower bounds. The supplied method just returns `LG_CHECK_OK`.

Deferred LODGER methods

These methods are intended to be replaced by classes that are used as components of a dialog box.

LG_SENSE_WIDTH — Sense required width

```
INT lg_sense_width(VOID);
```

This method is expected to return the width, in pixels, required to draw the lodger window contents.

An object will only receive this message when it is a component of a dialog box (see the `DLGBOX` `dl_set_size` method, described in the *Dialog Boxes* chapter).

Some system-supplied subclasses assume that this method is called only once in the lifetime of an instance.

LG_UPDATE Update file name

```
VOID lg_update(TEXT *pack, INT derr);
```

This method is defined for the FNSELWN and FNEDIT dialog box component classes. See the descriptions of these classes for an explanation of the method.

The draw/redraw mechanism

This section is written under the assumption that a window has been created without a back-up bitmap. An HWIM application does not normally create backed-up windows and will therefore have to handle redraw events. The main advantages that this confers on an application is that drawing to such a window is faster and much less memory is used for each window. See the *Windows* and *Redrawing* sections of the *Introduction* chapter of the *Window Server Reference* manual for the background to drawing and redrawing the content of a window.

All or part of the content of a window may have to be drawn for one of the following basic reasons:

- the application's displayed data has changed
- all or part of the window has been exposed, for example, by the disappearance of an overlying dialog, or by the application being brought to the foreground.

In both cases the drawing is ultimately handled by the window's wn_draw method, but the mechanism by which this is invoked differs in the two cases.

The first of these generally results from some operation within the application itself, such as a keypress or the execution of a command. Normally the application will be aware that such a change has taken place and can explicitly initiate the drawing. Most applications can do so by sending the window a WN_DODRAW message. This creates a temporary graphics context with default properties, sends a WN_DRAW message and then destroys the temporary graphics context. The graphics context may be modified, if necessary, in the window's wn_draw method.

The second case may occur at any time, without the application being aware of any particular need to do any redrawing. The window server process will, however, send the aplication a WM_REDRAW inter-process message, indicating that a particular area of a specific window needs to be redrawn. This is processed by the application's window server object which then sends a WN_REDRAW message to the specified window. As for the wn_dodraw method, a temporary graphics context is created around a WN_DRAW message, which may be processed exactly as in the previous case.

At the level of the wn_draw method, a window can not (and does not need to) distinguish between the two cases. In fact, as an alternative to sending the window a WN_DODRAW message when its data has changed, the application could simply invalidate the window (by calling wInvalidateWin). System code will ensure that the window eventually receives a WN_REDRAW message. In many cases this may be simpler from the point of view of coding, but is generally less efficient. It may lead to poor responsiveness, particularly in applications that make rapid changes to their data.

Resizing a window

The following suggestion for a wn_resize method illustrates a simple way to change the position and size of a window without changing any other attribute:

```
METHOD VOID mywin_wn_resize(PR_MYWIN *self, P_EXTENT *pext)
    {
    W_WINDATA wd;

    wd.extent=*pext;
    wSetWindow(self->win.id,W_WIN_EXTENT,&wd);
    }
```

S3/S3a client windows generally do not change size once they have been created and made visible. The main exception is a change in width corresponding to a change in the state of any permanent status window. Thus a common requirement is to change the width, without changing the position or height, as in the following example for a possible wn_change_width method:

```
METHOD VOID mywin_wn_change_width(PR_MYWIN *self, INT delta)
    {
    W_WINDATA wd;

    wInquireWin(self->win.id,&wd);
    wd.extent.width+=delta;
    wSetWindow(self->win.id,W_WIN_EXTENT,&wd);
    }
```

Window emphasis

Emphasis is used to indicate which of several windows is the one with which the user is currently interacting.

System code will send the application's client window a `WN_EMPHASISE` message to turn emphasis on or off; for example, when a dialog appears or disappears. Application code is not normally expected to send `WN_EMPHASISE` messages itself, except under the circumstances described below.

A window will usually respond to a `WN_EMPHASISE` message by changing its appearance in some way. The normal technique is to set or clear, as appropriate, the `PR_WIN_EMPHASISED` flag in the window's `win.flags` property and then trigger a redraw, either by calling `wInvalidateWin` or by sending itself a `WN_DODRAW` message. In either case the window's `wn_draw` method will subsequently be executed. This can then test the `PR_WIN_EMPHASISED` flag and draw the window as appropriate. Common means of showing that a window is not emphasised are:

- a window with a shadowed border is drawn without its shadow
- any highlighted region may be drawn without its highlight
- a window that has a text cursor does not display its cursor

The last of these three differs from the other two in that it must *not* be done in the window's `wn_draw` method. The reason is that there is only ever one cursor visible on the screen at any one time and the window server function `wEraseTextCursor` erases the text cursor *regardless of the window in which it appears*. An unemphasised window will receive `WN_DRAW` messages (for example, from the `wn_redraw` method) and so a call to `wEraseTextCursor` from within the `wn_draw` method could result in the removal of the text cursor from another currently emphasised window.

An application can avoid the possibility of 'stealing' another window's text cursor by confining all calls to `wTextCursor` and `wEraseTextCursor` to be from within a window's `wn_emphasise` method. When changing emphasis from one window to another, system code always sends a `WN_EMPHASISE,FALSE` message (which may call `wEraseTextCursor`) to the window losing emphasis before sending a `WN_EMPHASISE,TRUE` message (which will, if necessary, call `wTextCursor`) to the window gaining emphasis. The currently emphasised window will thus be guaranteed to display its text cursor, if it has one.

A window that contains one or more child windows may delegate all or part of the processing of a `WN_EMPHASISE` message to its child windows. An example of this is shown in the behaviour of a dialog box. On receipt of a `WN_EMPHASISE` message, a dialog box changes the appearance of its border and then sends a `WN_EMPHASISE` message to one of its items (the one with 'focus').

A dialog box also illustrates the case where application code may send `WN_EMPHASISE` messages other than in response to such a message sent by system code. When a user presses the up or down arrow keys, a dialog responds by changing the focus from one dialog item to another. As part of this process `WN_EMPHASISE` messages are sent to the two items concerned. A similar process occurs when switching between the Find window and the main display window in the Database application.

In such a situation it is important to obey the rule that a `WN_EMPHASISE,FALSE` message must be sent to the window losing emphasis before sending a `WN_EMPHASISE,TRUE` message to the window gaining emphasis.

CHAPTER 7

DIALOGS

A dialog box displays the current values of one or more data items and, in general, allows the user to modify one or more of these values.

This chapter describes the basic mechanisms provided in HWIM for the creation and operation of dialogs. Dialog boxes may be created and used at a number of levels; this chapter is intended to describe the more common uses that cover the great majority of cases.

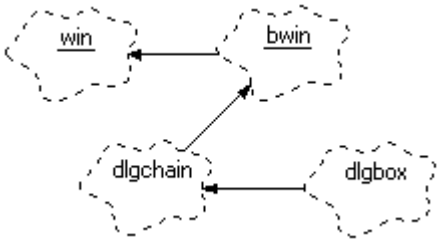
Some of the `DLGBOX` methods are complex, having to cope with a variety of cases. For most purposes it is not necessary to understand these methods in detail - the essential information for most uses will be found in this chapter. The *Dialog Controls* chapter contains the essential information about the standard dialog components that are supplied in the HWIM library.

In an HWIM application the most common use of a dialog is as a result of the user selecting a command from a command menu. In response to an *Open file* command, for example, a dialog would be presented to allow the user to specify the name (and possibly the type) of the file to be opened.

There are a number of system dialogs that may be run by specific `WSERV` methods, such as `ws_error_dialog`, `ws_query_dialog` and `ws_format_dialog`, described in the *Using the System Components* chapter. Application-specific dialogs are usually started by means of the window server object's `ws_do_dial` method, or the equivalent `hLaunchDial` utility function.

All HWIM dialogs are *modal*, that is, while the dialog is visible the user can interact with the application only via that dialog; the application enters a 'mode' such that all attempts to interact with, say the menu bar are disallowed. This mode terminates when the user satisfactorily completes the dialog.

Application-specific dialogs use an instance of (a subclass of) `DLGBOX` which is a subclass of the `BWIN` bordered window class, via the `DLGCHAIN` abstract class, as indicated in the following class diagram.



The following four sections describe the essential information about the `DLGCHAIN` and `DLGBOX` classes and methods. On first reading you may prefer to skim quickly through these sections and go on to the *Using dialog boxes* section.

The DLGCHAIN class

From the point of view of an application writer who wishes to make use of dialog boxes, this section is not essential reading. It is included largely for interest only, but may be of help in understanding the significance of the `DLGBOX_NO_DDP` flag, described in the following *The DLGBOX class* section.

The `DLGCHAIN` abstract class subclasses `BWIN`, but adds or modifies no methods. It adds a single item of property that makes all instances of its subclasses suitable for including in a chained list of objects. The

property, `dlgchain.next`, is used to contain the handle of the next item in a list of `DLGCHAIN` instances. The relevant header information is contained in the generated header file *dlgbox.g*.

A `DLGCHAIN` list is maintained by the application's instance of (a subclass of) `WSERV` to implement stacked dialog boxes. The foremost dialog's handle is stored in `wserv.dial`.

Other subclasses of `DLGCHAIN` may also be included in this list. For example, Help 'dialogs' (which do *not* subclass `DLGBOX`) may appear in the list, intermixed with true dialogs.

The `DLGCHAIN` class also defines the `PR_WIN_NO_DDP` flag, principally for use by the `DLGBOX` subclass. Any subclass of `DLGCHAIN` may, during , its initialisation, set this flag in its `win.flags` property. The normal state is for this flag to be clear, which signals that an instance whose handle appears in the `wserv.dial` list is prepared to have its handle written to the magic static `DatDialogPtr`. An advantage of doing so is that the handle does not need to be passed as a parameter to any utility functions. This feature is used, for example, by the dialog box utility functions described in the *HWIM Utility Functions* chapter.

A subclass of `DLGCHAIN` that does not set the `PR_WIN_NO_DDP` flag should write its handle to `DatDialogPtr` when it adds itself at the front of the `wserv.dial` list, during initialisation. Note that this handle may be overwritten by a different value when another subclass of `DLGCHAIN` adds itself to the list.

Any such subclass must cooperate with other items in the `wserv.dial` list. If, on destruction of an instance, `DatDialogPtr` contains its own handle, it must scan all the remaining items in the list and write to `DatDialogPtr` the handle of the first item for which `win.flags` does not contain `PR_WIN_NO_DDP`. An object's handle may therefore be written to `DatDialogPtr` during the destruction of another item in the list. As a consequence, this behaviour is mandatory, even if the instance does not actively write its own handle to `DatDialogPtr` on initialisation.

The DLGBOX class

The `DLGBOX` class provides a flexible set of mechanisms for displaying and controlling a wide variety of dialog boxes. Although formally an abstract class (since some methods are `DEFER`red) an instance of `DLGBOX` may be created and used to display simple notification dialogs. The majority of normal dialogs may be created and used with a subclass that replaces at most two methods: very few dialogs require all the `DEFER`red methods to be defined. This is discussed at greater length in the later *Using dialog boxes* section.

This and the following two sections contain information about dialog boxes and the available methods that is largely needed for reference. On first reading you may wish to skim quickly through them and concentrate on the *Using dialog boxes* section, which places greater emphasis on the more practical aspects.

An HWIM dialog box consists of a bordered window containing between one or more lines, or *items*. Each item may be plain text, a control, or a combination of a plain text prompt and a control. Each control may be an instance of one of many different classes, including:

- a choice list, allowing selection of one of a list of options
- an action list, containing one or more buttons (this may only be used as the last item in a dialog)
- an integer (WORD or LONG) numeric editor
- a floating point numeric editor
- a time editor
- a date editor
- a scrolling or non-scrolling text editor
- a secret data input box
- a filename choice list
- a filename editor
- an application-specific control

A dialog written for the Series 3 (or for the Series 3a in compatibility mode) may contain up to seven items, which may be divided into two groups by a single underline. A Series 3a dialog can display up to nine items,

any number of which may be underlined. Note that the inclusion of an action list, which occupies two lines, reduces the number of items that may be displayed. On the Series 3a the inclusion of more than one underline may also reduce the possible number of items.

In the majority of dialogs a single underline is used to separate the first item, designated to be the dialog title, from those that follow. The title is normally, but not necessarily, plain text.

Dialog boxes use the following structs, defined in the generated header file *dlgbox.g*:

```
typedef struct
    {
    UWORD flags;
    TEXT title[1];  /* ZTS string */
    /* followed by a byte count of the following items, */
    /* each of which has a leading length byte */
    } HD_DLGBOX_RSC;

typedef struct
    {
    WORD flags;         /* initialisable DB_ITEM flags */
    UBYTE class;        /* class of item */
    TEXT prompt[1];     /* ZTS string */
    /* followed by an IN_XXX component-specific initialisation structure */
    } AD_DLGBOX;        /* header for each item in resource information */

typedef struct
    {
    PR_LODGER *hand;      /* handle of item's control */
    PR_TEXTWIN *prompt;   /* handle of prompt for item */
    WORD flags;           /* collection of DLGBOX_ITEM_XXX flags */
    } DLGBOX_ITEM;        /* internal representation of items */
```

The first two of these structs correspond to the `DIALOG` and `CONTROL` resource structs respectively. They represent the in-memory copies of resources of the corresponding types. The resource structs themselves are described in the *Using dialog boxes* section, later in this chapter. The third struct is used for the internal representation of each of the dialog box items, in the `dlgbox.item` array that is described below.

Property

The `DLGBOX` class contains the following items of property that may be useful to developers.

`dlgbox.item`	used internally to provide access to an array of `DLGBOX_ITEM` structs for the component items. *Subclassers must not access the data in this array other than through the supplied methods, such as* `dl_index_to_handle`.
`dlgbox.rbuf`	the address of a user-supplied 'result' buffer, from which initialisation data may be read and to which result data may be written. The buffer's address may optionally be passed to the window server object's `ws_do_dial` method or the equivalent `hLaunchDial` utility function.
`dlgbox.dimrid`	either zero or the resource id of a text message that is to be displayed if a user attempts to modify a 'dimmed' control.
`dlgbox.helprid`	either zero or the resource id of a context-specific Help resource.
`dlgbox.flags`	a collection of state flags, described below.
`dlgbox.focus`	`TRUE` if keyboard focus is held by one of the dialog box controls. *Only system code may write to this item.*
`dlgbox.count`	a count of the number of items in the dialog box, including any dialog title. *Only system code may write to this item.*
`dlgbox.current`	the index of the dialog's component control that currently has focus. *Only system code may write to this item.*

Dialog box flags

There are three levels of flags that are associated with dialogs.

The first of these contains flags that are of significance to the dialog as a whole. They are normally specified by the `flags` field of a `DIALOG` resource and have names that start with `DLBOX_`.

The second level contains those flags that are of interest to the dialog box, but are associated with each separate control. These flags are normally specified by the flags field of a CONTROL resource, and have names that start with DLGBOX_ITEM_.

The third level is flags that are private to each component control. These flags are described for each type of control in the following chapter.

DLGBOX_ flags

The content of dlgbox.flags may be any sensible combination of the following flags:

DLGBOX_NOTIFY_ENTER	if this flag is set the dialog box will be sent a DL_KEY message when the dialog receives an Enter keypress.
DLGBOX_NOTIFY_ESCAPE	if this flag is set the dialog box will be sent a DL_KEY message when the dialog receives an Esc keypress.
DLGBOX_RBUF_FILLED	if this is not set and dlgbox.rbuf is not NULL, system code will write to *dlgbox.rbuf on exiting the dialog. The value that is written indicates the key that was pressed to termunate the dialog. A subclass that uses dlgbox.rbuf for its own purposes should set this flag.
DLGBOX_ACTION_LIST	this flag must only be set if the dialog contains an action list (class ACLIST) as its last item. It does not need to be explicitly included in the DIALOG resource as it is set automatically during the initialisation of an ACLIST component control. When set (provided dlgbox.absorb is FALSE) all received keys are first offered to the action list by sending it a WN_KEY message, which returns a value indicating whether or not the keypress matched one of the buttons.
DLGBOX_SMALL_ACTION_LIST	this flag must only be set if the dialog contains a 'small' action list (class SMACLIST) as its last item. It does not need to be explicitly included in the DIALOG resource as it is set automatically during the initialisation of an SMACLIST component control. When set (provided dlgbox.absorb is FALSE) all received keys are first offered to the action list by sending it a WN_KEY message, which returns a value indicating whether or not the keypress matched one of the buttons.
DLGBOX_NOTIFY_ALL_ACT	if this flag is clear, report only those keys that match an action button by sending the dialog box a DL_KEY message, otherwise send a DL_KEY message for all keypresses that have been offered to the action list, irrespective of whether they matched an action button. There is no need to include this flag unless the dialog contains one of the two forms of action list.
DLGBOX_REPORT_ACT_HORIZ	if a keypress to an action button causes the dialog to terminate, a value may be written to *dlgbox.rbuf (see DLGBOX_RBUF_FILLED). If DLGBOX_REPORT_ACT_HORIZ is set, the value written to *dlgbox.rbuf is the index number of the button that matched the keypress (or -1 if it did not match). Otherwise the value is the uppercased key code of the keypress. There is no need to include this flag unless the dialog contains one of the two forms of action list.
DLGBOX_NO_WAIT	if this flag is clear, the call to the ws_do_dial method or to hLaunchDial to start the dialog will not return until the dialog is complete. See *Dialogs with and without WAIT*, later in this chapter.
DLGBOX_APPEND_UNITS_TITLE	set this flag to append the current preferred units (cm or inches) to any dialog title.
DLGBOX_NO_SHADOW	if this flag is set the dialog box is created without a shadow to its border.

DLGBOX_NO_DDP	if this flag is clear, the dialog box's handle is written to DatDialogPtr during initialisation. On destruction, either NULL or the handle of another item in the wserv.dial list is written back to DatDialogPtr. The handle, if any, that is written is that of the foremost remaining item for which this flag is clear. Normally this flag is left clear to improve the efficiency of dialog-related utility functions, which then do not need to pass the dialog box handle as a parameter.
PR_WIN_FORCE_RIGHT	if set, the dialog will be positioned at the extreme right of the screen (see the *Windows* chapter).
PR_WIN_FORCE_LEFT	if set, the dialog will be positioned at the extreme left of the screen (see the *Windows* chapter).
PR_WIN_FORCE_BOTTOM	if set, the dialog will be positioned at the bottom of the screen (see the *Windows* chapter).
PR_WIN_FORCE_TOP	if set, the dialog will be positioned at the top of the screen (see the *Windows* chapter).

DLGBOX_ITEM_ flags

These flags apply to a single item within a dialog box. They are read from the initialisation data for each dialog box control, which is normally loaded from a resource file.They are thus normally specified by a flags field in a CONTROL resource that is used as a dialog box component. The flags are stored for each element in the array indicated by dlgbox.item and may be any combination of the following flags:

DLGBOX_ITEM_NOTIFY_CHANGED	if this flag is set, changes to the item result in the dialog box receiving a DL_CHANGED message. This flag is normally only set in the item's initialisation data.
DLGBOX_ITEM_NOTIFY_FOCUS	if this flag is set, each loss or gain of focus by the item results in the dialog box being sent a DL_FOCUS message. This flag is normally only set in the item's initialisation data.
DLGBOX_ITEM_UNDERLINED	if this flag is set, the corresponding item will be drawn with an underline extending across the full width of the dialog box. This flag is normally only set in the item's initialisation data. For a Series 3 application it must not be set for more than one item in the dialog box.
DLGBOX_ITEM_APPL_CAT	this flag should be set for any dialog item whose class definition is not in the HWIM category file. This flag may only be set in the item's initialisation data.
DLGBOX_ITEM_CENTRE	this flag indicates that the corresponding item is to be centred in the dialog box. This flag is normally only set in the item's initialisation data.
DLGBOX_ITEM_NEEDS_PACK	this flag indicates that, on initialisation of the dialog box, a 'pack selector' control is to be added as the item immediately following this one. This flag may only be set in the item's initialisation data, and *must* be set for FNSELWIN or FNEDIT component controls.
DLGBOX_ITEM_LOCKED	an item for which this flag is set is visible and can take focus, but its value may not be changed. This flag may be set or cleared by the dl_item_lock method.
DLGBOX_ITEM_DIMMED	if this flag is set, the item is 'dimmed', that is, its control is made invisible. This flag may be set or cleared by the dl_item_dim method.
DLGBOX_ITEM_DEAD	an item for which this flag is set may never take focus and may never be modified. This flag may only be set in the item's initialisation data.

DLGBOX_ITEM_CAN_DEFER_X	an item for which this flag is set can, on losing focus, defer its self-check (see *Consistency checks*, below). A deferred check is automatically registered by the setting of the DLGBOX_ITEM_X_PENDING flag, described below. Applications will normally only set the DLGBOX_ITEM_CAN_DEFER_X flag in an item's initialisation data, but the flag is set automatically during initialisation of FNSELWN or FNEDIT component controls.
DLGBOX_ITEM_X_PENDING	an item for which this flag is set must be checked before exiting the dialog. This flag is set and cleared by the system and should not be set or modified by application code.

DLGBOX methods

This section does not describe all the DLGBOX methods.With the exception of the wn_key method, only those methods are included that can reasonably be either used or replaced.

Dialog box items are generally accessed by an index number. The items are numbered, with the first item being item 0, in the order in which they are added to the dialog box (which is also the order in which they are displayed).

In addition to optionally providing replacements for the DEFERred methods:

```
dl_changed
dl_focus
dl_launch_sub
dl_item_new
```

applications are, in general, not expected to replace methods other than:

```
dl_dyn_init
dl_key
dl_inq_minsize
dl_set_size
```

Occasionally a dialog box may additionally need to replace one or more of:

```
dl_item_add
dl_dimmed_message
wn_sense_help
```

Consistency checks

A dialog box control may need to perform a consistency check on its content, a typical case being a numeric control whose value must remain within prescribed limits.

In simple cases a dialog may check the consistency of its data in its dl_key method. However, in many cases it is either not feasible or inappropriate to check the value at each keypress that modifies the value. For example, a numeric edit box whose value is constrained to lie between 10 and 50 may transiently contain the 'illegal' text "1" while the number "18" is being typed in.

The DLGBOX class provides a mechanism for checking the validity of the content of its controls, triggered either when focus is transferred to one of its controls, or when the dialog is being terminated by any means other than by pressing Esc. Since a control's validity may, for example, depend on the values of one or more of the other controls, the check on change of focus may be deferred until the dialog termination, when the final values of all controls are known. Such items are indicated by setting the DLGBOX_ITEM_CAN_DEFER_X flag.

The check on an item is deemed to succeed if the item is locked or dimmed, or if the check is deferred, otherwise the control is sent an LG_SELF_CHECK message. This message returns a value that indicates whether or not the content has changed as well as whether the check failed or succeeded. Regardless of the success or failure of the test, if the return value indicates that the value has changed, and provided the relevant dlgbox.item array element contains DLGBOX_ITEM_NOTIFY_CHANGED, the dialog box is sent a DL_CHANGED message.

On any failure the checking stops and focus is set to the item that failed its check. This may prevent the user from moving to another item in the dialog or from exiting the dialog (except by pressing Esc) until the item is modified so that it passes the check.

DL_DYN_INIT Dynamic initialisation

```
VOID dl_dyn_init(VOID);
```

This method is supplied so that it may be replaced to perform application-specific initialisation of the dialog box after all its items have been added, but before its size is calculated and it is made visible.

The most common use for this method is to set the initial value of one or more controls, depending on the current state of the application, but other actions may include dimming, locking, adding or replacing one or more dialog items. Initialisation data may conveniently be passed by means of a buffer pointed to by `dlgbox.rbuf`.

The supplied method does nothing.

DL_KEY Handle key input

```
INT dl_key(INT index, INT keycode, INT actbut);
```

Process a keypress that may potentially exit the dialog.

This method is supplied so that it may be replaced to perform application-specific processing of such keypresses.

This message is sent by the dialog box `wn_key` method in the following circumstances:

- when `dlgbox.flags` contains `DLGBOX_NOTIFY_ENTER` and the dialog is about to terminate after receiving a `W_KEY_RETURN`. The value of `keycode` is `W_KEY_RETURN` and `actbut` is -1
- when `dlgbox.flags` contains `DLGBOX_NOTIFY_ESCAPE` and the dialog is about to terminate after receiving a `W_KEY_ESCAPE`. The value of `keycode` is `W_KEY_ESCAPE` and `actbut` is -1
- the dialog is about to terminate after receiving a key that matches a button in an action list. The value of `keycode` is the uppercased key code that was passed to the action list's `wn_key` method and `actbut` is the index (0 for the leftmost button) of the matching button
- following any non-matching keypress received by an action list, provided `dlgbox.flags` contains `DLGBOX_NOTIFY_ALL_ACT`. The value of `keycode` is the uppercased key code that was passed to the action list's `wn_key` method and `actbut` is -1

In all cases `index` contains the current value of `dlgbox.current`.

The method should return `WN_KEY_NO_CHANGE` to prevent termination of the dialog, or `WN_KEY_CHANGED` to confirm termination (which causes the dialog box to be sent a `DESTROY` message). Before returning `WN_KEY_CHANGED` the method is responsible for ensuring that any relevant dialog box state is either saved or returned to the initiator of the dialog box. Data may conveniently be returned to the initiator via a user-supplied buffer pointed to by `dlgbox.rbuf`.

The supplied method simply returns `WN_KEY_CHANGED`.

WN_SET Set item by index

```
VOID wn_set(INT index, VOID *par);
```

Set one or more data elements in the property of the control associated with the dialog box item with index number `index`, by sending a `WN_SET` message to the control.

The parameter `par` is assumed to be a pointer to a struct that specifies the data to be set. The type of struct that is expected depends on the class of the control that is being set; the various structs are described in the following *Dialog Controls* chapter.

This method will normally be used (rather than replaced) by application writers. See also the various `hDlgSetXXX` utility functions.

WN_SENSE — Sense item by index

```
VOID wn_sense(INT index, VOID *par);
```

Sense the property of the control associated with the dialog box item with index number index, by sending a WN_SENSE message to the control.

The parameter par is assumed to be a pointer to a struct that matches the data to be sensed. The type of struct that is expected depends on the class of the control that is being sensed; the various structs are described in the following *Dialog Controls* chapter.

This method will normally be used (rather than replaced) by application writers. See also the various hDlgSenseXXX utility functions.

DL_INDEX_TO_HANDLE — Sense item handle

```
PR_LODGER *dl_index_to_handle(INT index);
```

Return the handle of the control in item number index.

It is a programming error to call this method with a value of index that does not correspond to an existing dialog box item.

This method is not intended to be replaced.

DL_HANDLE_TO_INDEX — Sense item index

```
INT dl_handle_to_index(PR_LODGER *handle);
```

Return the index number for the item whose control handle is handle.

Returns -1 if handle does not match any dialog item.

This method is not intended to be replaced.

DL_ITEM_DIM — Dim an item

```
VOID dl_item_dim(INT index, UINT flag);
```

Undim item number index if flag is FALSE, otherwise dim it. When an item is dimmed its control is not displayed and its prompt does not display a bullet point. It is harmless to send a dimmed item a DL_ITEM_DIM,TRUE message, or an undimmed item a DL_ITEM_DIM,FALSE message.

Dimming the item clears the DLGBOX_ITEM_DIMMED flag in item number index and sends any prompt (an instance of TEXTWIN) a WN_SET message to clear its PR_TEXTWIN_BULLET flag. The control is sent a WN_VISIBLE,FALSE message.

Undimming the item reverses these flag changes and sends the control a WN_VISIBLE,TRUE message.

In either case the WN_VISIBLE message is sent only if the dialog box win.flags contains PR_WIN_INITIALISED. This prevents any control from drawing itself during the initialisation of the dialog box, before the whole dialog box can be made visible.

If item number index has the DLGBOX_ITEM_NEEDS_PACK flag set, the dimming/undimming operation is also performed on the following PACKSEL item.

This method is not intended to be replaced.

DL_ITEM_LOCK — Lock an item

```
VOID dl_item_lock(INT index, UINT flag);
```

Unlock item number index if flag is FALSE, otherwise lock it. When an item is locked its prompt does not display a bullet point but, unlike a dimmed item, its control remains visible. It is harmless to send a locked item a DL_ITEM_LOCK,TRUE message, or an unlocked item a DL_ITEM_LOCK,FALSE message.

Locking the item clears the DLGBOX_ITEM_LOCKED flag in the item's flags field and sends any prompt (an instance of TEXTWIN) a WN_SET message to clear its PR_TEXTWIN_BULLET flag.

Unlocking the item reverses these flag changes.

If item number index has the DLGBOX_ITEM_NEEDS_PACK flag set, the locking/unlocking operation is also performed on the following PACKSEL item.

This method is not intended to be replaced.

DL_SET_PROMPT Change the prompt for an item

```
VOID dl_set_prompt(INT index, SE_TEXTWIN *par);
```

Send a WN_SET message to the prompt associated with item number index, passing the parameter par, assumed to be a pointer to an SE_TEXTWIN struct (see the Text windows section of the following *Dialog Controls* chapter).

This method is not intended to be replaced.

DL_SET_ITEM_FLAGS Set item flags

```
VOID dl_set_item_flags(PR_LODGER *lodger, UINT flags);
```

OR the passed flags into the flags field of the dialog box item corresponding to the control with handle lodger.

This method is intended for use by a component control to set some aspect of the dialog's state. It is used, for example, by the FNSELWN and ACLIST dialog control classes.

The DLGBOX_ITEM_LOCKED and DLGBOX_ITEM_DIMMED flags should be set (or cleared) by use of the appropriate specific method. Note that, apart from these two flags, no means is provided for clearing any of an item's flags.

This method is not intended to be replaced.

DL_TAKE_FOCUS Move focus to specified item

```
INT dl_take_focus(INT index);
```

Attempt to change the focus to item number index.

A deferred consistency check is applied to the current item. If the check fails the focus change is not made. Otherwise, provided the specified item is not already the one with focus, the focus is switched to the specified item:

- any prompt of the item that is losing focus is sent a WN_EMPHASISE,FALSE message
- provided its flags do not contain DLGBOX_ITEM_DIMMED or DLGBOX_ITEM_LOCKED, the control of the item that is losing focus is also sent a WN_EMPHASISE,FALSE message
- if the flags of the item that is losing focus contain DLGBOX_ITEM_NOTIFY_FOCUS, the dialog box is sent a DL_FOCUS message to inform it that the item is losing focus
- a TRUE value is written to dlgbox.focus and dlgbox.current is set to index, so that it now refers to the item that is gaining focus
- any prompt of the item that is gaining focus is sent a WN_EMPHASISE,TRUE message
- provided the item that is gaining does not have the either of the flags DLGBOX_ITEM_DIMMED or DLGBOX_ITEM_LOCKED set, the control of this item is also sent a WN_EMPHASISE,TRUE message
- if the item that is gaining focus has the flag DLGBOX_ITEM_NOTIFY_FOCUS set, the dialog box is sent a DL_FOCUS message to inform it that the new item is gaining focus

The method returns FALSE if a failed consistency check prevents a change of focus being made. In all other circumstances (including the case where the specified item is already the one with focus) the method returns TRUE.

This method is not intended to be replaced. It is not suitable for being called from the dl_dyn_init method. If an application wishes to set the focus on initialisation, it should do so from a replaced dl_set_size method. It may be called from any other method (such as dl_key) once the dialog has been made visible.

DL_ITEM_ADD — Add an item

```
INT dl_item_add(AD_DLGBOX *par);
```

Append, as the current last line of the dialog box, the control specified by the AD_DLGBOX struct pointed to by par. This struct is defined in *dlgbox.g* as:

```
typedef struct
    {
    WORD flags;         /* initialisable DB_ITEM flags */
    UBYTE class;        /* class of item */
    TEXT prompt[1];     /* ZTS string */
    /* followed by an IN_XXX component-specific initialisation structure */
    } AD_DLGBOX;        /* header for each item in resource information */
```

It is a programming error, with unpredictable results, to add an item if the dialog box currently contains the maximum number of items.

The method first creates an instance of the class par->class, writing its handle to the appropriate element of the array indicated by dlgbox.item. If par->flags does not contain DLGBOX_ITEM_APPL_CAT the class is assumed to be a standard control in the HWIM category. Otherwise the method sends a DL_ITEM_NEW message to create the instance. The par->flags value is copied into the appropriate dlgbox.item array element.

If the string at the start of the par->prompt buffer is not a null string, an instance of TEXTWIN is created, writing its handle to the appropriate dlgbox.item array element. This instance is sent a WN_INIT message, passing the address of an IN_TEXTWIN struct and the dialog box handle. The IN_TEXTWIN struct contains the prompt string and a state value of 0 if par->flags contains any of DLGBOX_ITEM_DIMMED, DLGBOX_ITEM_LOCKED or DLGBOX_ITEM_DEAD, otherwise state is set to IN_TEXTWIN_BULLET (see also the *Text windows* section of the *Dialog Controls* chapter).

The method then sends a WN_INIT message to the item's control, passing a pointer to any data (assumed to be a suitable initialisation structure) that follows the terminating zero of the string in the par->prompt buffer. Further parameters are the dialog box handle and the handle of the control in the previous line[1] (see also the *Special note* below).

If par->flags contains DLGBOX_ITEM_UNDERLINED, the item will be drawn with an underline (only one underline is allowed on the Series 3).

Finally, the value of dlgbox.count is incremented by one.

The method returns zero to indicate that no failure has occurred. It is thus suitable for calling under the protection of p_enter.

This method may be either used or replaced by application writers. It is called by system code, once for each component control, from the WSERV ws_do_dial method after the dialog resource has been loaded from the resource file.

An application may call this method to add one or more items, depending on run-time circumstances, although the dl_item_append method - which also loads the item's resource - will usually be more appropriate. All such uses must be before the DL_SET_SIZE message is processed at the DLGBOX level. Application code will typically send a DL_ITEM_ADD message from the dl_dyn_init method.

[1]This will contain an indeterminate value when adding the first item, but the **wn_init** methods of most controls ignore this value. Those that make use of this parameter are never used as the first item in the dialog box.

Since the method is called by system code once for each item that appears in the dialog resource, it may be replaced to omit one or more items, or to add items at positions other than the end of the dialog, depending on run-time circumstances.

The following example optionally omits the third item (with index number 2) from a dialog. It assumes that the MYDLG subclass of DLGBOX adds two items of property: mydlg.needed, which is TRUE if the item is to be included, and mydlg.omitted which is initially FALSE and is set to TRUE if the item is not included in the dialog (an alternative would be to store this information in a result buffer, pointed to by dlgbox.rbuf).

```
METHOD VOID mydlg_dl_item_add(PR_MYDLG *self, AD_DLGBOX *par)
    {
    if ((self->dlgbox.count==2)&&(!self->mydlg.needed)&&(!self-
>mydlg.omitted))
        self->mydlg.omitted=TRUE;
    else
        p_supersend3(self,O_DL_ITEM_ADD,par);
    }
```

Special note

If par->flags contains DLGBOX_ITEM_NEEDS_PACK (normally only if adding a control of either the FNEDIT or the FNSELWN class) then this method will result in the addition of *two* controls, thus occupying two lines in the dialog box. After the creation of the first control, but before its initialisation, the method sends a further, recursive, DL_ITEM_ADD message to create a PACKSEL control with an associated "Disk" prompt string (this prompt is read from the SYS_PACK resource in the system resource file). In this case the final parameter passed in the WN_INIT message to the first control is the handle of the *following* control (that is, the instance of PACKSEL).

DL_ITEM_APPEND — Add an item by resource id

```
VOID dl_item_append(INT rid);
```

Append the item specified by the resource item with resource id rid.

The resource item is loaded into a temporarily allocated buffer, the loaded data being assumed to be an AD_DLGBOX struct. The control and any associated prompt are appended to the dialog box, as described for the dl_item_add method. The temporarily allocated buffer is freed before the method returns.

The method must not be used to add an item following any action list item.

An application may call this method to add one or more items at the bottom of a dialog, depending on run-time circumstances, for example, to generate two different, but similar, dialogs from a common dialog box resource. All such uses must be before the DL_SET_SIZE message is processed at the DLGBOX level. A call from application code to the dl_item_append method will typically be from the dl_dyn_init method.

This method is not intended to be replaced.

DL_ITEM_REPLACE — Replace an existing item

```
VOID dl_item_replace(INT index, INT rid);
```

Replace existing item number index by the item specified by the resource item with resource id rid.

The existing control and any corresponding prompt are sent DESTROY messages. The resource item is then loaded into a temporarily allocated buffer, the loaded data being assumed to be an AD_DLGBOX struct. The replacement item and any corresponding prompt are created and initialised as described for the dl_item_add method, their handles overwriting those that they replace. The method does not increment dlgbox.count. The temporarily allocated buffer is freed before the method returns.

This method must not be used with a resource which has the DLGBOX_ITEM_NEEDS_PACK flag set in its flags data.

An application may call this method to replace one or more items, depending on run-time circumstances, for example, to generate two different, but similar, dialogs from a common dialog box resource. All such uses must be before the DL_SET_SIZE message is processed at the DLGBOX level. A call from application code to the dl_item_replace method will typically be from the dl_dyn_init method.

This method is not intended to be replaced.

DL_INQ_MINSIZE — Specify minimum widths

```
VOID dl_inq_minsize(INT *pOverallWidth, INT *pPromptWidth, INT *pControlWidth);
```

Specify the minimum widths for whole line items (*pOverallWidth) and for the prompt (*pPromptWidth) and control (*pControlWidth) segments of two-part items. The method is called from dl_set_size and, on entry, all three parameters point to locations containing zero.

The supplied method does nothing.

A subclass may replace this method to write application-specific minimum values to any or all of *pOverallWidth, *pPromptWidth or *pControlWidth. Writing values that force a dialog to exceed the width of the screen will cause dl_set_size to call p_leave(E_GEN_TOOWIDE).

DL_SET_SIZE — Set size of dialog

```
INT dl_set_size(VOID);
```

Set the height and width of the dialog to the values required to display all the component items, and set the position all components within the dialog box. An LG_SENSE_WIDTH message is sent to each component to determine the width needed to display it.

If dlgbox.flags contains DLGBOX_APPEND_UNITS_TITLE the SYS_CENTIMETRES or SYS_INCHES resource (depending on whether w_ws->wserv.flags contains PR_WSERV_METRIC) is loaded from the system resource file and appended to the dialog title. This will call p_panic if the dialog has no title.

Some dialog items (such as an instance of TEXTWIN used as a title) are composed of a single control component, whereas others are formed from two components - a prompt and a control. The width of each component is found by sending it an LG_SENSE_WIDTH message. The dialog box width is set to contain the widest item from each of these two groups, subject to any explicit minimum widths set by sending each control a DL_INQ_MINSIZE message.

The width for single-component items is the larger of the width written to *pOverallWidth by dl_inq_minsize (zero by default) and the single-component item of greatest width.

The width for two-component items is the width of the widest prompt plus the width of the widest control (respectively not less than any values written to *pPromptWidth and *pControlWidth by dl_inq_minsize) plus a minimum gutter width separation between them. Provided at least one item is not marked with the DLGBOX_ITEM_DEAD flag, the prompt width allows space for the prompt to include a leading bullet to show that an item can be modified.

The width of the dialog box is the greater of these two widths plus an allowance for the dialog box borders and a small gap at either side. If this is wider than the screen, an attempt is made to clip the right hand side of the control components. If such clipping means that one or more controls will be entirely invisible the method calls p_leave(E_GEN_TOOWIDE).

The height of the dialog box is just the sum of the heights of the controls, plus the top and bottom borders and a small additional amount for each item that is underlined. If the total exceeds the screen height the method calls p_leave(E_GEN_TOOMANY). In most cases there will always be room for the maximum number of items, but some items (for example, instances of ACLIST) take up additional height.

The method then positions each prompt and control by sending it an LG_SET_ID_POS message. Unless marked as centred, prompts are positioned at the left side, aligned at their left edges, and controls occupy a right hand region, again aligned at their left edges. The prompt and control regions are aligned with the edges of the widest centred control, subject to their being separated by a small gap.

Finally, provided that not all items are marked with the DLGBOX_ITEM_DEAD flag, the method sets the focus to the first such unmarked item - dlgbox.focus is set TRUE, dlgbox.current is set to the item and a WN_EMPHASISE,TRUE message is sent to the item's prompt, if it exists. The control itself is not sent a WN_EMPHASISE message as it will receive one later, when the dialog box itself receives a WN_EMPHASISE message.

The method returns zero to indicate that p_leave has not been called. It is suitable for calling under the protection of p_enter.

This method may be replaced, but should not be called explicitly from application code. It offers the last opportunity to modify the dialog box content before it becomes visible. A common use is to modify the content after the size of the dialog box has been calculated.

Any replacement method should only add further processing, before and/or after supersending the `DL_SET_SIZE` message.

DL_DIMMED_MESSAGE — Display 'dimmed' message

```
VOID dl_dimmed_message(VOID);
```

Display a status message, using `hInfoPrint`, indicating that a locked or dimmed can not be modified. This method will only be called when the current dialog item is either dimmed or locked.

The resource id passed to `hInfoPrint` is `dlgbox.dimrid` or, if this is zero, either of the system resource id's `SYS_DIMMED_MSG` or `SYS_LOCKED_MSG` depending on whether or not the item's flags contain `DLGBOX_ITEM_DIMMED` (if this flag is not present the item is assumed to be locked, without testing for the presence of `DLGBOX_ITEM_LOCKED`).

This method can be replaced, for example, to display a context-sensitive message.

WN_SENSE_HELP — Sense start id for Help

```
INT wn_sense_help(PR_DLGBOX *self);
```

The action depends on the value of `dlgbox.helprid` as follows:

- return `dlgbox.helprid`, if it is greater than zero.
- if `dlgbox.helprid` is zero, return the result of supersending the `WN_SENSE_HELP` message. This is handled by the `WIN` superclass and returns either `w_ws->wserv.help_index_id` or, if this is zero, the (negative) system resource id `-SYS_HELP_ON_HELP`.
- if `dlgbox.helprid` is -1, call `p_leave(RUN_ACTIVE_USED)`. This is used by the `ERRORDLG` class, to disable the display of help information when an error is being reported (so that, for example, multiple nested out of memory errors can not be generated by requesting Help while an out of memory error report is visible).

WN_KEY — Handle a keypress

```
INT wn_key(INT keycode, INT modifiers);
```

The description of this method is included for interest only. Applications should neither replace this method nor call it explicitly.

Handle a keypress, directing it, if necessary, to a component control. This message is sent by the application's instance of `WSERV`. Any non-zero return value from the `wn_key` method will terminate the dialog, causing the dialog box to receive a `DESTROY` message.

If `dlgbox.absorb` is `TRUE`, all keys are directed to the current control, as described later. Otherwise keys are processed as follows.

If the dialog contains an action list (which must always be the last item) the action list is sent a `WN_KEY` message, passing a key code of `p_toupper(keycode)`. This message returns either a negative value if the keypress does not match any of the action buttons, or the index number of the button that matched the keypress (the leftmost button has index number zero).

If there was no match and `dlgbox.flags` does not contain `DLGBOX_NOTIFY_ALL_ACT` , processing continues with the testing for specific keys, as described later.

Otherwise (that is, either the keypress matches an action button, or `dlgbox.flags` contains `DLGBOX_NOTIFY_ALL_ACT`) a forced consistency check is applied to all controls. If any control fails its check the method terminates, returning `WN_KEY_NO_CHANGE` (0) so that the dialog box is not exited. If the consistency check succeeeds, the method sends a `DL_KEY` message, passing `dlgbox.current`, `p_toupper(keycode)` and the key-matching value returned by the `WN_KEY` message that was sent to the

action list. The method terminates, returning the value returned by the DL_KEY message. Before returning, and provided that:

- the return value is non-zero (that is, the dialog is terminating)
- dlgbox.rbuf is not NULL
- dlgbox.flags does not contain DLGBOX_RBUF_FILLED

a WORD of data is written to *dlgbox.rbuf. If dlgbox.flags contains DLGBOX_REPORT_ACT_HORIZ this data is the index number of the action button that was selected. Otherwise the value written to *dlgbox.rbuf is the uppercased key code.

Provided the keypress has not been processed by an action list, the following specific keypress codes are tested:

W_KEY_RETURN — this key is ignored if the dialog contains an action list (dlgbox.flags contains DLGBOX_ACTION_LIST or DLGBOX_SMALL_ACTION_LIST) in which case the method terminates immediately, returning WN_KEY_NO_CHANGE. Otherwise a forced consistency check is applied to all controls. If any control fails its check the method terminates, returning WN_KEY_NO_CHANGE.

If dlgbox.flags does not contain DLGBOX_NOTIFY_ENTER the method just returns WN_KEY_CHANGED, otherwise it returns the result from sending a DL_KEY message. In either case, provided:

- the return value is non-zero (that is, the dialog is terminating)
- dlgbox.rbuf is not NULL
- dlgbox.flags does not contain DLGBOX_RBUF_FILLED

the value of dlgbox.current is written to the WORD pointed to by dlgbox.rbuf.

W_KEY_ESCAPE — provided dlgbox.rbuf is not NULL and dlgbox.flags does not contain DLGBOX_RBUF_FILLED write, to the WORD pointed to by dlgbox.rbuf, either W_KEY_ESCAPE or, if dlgbox.flags contains DLGBOX_REPORT_ACT_HORIZ, a value of -1

W_KEY_UP — move up 'one' item in the dialog. The current item becomes the first earlier item whose flags do not contain DLGBOX_ITEM_DEAD. A DL_TAKE_FOCUS message is sent, passing the new current item's index number. Moving up from the first item in the dialog positions to the last item

W_KEY_DOWN — move down 'one' item in the dialog. The current item becomes the first following item whose flags do not contain DLGBOX_ITEM_DEAD. A DL_TAKE_FOCUS message is sent, passing the new current item's index number. Moving down from the last item in the dialog positions to the first item

W_KEY_PAGE_UP — move to the 'first' item in the dialog. The current item becomes the first item whose flags do not contain DLGBOX_ITEM_DEAD. A DL_TAKE_FOCUS message is sent, passing the new current item's index number.

W_KEY_PAGE_DOWN — move to the 'last' item in the dialog. The current item becomes the last item whose flags do not contain DLGBOX_ITEM_DEAD. A DL_TAKE_FOCUS message is sent, passing the new current item's index number.

If dlgbox.focus is FALSE, any other keypress is ignored and the method returns WN_KEY_NO_CHANGE. Otherwise the keypress is directed to the dialog's current component control, as explained in the following paragraphs. All incoming keys are processed in this way if dlgbox.absorb is TRUE.

If the flags field of the current item contains DLGBOX_ITEM_DIMMED or DLGBOX_ITEM_LOCKED, a DL_DIMMED message is sent, to inform the user that the item can not be modified, and the method then returns WN_KEY_NO_CHANGE.

Otherwise the current control is sent a WN_KEY message, and further processing depends on the return value:

- if the return value is WN_KEY_ABSORB_ON, dlgbox.absorb is set TRUE, so that all subsequent keys will be directed to the current control
- if the return value is anything other than WN_KEY_NO_CHANGE and dlgbox.absorb is TRUE, dlgbox.absorb is set FALSE, so that subsequent keys may be processed by the dialog box as described above
- if the return value is WN_KEY_CHANGED and the current item's flags contain DLGBOX_ITEM_NOTIFY_CHANGED, the dialog box is sent a DL_CHANGED message

In all these cases the dialog wn_key method returns WN_KEY_NO_CHANGE.

Deferred DLGBOX methods

There is no general requirement to subclass DLGBOX to provide any of these DEFERred methods. Each method need be supplied only if the conditions are satisfied for the corresponding message to be received.

DL_CHANGED Item changed message

```
VOID dl_changed(INT index);
```

Notify the dialog box that the item with index number index has changed in some way.

This message will only be received for values of index for which the flags field of the corresponding item contains DLGBOX_ITEM_NOTIFY_CHANGED. The method need not be supplied for any dialog in which no items are so marked.

A typical use would be to modify an item - say the allowed range of a numeric edit box - as a result of changes made by the user to some other item.

DL_FOCUS Focus changed message

```
VOID dl_focus(WORD index, WORD flag);
```

Notify the dialog box that the item with index number index has gained or lost focus. The value of flag is TRUE if the item has gained focus, or FALSE if the item has lost focus.

This message will only be received for values of index for which the flags field of the corresponding item contains DLGBOX_ITEM_NOTIFY_FOCUS. The method need not be supplied for any dialog in which no items are so marked.

A common use is to detect when an edit box loses focus. This may be an appropriate time to sense the value and make any necessary modifications to other dialog box items.

DL_LAUNCH_SUB Launch sub-dialog if required

```
VOID dl_launch_sub(INT index);
```

This message is sent by the window server object if, on return from a wn_key message sent to the dialog, the value of w_ws->wserv.subdial is non-zero. The value of index is the item number of the item that is launching the subdialog, *plus one*.

The method need not be supplied for any dialog that does not write a non-zero value to w_ws->wserv.subdial. For further information, see *Subdialogs* in the following *Using dialog boxes* section.

DL_ITEM_NEW Create non-system dialog item

```
VOID *dl_item_new(AD_DLGBOX *par);
```

Create an instance of the application-specific dialog item with class number par->class.

Assuming that the application category file is *myapp.cat*, the code of a dl_item_new method for a dialog box that has application-specific items defined only in this category would be:

```
f_new(CAT_MYAPP_MYAPP,par->class);
```

If, exceptionally, a dialog box contains two or more application-specific items, with classes defined in different categories, the method will need additional logic to create the item from the appropriate category.

This message will only be received if the flags field associated with one or more of the items in a dialog box contains `DLGBOX_ITEM_APPL_CAT`, indicating that the item does not have its class definition in the HWIM category. The method need not be supplied for any dialog whose items are not so marked.

Using dialog boxes

In all cases, the dialog must ultimately be started up by means of the `WSERV` `ws_do_dial` method. This may, however, be indirect, such as when using the `hLaunchDial` utility function or an equivalent application-specific function.

All sample code in this section assumes that the application category file is *myapp.cat*.

Default dialog behaviour

One item in the dialog box generally has *focus*, that is, it receives all keys (via a `WN_KEY` message to its control) directed to the dialog box, except for those, listed below, that are processed by the dialog box itself. On receipt of a key, a control may elect to absorb all further keys directed to the dialog box.

Provided that one of the dialog's component controls has not elected to absorb all keys directed to the dialog box:

- the up and down cursor keys respectively move focus to the previous or the next item in a cyclic fashion
- the page up and page down keys respectively move the focus to the first or the last item that is capable of receiving focus
- the Enter key terminates the dialog. If the dialog has a result buffer (that is `dlgbox.rbuf` is not `NULL`) the value of `dlgbox.current` is written to the first `WORD` of this buffer
- the Esc key terminates the dialog. If the dialog has a result buffer (that is `dlgbox.rbuf` is not `NULL`) a value of `W_KEY_ESCAPE` is written to the first `WORD` of this buffer

If no control is absorbing all keys and the dialog contains an action list of buttons as its last item, this behaviour is modified. In this case all incoming keys are first offered to the action list and if the key matches one of the buttons in the action list the dialog is terminated. If the dialog has a result buffer (that is `dlgbox.rbuf` is not `NULL`) the uppercased key code is written to the first `WORD` of this buffer. Unless it matches a button in the action list, the Enter key is ignored. Only if the key is not recognised by the action list is it then offered for processing as described above.

This action may be further varied by other `dlgbox.flags` values as follows:

- if the `DLGBOX_NOTIFY_ALL_ACT` flag is set, keys that do not match a button in the action list also cause the dialog to terminate
- if the `DLGBOX_REPORT_ACT_HORIZ` flag is set, the value written to the result buffer will be the index of the matching button (the leftmost button has an index of zero, and non-matching keys give a value of -1) rather than the key code

Other variations are generally accomplished by subclassing `DLGBOX`, and some such variations are described later.

Dialogs and resource files

The content of a dialog is normally specified by a resource file item, using dialog box resource file structures that are defined in *hwim.rh*. These resources also use definitions of constants (in *hwim.rg*) that are derived from the HWIM category file. All resource files that contain dialog resources must contain the following two lines before the definition of any dialog resource:

```
#include <hwim.rh>
#include <hwim.rg>
```

The `DIALOG` resource struct is defined in *hwim.rh* as:

```
STRUCT DIALOG
    {
    WORD flags=0;
    TEXT title="";
    LEN BYTE STRUCT controls[];    /* array of CONTROL resource items only */
    }
```

and the CONTROL resource struct is defined as:

```
STRUCT CONTROL
    LEN {
    WORD flags=0;
    BYTE class;     /* class of control */
    TEXT prompt=""; /* Prompt for item */
    STRUCT info;    /* eg CHLIST, TXTMESS, EDWIN, or NCEDIT */
    }
```

A simple dialog, consisting of a title, a fixed message and a 'Continue' button, might be defined by the following resource, assumed to be in a *myapp.rss* source file:

```
RESOURCE DIALOG simple_dialog
    {
    flags=DLGBOX_NO_DDP;
    title="Dialog title"; /* an empty string means that the dialog has no
title */
    controls=
        {
        CONTROL
            {
            class=C_TEXTWIN;
            flags=DLGBOX_ITEM_CENTRE|DLGBOX_ITEM_DEAD;
            info=TXTMESS
                {
                flags=IN_TEXTWIN_AL_CENTRE;
                str="This is a message";
                };
            },
        CONTROL
            {
            class=C_ACLIST;
            info=ACLIST
                {
                rid=-SYS_AC_CONTINUE; /* the id of a system action list
resource */
                };
            }
        };
    }
```

In use, this will display (in compatibility mode on the Series 3a) the following dialog, which exits when the user presses the Esc key.

The dialog resource uses a one-button action list defined by the the system resource SYS_AC_CONTINUE (see the *HWIM Resource Files* chapter) and is based on the following *hwim.rh* resource structs:

```
STRUCT TXTMESS /* Initialising struct for a text window */
    {
    WORD flags=0;
    TEXT str="";      /* message defaults to empty string */
    }
```

```
STRUCT ACLIST  /* Initialising struct for action list */
    {
    LINK rid;
    }
```

together with the DIALOG and CONTROL structs, described earlier.

Note that, in general, a dialog resource contains three levels of flags data:

- the highest level, associated with the whole dialog box, can contain a combination of the DLGBOX_XXX flags that may appear in dlgbox.flags. Exceptionally, the example sets DLGBOX_NO_DDP. This is done because the dialog does not use any dialog utility functions, so there is no advantage in writing its handle to DatDialogPtr (but it would do no harm to omit this flag). Although the dialog contains an action list as its last item, there is no need to set DLGBOX_ACTION_LIST since this happens automatically
- the second level, associated with a particular control, can be a combination of the DLGBOX_ITEM_XXX flags that may appear in the flags field of each of the dialog's component items. In the above example, the text message control is to be centred in the dialog box and not selectable with the cursor keys
- finally, there may be a set of flags (usually IN_XXX) specific to the initialisation of the particular class of which the control is an instance. In the example, the IN_TEXTWIN_AL_CENTRE flag ensures that the text is centre aligned in its control (a text window)

Launching a dialog

A dialog is launched by means of the WSERV ws_do_dial method. This loads the dialog data from a resource file, and then initialises and runs the dialog. It is used as shown below:

```
p_send5(w_ws,O_WS_DO_DIAL,p_getlibh(cat),class,pdata);
```

where cat and class are respectively the category and class numbers of the dialog box class (DLGBOX or a subclass of DLGBOX) that is to be run, and pdata is a pointer to a DL_DATA struct, defined in *hwimman.g* as:

```
typedef struct
    {
    UWORD id;           /* resource id of a DIALOG resource*/
    VOID *rbuf;         /* address of result buffer, or NULL */
    PR_DLGBOX **pdlg;   /* address of where to write handle of dialog, or NULL
*/
    } DL_DATA;
```

An alternative is to use the hLaunchDial utility function:

```
INT hLaunchDial(P_CATID cat, INT class, DL_DATA *pdata);
```

Examples of the use of this function appear in the following text.

Simple dialogs

A simple dialog, for the purposes of this section, is one that uses the DLGBOX class, rather than subclassing it. Such a dialog is easy to define and run, but suffers from the following limitations:

- the initial values it displays are entirely determined by the (static) resource file data: the dialog data may not be determined dynamically from current values stored in the application
- knowledge of the final state of the dialog is limited to the default information that is written to a result buffer - effectively only indicating which keypress terminated the dialog

Despite these restrictions, such dialogs may be useful, say, to make a specific warning with a characteristic appearance, or to elicit multiple choice responses (but bear in mind the system-supplied Error and Query dialogs, run by WSERV methods).

As an example, the dialog defined earlier may be run using code as follows:

```
#include <hwimman.g>
#include <dlgbox.g>
#include <myapp.rsg> /* resource file generated header file */

LOCAL_C VOID RunDlgNoResponse()
    {
    DL_DATA data;

    data.id=SIMPLE_DIALOG;
    data.rbuf=NULL;
    data.pdlg=NULL;
    hLaunchDial(CAT_MYAPP_HWIM,C_DLGBOX,&data);
    }
```

Should you wish to know whether the dialog was terminated by pressing Enter or Esc, you could use the following alternative:

```
INT RunDlgWithResponse()
    {
    WORD result;
    DL_DATA data;

    data.id=SIMPLE_DIALOG;
    data.rbuf=&result;
    data.pdlg=NULL;
    hLaunchDial(CAT_MYAPP_HWIM,C_DLGBOX,&data);
    return(result);
    }
```

The return value is `W_KEY_ESCAPE` only if the dialog was exited by pressing Esc.

To go beyond the range of possibilities discussed above, `DLGBOX` must be subclassed. In the majority of cases this will involve no more than supplying replacements for one or both of the `dl_dyn_init` and `dl_key` methods.

Dynamically initialised dialogs

The `dl_dyn_init` method is intended to be used for setting the initial values of dialog box controls dynamically, as opposed to the static initialisation, from data in a resource file, used by simple dialogs.

Suppose, for example that an application needs to use a dialog, similar to the one described earlier, but able to display one of two alternative text messages. Such a dialog might use two string resources and a dialog resource as follows:

```
RESOURCE STRING dial_msg_1 { str="Message one"; }
RESOURCE STRING dial_msg_2 { str="Message two"; }
```

```
RESOURCE DIALOG message_dialog
    {
    controls=
        {
        CONTROL /* the title */
            {
            class=C_TEXTWIN;
            flags=DLGBOX_ITEM_CENTRE|DLGBOX_ITEM_DEAD|DLGBOX_ITEM_UNDERLINED;
            info=TXTMESS
                {
                flags=IN_TEXTWIN_AL_CENTRE;
                str="Dialog title";
                };
            },
        CONTROL /* the message */
            {
            class=C_TEXTWIN;
            flags=DLGBOX_ITEM_CENTRE|DLGBOX_ITEM_DEAD;
            info=TXTMESS
                {
                flags=IN_TEXTWIN_AL_CENTRE;
                };
            },
        CONTROL
            {
            class=C_ACLIST;
            info=ACLIST
                {
                rid=-SYS_AC_CONTINUE;
                };
            }
        };
    }
```

Note that the control that is to contain the message does not specify a message, and so has the default of an empty string, since the message text is always replaced.

In addition, this dialog resource uses an alternative way of specifying the dialog title, compared with the previous example. Although it takes up a few more bytes than in the previous case, it allows the possibility for the title to be left as the default empty string in cases where, for example, the dialog title itself is to be generated.

The dialog could subclass `DLGBOX` as follows:

```
CLASS msgdial dlgbox
    {
    REPLACE dl_dyn_init
    }
```

with the corresponding message function to set the message text:

```
METHOD VOID msgdial_dl_dyn_init(PR_MSGDIAL *self)
    {
    INT flag;
    TEXT buf[40];

    flag=*(WORD *)self->dlgbox.rbuf;
    hLoadResBuf(flag?DIAL_MSG_1:DIAL_MSG_2,&buf[0]);
    hDlgSetText(1,&buf[0]);
    }
```

This method uses the dialog utility function `hDlgSetText` to set the text for the TEXTWIN item with index number 1 (the message). This function, which is described in the *Dialog box utilities* section of the *HWIM Utility Functions* chapter, assumes that the dialog's handle is stored in `DatDialogPtr`. Thus, in contrast with the previous example, the resource for this dialog must *not* set the `DLGBOX_NO_DDP` flag.

In this case, the flag to select the required message string is passed to the dialog code in the dialog box result buffer. A possible means of running this dialog is:

```
LOCAL_C INT RunMessageDlg(INT flag)
    {
    WORD result;
    DL_DATA data;

    result=flag;
    data.id=SIMPLE_DIALOG;
    data.rbuf=&result;
    data.pdlg=NULL;
    hLaunchDial(CAT_MYAPP_MYAPP,C_MSGDIAL,&data);
return(result);
    }
```

On completion of the dialog, the `WORD` pointed to by `data.rbuf` (that is, `result`) will be overwritten, as in the previous example, with a value indicating how the dialog was terminated.

A dialog that needs to dynamically initialise several controls may use the above technique with a `dlgbox.rbuf` that points to a structure containing the initialisation data for all the controls, although this may require a considerable amount of code to set up the data. An alternative would be to allow the `dl_dyn_init` method to obtain its data directly from other objects (ideally, via sensing methods) or from static variables. A possible technique is to use the result buffer pointer to point to a particular structure, which may be in the property of some other object. The method chosen in a particular circumstance may depend on the trade-off between such factors as the amount of code needed, the clarity of the code and the preservation of modularity.

Retrieving dialog results

In general, it is not sufficient merely to know which keypress caused the dialog to terminate. Most dialogs gather information from the user and this information must be communicated to the rest of the application. Such a dialog will generally set `DLGBOX_RBUF_FILLED` in `dlgbox.flags`, to prevent system code from writing to `*dlgbox.rbuf`. The dialog may then safely write its own specific results to that location. The collection of the results is normally done by a replacement `dl_key` method.

The `dl_key` method is called from the `wn_key` method when the dialog box is potentially about to terminate, in the following circumstances:

- when the keypress `W_KEY_RETURN` is received, provided `dlgbox.flags` contains `DLGBOX_NOTIFY_ENTER`, but *not* `DLGBOX_ACTION_LIST` or `DLGBOX_SMALL_ACTION_LIST`
- when the keypress `W_KEY_ESCAPE` is received, provided `dlgbox.flags` contains `DLGBOX_NOTIFY_ESCAPE`
- the dialog contains an action list, `dlgbox.flags` contains `DLGBOX_ACTION_LIST` (or `DLGBOX_SMALL_ACTION_LIST`) and a keypress matches one of the buttons in the action list
- for all keypresses, provided the dialog contains an action list and `dlgbox.flags` contains `DLGBOX_NOTIFY_ALL_ACT` as well as `DLGBOX_ACTION_LIST` (or `DLGBOX_SMALL_ACTION_LIST`)

A typical `dl_key` method will sense the data in one or more of the dialog's component controls, write this data into a buffer or structure pointed to by `dlgbox.rbuf` and return `WN_KEY_CHANGED`. The choices available for transferring information to other objects within the application are similar to those already discussed for the `dl_dyn_init` method.

The method may test the keypress that caused it to be called, since this is passed as a parameter. It may, as a result of this test - or of other tests on the values of one or more component controls - return the value `WN_KEY_NO_CHANGE` to indicate that the dialog box should not be terminated.

Dialogs with and without 'WAIT'

By default, the `WSERV` `ws_do_dial` method (and hence the `hLaunchDial` utility function) will not return until the dialog terminates and the dialog box has been destroyed.

The processing of the dialog's results may therefore be divided between the `dl_key` method (which could, in this case, be viewed as a simple collector of the data) and code that immediately follows a call to, say, `hLaunchDial`. Most of the system-supplied dialogs use this technique since it allows application-specific code to follow the generic processing performed in the dialog's `dl_key` method.

If `dlgbox.flags` contains `DLGBOX_NO_WAIT`, the `ws_do_dial` method (and `hLaunchDial`) will return as soon as the dialog is launched, rather than waiting until the dialog is destroyed. In such a case none of the processing of the dialog completion can be performed by code following a call to, say, `hLaunchDial`, since

the dialog is still running at that time. All the completion processing must be done in the dialog's `dl_key` method.

Although this technique is more difficult to handle, it does have a number of advantages for the application writer, one of the more significant being that it is less expensive in terms of stack use.

Controlling the width of a dialog

The width of a dialog is normally set - following its initialisation and before it is made visible, in the `dl_set_size` method - so that it exactly fits the widest of its components. In cases where the size of a component may vary after the dialog becomes visible, this may not be adequate.

An example of such a situation would be a dialog containing a centred text message that shows a page count used, say, to report the progress of document printing. The message may have to display page numbers up to 999, but would normally start at page 1. If the dialog box width is determined for the initial message "Page 1", it may be too narrow to display "Page 999". Two possible solutions are described below.

If the maximum width of an item can be easily determined, the dialog can be forced to the required width by replacing the `dl_inq_minsize` method. Using the above example, the dialog resource could contain the following initialisation data for a page count control:

```
...
CONTROL
    {
    class=C_TEXTWIN;
    flags=DLGBOX_ITEM_CENTRE|DLGBOX_ITEM_DEAD;
    info=TXTMESS
        {
        flags=IN_TEXTWIN_AL_CENTRE;
        str="Page 1";
        };
    },
...
```

A suitable `dl_inq_minsize` method could be of the form:

```
METHOD VOID mydlg_dl_inq_minsize(PR_MYDLG *self,INT *pCent,INT *pPmpt,INT
*pCtl);
    {
    *pCent=gTextWidth(WS_FONT_SYSTEM,G_STY_NORMAL,"Page 999",8);
    }
```

An alternative solution is to replace the `dl_set_size` method itself. In the present example this is possibly a better solution, since it reuses the code used elsewhere to set the page number. In this case the resource file initialises the control to its maximum size:

```
...
CONTROL
    {
    class=C_TEXTWIN;
    flags=DLGBOX_ITEM_CENTRE|DLGBOX_ITEM_DEAD;
    info=TXTMESS
        {
        flags=IN_TEXTWIN_AL_CENTRE;
        str="Page 999";
        };
    },
...
```

Assuming that this control immediately follows a dialog title (that is, it has an index number of 1) and that the resource file also contains a string resource of the form:

```
RESOURCE STRING page_num_str {str="Page %u";}
```

the required code could then be:

```
LOCAL_C VOID SetPage(INT pagenum)
    {
    TEXT buf[10];

    hAtos(&buf[0],PAGE_NUM_STR,pagenum);
    hDlgSetText(1,&buf[0]);
    }

#pragma METHOD_CALL

METHOD VOID mydlg_dl_set_size(PR_MYDLG *self);
    {
    p_supersend2(self,O_DL_SET_SIZE); /* sets width for largest case */
    SetPage(1);                       /* now set initial page number value */
    }
```

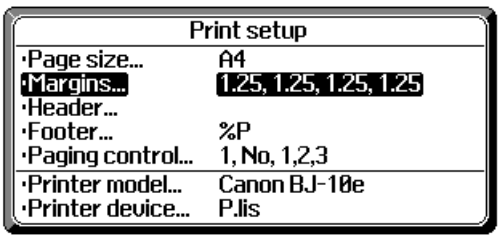
On first being made visible the dialog displays the text "Page 1" but is wide enough to display "Page 999".

Subdialogs

A dialog box may contain one or more items that can 'explode' into a separate subdialog, displayed over the main dialog. An example is the Margins line of the Print setup dialog as used, for example, in the Word application. The `CONTROL` resource for this dialog item is as follows:

```
...
CONTROL
    {
    class=C_TEXTWIN;
    prompt="Margins"<WS_SYMBOL_ELLIPSIS>;
    info=TXTMESS
        {
        str="";
        flags=IN_TEXTWIN_POPOUT;
        };
    },
...
```

The dialog is shown, with the Margins item highlighted, in the following diagram:

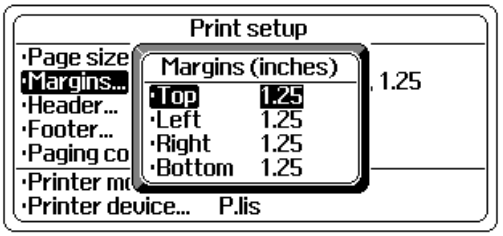


The significant points about the above `CONTROL` resource are that the control is an instance of the `TEXTWIN` class and that it is initialised with the `IN_TEXTWIN_POPOUT` flag. By convention, the prompt (which is also an instance of `TEXTWIN`) for such an item terminates with an ellipsis.

The text of the control, by convention, shows a summary of the current values of the information that may be modified by the subdialog and should therefore be set dynamically, in either the `dl_dyn_init` or the `dl_set_size` methods. Which method is the most appropriate depends on the nature of the summary text to be displayed. If it is of fixed width the resource text may be a null string (as it is in the above example) and the replacement text may safely be set in the `dl_dyn_init` method. If the summary text is of variable size, the resource file should contain a string that is guaranteed to be the longest that can be displayed. This text should be replaced in the `dl_set_size` method, after superseding the `DL_SET_SIZE` message, so that the dialog is guaranteed to be wide enough. A `CONTROL` resource that uses this technique is shown below.

```
...
CONTROL
    {
    class=C_TEXTWIN;
    prompt="Font"<WS_SYMBOL_ELLIPSIS>;
    info=TXTMESS
        {
        str="MMMMMMMMMMMMMMMMMMMM 00"; /* max font name + space + max font
size */
        flags=IN_TEXTWIN_POPOUT;
        };
    },
...
```

The `IN_TEXTWIN_POPOUT` flag ensures that the dialog item will respond to the Tab key by sending the dialog a `DL_LAUNCH_SUB` message (and to any other key by displaying the `SYS_POPOUT_HELP` information message which, in English, is "Press Tab to change this item"). The main dialog's `dl_launch_sub` method launches the subdialog in exactly the same way as any other dialog is launched, that is by means of either the window server object's `ws_do_dial method`, or the `hLaunchDial` utility function. The following diagram illustrates the Margins subdialog launched from the Print setup dialog.



A parameter to `dl_launch_sub` indicates which dialog item launched the subdialog. Note that this parameter is one greater than the index number of the corresponding dialog item. When launching the Margins subdialog, for example, this parameter has a value of 2.

Processing within the subdialog is no different from that needed for a main dialog. On completion of the subdialog, the return to the main dialog is handled automatically by system code.

CHAPTER 8

DIALOG CONTROLS

This chapter describes the use of the dialog controls provided by the HWIM object library. These controls are used as components of dialog boxes, allowing a wide variety of dialogs to be constructed. The following diagram, for example, shows a simple dialog constructed from a title and a prompted numeric editor control.

A control is implemented as an instance of a dialog control class: dialog control classes are not usually subclassed since the base class usually provides sufficient functionality and flexibility for most dialogs. Dialog control classes ultimately subclass the `LODGER` class and thus inherit its property and methods.

The initial appearance and behaviour of a dialog control are primarily determined by the (static) content of a `CONTROL` resource (in the application resource file). They may be dynamically modified by an optional `WN_SET` message sent from the `dl_dyn_init` method of the dialog box itself. The use of dynamic initialisation allows the control to be modified to take into account the current state of the application (in the above example the initial value will have been set to correspond to the position of an item in the current to-do list).

The preferred method of sensing and setting a dialog control is to use the `wn_sense` and `wn_set` methods of the dialog box, as in the following code fragment:

```
p_send4(DatDialogPtr,O_WN_SET,control_index,pset)
```

The control is identified by its index `control_index`. It is passed a pointer, `pset`, to an appropriate `SE_XXX` struct that holds the replacement data. The reserved static `DatDialogPtr` is assumed to point to the dialog; an assumption that is made throughout this chapter. It will always be true unless the dialog that owns the control sets the `PR_WIN_NO_DDP` flag in its `win.flags` property (see the *Dialogs* chapter for further details).

The alternative method of sensing/setting a dialog control is to use the `wn_sense` and `wn_set` methods of the particular dialog control, as in the following example code fragment.

```
PR_LODGER *hand;

hand=(PR_LODGER *)p_send3(DatDialogPtr,O_DL_INDEX_TO_HANDLE,control_index)
p_send3(hand,O_WN_SET,pset)
```

The first method is clearly preferable.

On termination of a dialog, the associated data can be sensed by means of a `wn_sense` method call from within the dialog box `dl_key` method. This is the normal practice for any but the simplest of dialogs.

The HWIM object library also includes some convenient utility functions that can perform the more common dialog control sensing and setting actions. The full set of available functions is described in the *Dialog box utilities* section of the *HWIM Utility Functions* chapter.

Each type of component control has an associated `SE_XXX` struct, used for both setting and sensing the control's data. In almost all cases the control allows selective setting of its data: which items of data that are set is determined by the value of a flags field in the appropriate `SE_XXX` struct. The flags field has no effect on sensing: the `wn_sense` method always senses all relevant data.

Text windows

The *textwin.g* header file should be included when using a text window control.

A TEXTWIN text window control is widely used to provide dialog titles, prompts for controls, and simple messages (and exceptionally to launch sub-dialogs).

The above dialog could be created with the following DIALOG resource.

```
RESOURCE DIALOG demodlg
    {
    title="Hello";
    flags=0;
    controls=
        {
        CONTROL
            {
            class=C_TEXTWIN;
            flags=DLGBOX_ITEM_CENTRE|DLGBOX_ITEM_DEAD;
            info=TXTMESS
                {
                flags=IN_TEXTWIN_AL_CENTRE;
                str="message";
                };
            }
        };
    }
```

The first item in this dialog is the title and is specified by the resource line.

```
    title="Hello";
```

This creates an underlined centred item, with index zero, displaying the appropriate text.

Note that the title is, in fact, simply a TEXTWIN control with index zero. The above line is exactly equivalent to including the following CONTROL resource as the first item in the dialog:

```
RESOURCE CONTROL demonstration
    {
    class=C_TEXTWIN;
    flags=DLGBOX_ITEM_CENTRE|DLGBOX_ITEM_DEAD|DLGBOX_ITEM_UNDERLINED;
    info=TXTMESS
        {
        flags=IN_TEXTWIN_AL_CENTRE;
        str="Hello";
        };
    }
```

As a result the title can easily be replaced dynamically using the hDlgSetText utility function called from within, say, the dl_dyn_init method of the dialog.

The second item in the demodlg resource is a simple text window and is specified by the CONTROL struct:

```
        CONTROL
            {
            class=C_TEXTWIN;
            flags=DLGBOX_ITEM_CENTRE|DLGBOX_ITEM_DEAD;
            info=TXTMESS
                {
                flags=IN_TEXTWIN_AL_CENTRE;
                str="message";
                };
            }
```

where the IN_TEXTWIN_AL_CENTRE flag ensures that the text is centred within the control. Note also the DLGBOX_ITEM_CENTRE flag that ensures that the control itself is centred within the dialog.

A TEXTWIN control can also have a prompt as shown in the following example.

This dialog could be specified with the following resource:

```
RESOURCE DIALOG demodlg
    {
    title="Hello";
    flags=0;
    controls=
        {
        CONTROL
            {
            class=C_TEXTWIN;
            flags=DLGBOX_ITEM_DEAD;
            prompt="Prompt";
            info=TXTMESS
                {
                flags=IN_TEXTWIN_AL_CENTRE;
                str="message";
                };
           }
        };
    }
```

As in this example, the text of a prompt is specified by a line of the form:

```
prompt="Prompt";
```

The text of a prompt is left aligned automatically, and is normally preceded by a bullet symbol , to indicate that the content of the corresponding control can be modified. The bullet symbol can be removed permanently by setting the DLGBOX_ITEM_DEAD flag.

Specifying prompt text does not necessarily mean that the prompt will appear. If, in addition to a prompt, a control also has the DLGBOX_ITEM_CENTRE flag set (as in the earlier example) the prompt will be suppressed.This is, however, an abnormal case and is not a recommended technique, since it can lead to unexpected behaviour. A dialog set up in this way, and with the DLGBOX_ITEM_DEAD flag left clear, has the rather strange appearance shown in the following diagram.

By initialising a text window used as dialog box control with the IN_TEXTWIN_POPOUT flag, it may be used to trigger a subdialog, as described in the previous chapter.

Initialisation

The initial content and appearance of a text window control are specified by means of a TXTMESS resource struct:

```
STRUCT TXTMESS
    {
    WORD flags=0;
    TEXT str="";
    }
```

The str element specifies the initial text and flags may contain any sensible ORed combination of:

IN_TEXTWIN_AL_LEFT	align text left
IN_TEXTWIN_AL_RIGHT	align text right
IN_TEXTWIN_AL_CENTRE	centre text
IN_TEXTWIN_BOLD	text in bold typeface
IN_TEXTWIN_POPOUT	launch a subdialog on receipt of a Tab key

Setting

The SE_TEXTWIN struct is used for setting and sensing text window property.

```
typedef struct
    {
    INT flags;
    UWORD state;
    TEXT *buf;
    UWORD len;
    }SE_TEXTWIN;
```

When setting a text window, a sensible combination of the following values should be ORed into the flags field, to indicate which aspects are to be set (or cleared).

SE_TEXTWIN_ALIGN	setting or clearing an alignment flag
SE_TEXTWIN_BOLD	setting or clearing the bold typeface flag
SE_TEXTWIN_TEXT	setting the text

For each flag that is *not* set, the corresponding data will not be changed by the wn_set method.

If setting the text, a pointer to the replacement text must be written to *buf and the length of the text must be written to len.

Any sensible combination of the following flags may be placed in the state element:

PR_TEXTWIN_AL_LEFT	text is left aligned
PR_TEXTWIN_AL_RIGHT	text is right aligned
PR_TEXTWIN_AL_CENTRE	text is centred
PR_TEXTWIN_BOLD	text is in a bold typeface

The following example sets, for the item with item number index (assumed to be a text window) the text to the string pointed to by str. It also clears any PR_TEXTWIN_BOLD flag and sets PR_TEXTWIN_AL_RIGHT (clearing PR_TEXTWIN_AL_LEFT and PR_TEXTWIN_AL_CENTRE in the process). No other flags that may exist in the text window's property are affected.

```
LOCAL_C VOID SetTextWin(INT index,TEXT *str)
    {
    SE_TEXTWIN set;

    set.flags=SE_TEXTWIN_TEXT|SE_TEXTWIN_ALIGN|SE_TEXTWIN_BOLD;
    set.state=PR_TEXTWIN_AL_RIGHT; /* PR_TEXTWIN_BOLD is not set, so will be
cleared */
    set.buf=str;
    set.len=p_slen(str);
    p_send3(DatDialogPtr,O_WN_SET,index,&set);
    }
```

It is rare for application code to set more than the text of an instance of TEXTWIN used as a dialog box component. In such a case the hDlgSetText utility function may be used. As mentioned earlier, this utility function can also be used to replace the text of the title by passing an index of zero.

The text window's flags are generally either set on initialisation (usually from IN_TEXTWIN_XXX values set in a resource item) or are set or cleared by system code (see, for example, the DLGBOX dl_item_lock and dl_item_dim methods).

Sensing

Sensing a text window writes a pointer to the text and the text length to the buf and len elements of an SE_TEXTWIN struct. It does not provide any information about the text window's state flags. A typical call is as follows:

```
SE_TEXTWIN set;

p_send3(DatDialogPtr,O_WN_SENSE,index,&set);
```

A text window must not be sensed if it contains no text.

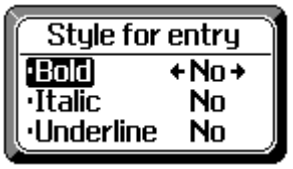
Choice lists

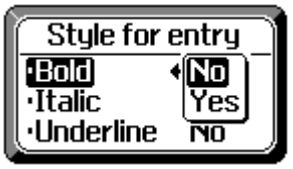
The *chlist.g* header file should be included when using a choice list control.

A choice list dialog control presents the user with a list of choice items only one of which is visible and hence selected. The selection can be changed by the following means:

- using the left and right arrow keys
- using first letter matching
- via a pop-out expanded view obtained with the tab key
- or optionally, via incremental matching with a sequence of key presses (the control must be specially configured to allow incremental matching by use of the appropriate flag - see below).

In the following two illustrations a dialog containing three choice lists is shown. In the right hand picture the user has pressed the Tab key to obtain the pop-out expanded view for the first, highlighted, choice list.





A choice list control, allowing the user to select one of four presidents of the United States, could be defined with the following two resources:

```
RESOURCE MENU presidents
    {
    items=
        {
        CHOICE_ITEM {str="Kennedy"};
        CHOICE_ITEM {str="Johnson"};
        CHOICE_ITEM {str="Nixon"};
        CHOICE_ITEM {str="Ford"};
        }
    }

RESOURCE CONTROL demonstration
    {
    class=C_CHLIST;
    flags=DLGBOX_ITEM_NOTIFY_CHANGED;
    prompt="President";
    info=CHLIST{rid=presidents;};
    }
```

In this case, the dialog control would initially display "Kennedy", surrounded by a pair of little arrows, to the right of the "President" prompt.

Initialisation

The `CHLIST` resource struct specifies the initial content and appearance of a choice list:

```
STRUCT CHLIST
    {
    LINK rid=0;
    BYTE nsel=0;
    BYTE flags=0;
    }
```

The `rid` element identifies the `MENU` resource that contains the list of selections as an array of `CHOICE_ITEM` structs:

```
RESOURCE MENU example_menu
    {
    items=
        {
        CHOICE_ITEM {str="zero";}
        CHOICE_ITEM {str="one";}
        CHOICE_ITEM {str="two";}
        };
    }
```

The CHOICE_ITEM structs are indexed according to the order in which they are listed in the MENU resource. Thus the first has index zero, the second has index one, and so on. In *hwim.rh* the CHOICE_ITEM struct is defined as follows:

```
STRUCT CHOICE_ITEM /* choice list item */
    BYTE {
    TEXT str=""; /* identification text */
    }
```

The nsel element of a CHLIST resource struct specifies the index number of the initially selected item.

The flags element may optionally contain the IN_CHLIST_INCREMENTAL flag, to allow choice list selection to be made using incremental key matching. Note that this option can not be set dynamically.

Setting

A choice list is set by passing a pointer to an SE_CHLIST struct to the wn_set method

```
typedef struct
   {
    UWORD set_flags;    /* which fields are significant */
    PR_VAROOT *data;    /* pointer to array containing data */
    UWORD nsel;         /* index of current item */
    } SE_CHLIST;
```

The property to be set is indicated by ORing one or more of the following flags into the flags field of the above struct.

SE_CHLIST_NSEL	the index of the current item is to be set.
SE_CHLIST_DATA	the data is to be replaced. The replacement data is a string array - see variable arrays in the *OLIB Reference* manual.
SE_CHLIST_RETAIN	data should not be destroyed on destruction of the choice list control - once set this flag cannot be cleared.

The content of a choice list can be set dynamically, say, from the dialog's dl_dyn_init method. However, changing the content of a choice list once the dialog has become visible is not recommended, since the width of the dialog box is set on initialisation. If the choice list content must be replaced, then care should be taken to ensure that the text does not become too wide for the dialog box to display.

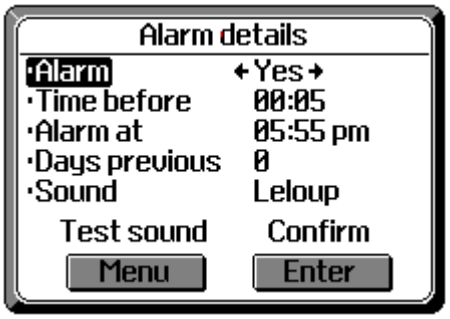
Sensing

A choice list is sensed by passing a pointer to an SE_CHLIST struct to the wn_sense method. The SE_CHLIST struct is defined above. Both nsel and data are sensed.

Push buttons and action lists

The *aclist.g* header file should be included when using an action list control.

An action list control presents the user with a horizontal list of one or more push-button options, as illustrated in the following diagram:



This action list consists of the two buttons, and their accompanying labels "Test sound" and "Confirm",at the bottom of the dialog (note that the action list must either be the last of a series of controls, or it must appear on its own).An action list, allowing the user to select either Yes or No, could be defined with the following resource:

```
RESOURCE ACLIST_ARRAY yes_or_no
    {
    button=
        {
        PUSH_BUT
            {
            keycode=-'n';
            str="No";
            },
        PUSH_BUT
            {
            keycode='y';
            str="Yes";
            }
        }
    };
```

The push-button is defined by a text string `str` that appears above the button, and a `keycode` indicating the key to be pressed by the user. For non-special keys, the uppercased key symbol will appear on the push-button. For special keys, the keycode is conveniently specified by a symbolic constant. The symbolic constants, and the text that will appear on the push-button, are as follows:

`W_KEY_RETURN`	"Enter"
`W_KEY_ESCAPE`	"Esc"
`W_KEY_DELETE_LEFT`	"Del"
`W_KEY_SPACE`	"Space"
`W_KEY_UP`	<WS_SYMBOL_UP_KEY>
`W_KEY_DOWN`	<WS_SYMBOL_DOWN_KEY>
`W_KEY_RIGHT`	<WS_SYMBOL_RIGHT_KEY>
`W_KEY_LEFT`	<WS_SYMBOL_LEFT_KEY>
`W_KEY_TAB`	"Tab"
`W_KEY_MENU`	"Menu"

For example, the leftmost button in the dialog, illustrated above, could be defined with the following `PUSH_BUT` resource.

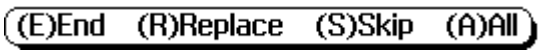
```
RESOURCE PUSH_BUT test_sound_button
    {
    keycode=W_KEY_MENU;
    str="Test sound";
    }
```

A negative `keycode` (as in an earlier example) indicates that the escape key can also be used to obtain the same effect (note that this is not possible if the escape key has already been assigned). The `yes_or_no` resource, defined earlier, could be used to create a dialog, asking the user to press 'y' or 'n', as follows:

```
RESOURCE DIALOG get_answer
    {
    title="Accept changes ?"
    flags=DLGBOX_NOTIFY_ESCAPE|DLGBOX_RBUF_FILLED;
    controls=
        {
        CONTROL
            {
            class=C_ACLIST;
            info=ACLIST
                {
                rid=yes_or_no;
                };
             }
        };
    }
```

The same effect could be obtained by using the SYS_AC_NO_YES resource defined in the system resource file.

There is an alternative form of the above action list, which is useful when space is at a premium, as illustrated in the following diagram.



As in this case, the compact, or small, action list is not usually accompanied by any other controls.

A small action list uses the same ACLIST_ARRAY resource as the standard action list. The example given above, yes_or_no, could be used to create a dialog consisting of a small action list, and no other controls, as follows:

```
RESOURCE DIALOG get_answer
    {
    flags=PR_WIN_FORCE_BOTTOM
    controls=
        {
        CONTROL
            {
            class=C_SMACLIST;
            info=ACLIST
                {
                rid=yes_or_no;
                };
            }
        };
    }
```

The only difference, between the dialog resources for the two action lists, is the class: the standard action list is an instance of the ACLIST class whereas the small action list is an instance of the SMACLIST class.

Initialisation

The appearance and behaviour of a push button are specified with a PUSH_BUT resource.

```
STRUCT PUSH_BUT
    BYTE {
    WORD keycode;
    TEXT str;       /* text associated with button */
    }
```

One or more buttons are in turn collected into an array: the first button in the array has index zero, the second has index one, and so on. The first button will appear on the far left of the control.

```
STRUCT ACLIST_ARRAY rid
    {
    LEN BYTE STRUCT button[];   /* array of push_buttons */
    }
```

This array is included as a dialog control by means of the ACLIST struct.

```
STRUCT ACLIST  /* Initialising struct for action list */
    {
    LINK rid;
    }
```

The resources are the same for both the standard and the small action lists.

There are no flags associated with this control, since there is nothing to change on initialisation, or via setting.

Setting and sensing

There is nothing that can usefully be set or sensed.

Edit boxes

The *edwin.g* header file should be included when using an edit box control.

An edit box control presents an editable string which may be wholly or partially visible. A wide range of options are available for customising this control: see the resources section. In the following example an edit box is displaying "rabbit burrow".

An edit box control accepting up to 40 characters, including tabs, with 20 visible at any one time could be created using the following resource:

```
RESOURCE CONTROL
    {
    class=C_EDWIN;
    prompt="string";
    info=EDWIN
        {
        str="";
        flags=IN_EDWIN_VULEN_CHARACTERS|IN_EDWIN_ACCEPT_TABS;
        maxlen=40;
        vulen=20;
        };
    }
```

Initialisation

The initial content and appearance of an edit box are specified by an `EDWIN` resource struct.

```
STRUCT EDWIN        /* edit box */
    {
    WORD vulen;     /* ignored unless either _VULEN_ flag set */
    WORD flags=0;
    WORD maxlen;
    TEXT str="";
    }
```

where `maxlen` specifies the default edit box width, and the maximum length of string that may be edited.

The behaviour of an edit box is specified by `OR`ing one or more of the following flags into the `flags` member of the `EDWIN` struct.

`IN_EDWIN_DIALLABLE`	indicates that the edit box should accept the telephone symbol via the shift+dial key combination.
`IN_EDWIN_ACCEPT_TABS`	indicates that the edit box should accept tabs.
`IN_EDWIN_AUTO_CUR_END`	indicates that the text, when first displayed, is not to be highlighted and that the cursor is to be placed at the rightmost position.
`IN_EDWIN_NO_AUTOSELECT`	indicates that the text, when first displayed, is not to be highlighted and that the cursor is to be placed at the leftmost position.

IN_EDWIN_VULEN_CHARACTERS	indicates that the width of the edit box is specified, in characters, in member vulen of the EDWIN struct.
IN_EDWIN_VULEN_PIXELS	indicates that the width of the edit box is specified, in pixels, in member vulen of the EDWIN struct.

Setting

The SE_EDWIN struct is used for setting an edit box.

```
struct
    {
    TEXT *buf;
    UWORD len;
} SE_EDWIN;
```

The following code fragment demonstrates the setting of an edit box, assumed to be the item with index 2:

```
SE_EDWIN set;

set.buf="Text";
set.len=4;
p_send4(DatDialogPtr,O_WN_SET,2,&set);
```

Sensing

The SE_EDWIN struct defined above is also used for sensing an edit box. The following code fragment demonstrates the sensing of an edit box:

```
SE_EDWIN sense;

p_send4(DatDialogPtr,O_WN_SENSE,2,&sense);
```

LONG numeric editor

The *ncedit.g* header file should be included when using a long numeric editor control.

A long numeric editor control presents an editable long integer value. In the following diagram, the long numeric editor control has a current value of five hundred thousand.

A numeric editor control could be created using the following resource:

```
RESOURCE CONTROL
    {
    class=C_LNCEDIT;
    prompt="LONG";
    info=LNCEDIT
        {
        low=1;
        high=700000;
        current=500000;
        };
    }
```

Initialisation

The initial content of a long numeric integer is specified by means of an LNCEDIT resource struct.

```
STRUCT LNCEDIT   /* long number edit box */
    {
    LONG current = 0;
    LONG low = 0;           /* lowest allowed value */
    LONG high = 10000000;   /* highest allowed value */
    }
```

where the current value may not be less than low or greater than high.

Setting

The SE_LNCEDIT struct is used for setting a long integer numeric editor.

```
typedef struct
    {
    LONG value;
    LONG low;
    LONG high;
    UWORD flags;
    } SE_LNCEDIT;
```

The property to be set is indicated by ORing one or more of the following flags into the flags field of the above struct

SE_LNCEDIT_VALUE	indicates that the current value is to be set
SE_LNCEDIT_LOW	indicates that the lower limit is to be set
SE_LNCEDIT_HIGH	indicates that the upper limit is to be set

The following code fragment illustrates the setting of a long integer numeric editor:

```
SE_LNCEDIT set;

set.flags=SE_LNCEDIT_VALUE|SE_LNCEDIT_HIGH;
set.value=100;
set.high=50000;
p_send4(DatDialogPtr,O_WN_SET,4,&set);
```

Sensing

A pointer to a LONG is used for sensing the current value. The low and high values can not be sensed. The following code fragment demonstrates the sensing of a long numeric editor:

```
LONG value;

p_send4(DatDialogPtr,O_WN_SENSE,4,&value);
```

Integer numeric editor

The *ncedit.g* header file should be included when using an integer numeric editor control.

An integer numeric editor control presents an editable unsigned integer value. In the following diagram the integer numeric editor has a current value of one.

An integer numeric editor control could be created using the following example resource:

```
RESOURCE CONTROL
    {
    class=C_NCEDIT;
    prompt="Number of cars"
    info=NCEDIT
        {
        low=0;
        high=6;
        current=4;
        };
    }
```

Initialisation

The initial content of an integer numeric editor are specified by means of an NCEDIT resource struct:

```
STRUCT NCEDIT   /* UWORD number edit box */
    {
    UWORD current = 0;
    UWORD low = 0;           /* lowest allowed value */
    UWORD high = 65535;   /* highest allowed value */
    }
```

where the `current` value may not be less than `low` or greater than `high`.

Setting

The `SE_NCEDIT` struct is used for setting an integer numeric editor.

```
typedef struct
    {
    UWORD value;
    UWORD low;
    UWORD high;
    UWORD flags;
    } SE_NCEDIT;
```

The property to be set is indicated by `OR`ing one or more of the following flags into the `flags` field of the above struct.

`SE_NCEDIT_VALUE` indicates that the current value is to be set.

`SE_NCEDIT_LOW` indicates that the lower limit is to be set.

`SE_NCEDIT_HIGH` indicates that the upper limit is to be set.

The following code fragment demonstrates the setting of an integer numeric editor:

```
SE_NCEDIT set;

set.flags=SE_NCEDIT_VALUE|SE_NCEDIT_HIGH;
set.high=100;
set.value=10;
p_send4(DatDialogPtr,O_WN_SET,4,&set);
```

Sensing

A pointer to a `UWORD` is used for sensing the current `value`. The `low` and `high` values can not be sensed. The following code fragment demonstrates the sensing of an integer numeric editor:

```
UWORD value;

p_send4(DatDialogPtr,O_WN_SENSE,4,&value);
```

WORD numeric editor

The *ncedit.g* header file should be included when using a word numeric editor control.

A word numeric editor control presents an editable signed integer value. In the following example the word numeric editor has a current value of minus one hundred.

A word numeric editor control could be created using the following resource:

```
RESOURCE CONTROL
    {
    class=C_WNCEDIT;
    prompt="Temperature"
    info=WNCEDIT
        {
        current=-100;
        };
    }
```

Initialisation

The initial content and appearance of a word numeric editor are specified by means of a `WNCEDIT` resource struct:

```
STRUCT WNCEDIT   /* numeric control edit box (signed words) */
    {
    WORD current = 0;
    WORD low = -32768;  /* lowest allowed value */
    WORD high = 32767;  /* highest allowed value */
    }
```

where the `current` value may not be less than `low` or greater than `high`.

Setting

The `SE_WNCEDIT` struct is used for setting the word numeric editor.

```
typedef struct
    {
    WORD value;
    WORD low;
    WORD high;
    WORD flags;
    } SE_WNCEDIT;
```

The property to be set is indicated by `OR`ing one or more of the following flags into the `flags` field of the above struct.

`SE_WNCEDIT_VALUE` indicates that the current value is to be set.

`SE_WNCEDIT_LOW` indicates that the lower limit is to be set.

`SE_WNCEDIT_HIGH` indicates that the upper limit is to be set.

The following code fragment demonstrates the setting of a word numeric integer:

```
SE_WNCEDIT set;

set.flags=SE_WNCEDIT_VALUE|SE_WNCEDIT_LOW;
set.low=4;
set.value=10;
p_send4(DatDialogPtr,O_WN_SET,3,&set);
```

Sensing

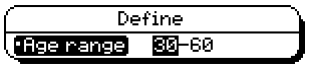
A pointer to a `WORD` is used for sensing the current `value`. The `low` and `high` values can not be sensed. The following code fragment demonstrates the sensing of a word integer numeric editor:

```
WORD value;

p_send4(DatDialogPtr,O_WN_SENSE,3,&value);
```

Range numeric editor

The *rgedit.g* header file should be included when using a range numeric editor control.

A range numeric editor control presents two editable unsigned words specifying upper and lower values of a range. In the following example a range numeric editor is shown with a lower value of thirty and an upper value of sixty.



A range numeric editor control could be defined using the following resource:

```
RESOURCE CONTROL
    {
    class=C_RGEDIT;
    prompt="Age range";
    info=RGEDIT
        {
        value_1=100;
        value_2=200;
        };
    }
```

Initialisation

The initial content of a range numeric editor is specified by means of an `RGEDIT` resource struct:

```
STRUCT RGEDIT    /* range editor */
    {
    WORD low=1;         /* lowest allowed value */
    WORD value_1=1;           /* lower value of range */
    WORD value_2=9999;  /* higher value of range */
    WORD high=9999;           /* highest allowed value */
    }
```

where `value_1` must be less than or equal to `value_2` and both values may not be less than `low` or greater than `high`.

Setting

The `SE_RGEDIT` struct is used for setting the range numeric editor.

```
typedef struct
    {
    UWORD value[4];
    UWORD flags;
    } SE_RGEDIT;
```

The `value` array is indexed as follows:

`IX_RGEDIT_LOW`	index of lower limit in `value` array.
`IX_RGEDIT_VALUE_1`	index of lower current value in `value` array.
`IX_RGEDIT_VALUE_2`	index of upper current value in `value` array.
`IX_RGEDIT_HIGH`	index of upper limit in `value` array.

The property to be set is indicated by `OR`ing one or more of the following flags into the flags field of the above struct.

`SE_RGEDIT_LOW`	indicates that the lower limit is to be set.
`SE_RGEDIT_VALUE_1`	indicates that the current lower value is to be set.
`SE_RGEDIT_VALUE_2`	indicates that the current upper value is to be set.
`SE_RGEDIT_HIGH`	indicates that the upper limit is to be set.

The following code fragment demonstrates the setting of a range numeric integer:

```
SE_RGEDIT set;

set.flags=SE_RGEDIT_VALUE_1|SE_RGEDIT_VALUE_2;
set.value[IX_RGEDIT_VALUE_1]=4;
set.value[IX_RGEDIT_VALUE_2]=10;
p_send4(DatDialogPtr,O_WN_SET,2,&set);
```

Sensing

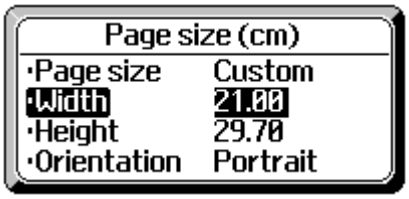
The `SE_RGEDIT` struct (see above) is used for sensing the range numeric editor: all four members of the struct are sensed. The following code fragment demonstrates the sensing of a range numeric editor:

```
SE_RGEDIT sense;

p_send4(DatDialogPtr,O_WN_SENSE,2,&sense);
```

Floating point editor

The *fltedit.g* header file should be included when using a floating point editor control.

A floating point editor control presents an editable floating point number. In the following example two floating point editors are shown with current values of 21.00 and 29.70.



A floating point editor control could be defined using the following resource:

```
RESOURCE CONTROL
    {
    class=C_FLTEDIT;
    prompt="Width";
    info=FLEDIT
        {
        current=21.00;
        low=0.0;
        high=40.0;
        ndec=2;
        };
    }
```

Initialisation

The initial content and appearance of a floating point editor are specified by means of an `FLTEDIT` resource struct:

```
STRUCT FLTEDIT   /* floating point edit box */
    {
    DOUBLE current=0.0;
    DOUBLE low =-9.9999999999e99;  /* lower bound */
    DOUBLE high =9.9999999999e99;  /* upper bound */
    BYTE vulen=5;       /* width of editor in characters */
    BYTE ndec=0;        /* P_DTOB_GENERAL */
    }
```

where the `current` value may not be less than `low` or greater than `high`. The `current` value is displayed to `ndec` decimal places in a box of width `vulen` characters. If `ndec` is zero, the `current` value is displayed in general format as defined for the PLIB routine `p_dtob` (see the *Plib Reference* manual).

Setting

The `SE_FLEDIT` struct is used for setting the floating point editor.

```
typedef struct
    {
    DOUBLE current;   /* current value */
    DOUBLE low;       /* lower bound */
    DOUBLE high;      /* upper bound */
    WORD set_flags;   /* which fields to set */
    } SE_FLTEDIT;
```

The property to be set is indicated by `OR`ing one or more of the following flags into the `set_flags` field of the above struct.

`SE_FLTEDIT_CURRENT` indicates that the current value is to be set.

`SE_FLTEDIT_LOW` indicates that the lower limit is to be set.

`SE_FLTEDIT_HIGH` indicates that the upper limit is to be set.

The following code fragment demonstrates the setting of a floating point numeric integer:

```
SE_FLTEDIT set;

set.set_flags=SE_FLTEDIT_VALUE|SE_FLTEDIT_LOW;
set.low=4;
set.value=10.3;
p_send4(DatDialogPtr,O_WN_SET,2,&set);
```

Sensing

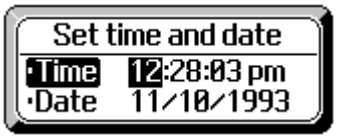
A pointer to a `DOUBLE` is used for sensing the `current` value. The `low` and `high` values can not be sensed. The following code fragment demonstrates the sensing of a range integer numeric editor:

```
DOUBLE current;

p_send4(DatDialogPtr,O_WN_SENSE,2,&current);
```

Date/time editor

The *dtedit.g* header file should be included when using a date/time editor control.

The date/time editor presents either an editable date, an editable time or an editable duration. In the following example the upper date/time editor is showing the time in am/pm format in hours, minutes and seconds, the lower editor is showing the date.



A date/time editor showing a duration of one hour and ten minutes could be defined using the following resource:

```
RESOURCE CONTROL
    {
    class=C_DTEDIT;
    prompt="Time left";
    info=DTEDIT
        {
        flags=IN_DTEDIT_HHMM_D
        current=4200;
        };
    }
```

Initialisation

The initial content of a date/time editor is specified by means of a `DTEDIT` resource struct:

```
STRUCT DTEDIT    /* combined date/time editor */
    {
    WORD flags;
    LONG current;
    LONG low;       /* lowest allowed value */
    LONG high;      /* highest allowed value */
    }
```

where the `current` value may not be less than `low` or greater than `high`.

The content of the date/time editor is specified by assigning **one** of the following flags into the flags member.

`IN_DTEDIT_DDMMYYYY`	initialise as a date editor to display a date in day, month and year format: the `current` date is specified as days elapsed since 1/1/1900.
`IN_DTEDIT_HHMMSS`	initialise as a time editor to display a time as hours, minutes and seconds. The `current` time is specified in seconds elapsed since midnight.
`IN_DTEDIT_HHMM`	initialise as a time editor to display a time as hours and minutes. The `current` time is specified as seconds elapsed since midnight.

IN_DTEDIT_HHMMSS_D	initialise as a time editor to display a duration as hours, minutes and seconds. The current duration is specified in seconds.
IN_DTEDIT_HHMM_D	initialise as a time editor to display a duration as hours and minutes. The current duration is specified in seconds.
IN_DTEDIT_HHMMSS_ND	initialise as a time editor to display a negative duration as hours, minutes and seconds. The current duration is specified in seconds.
IN_DTEDIT_HHMM_ND	initialise as a time editor to display a negative duration as hours and minutes. The current duration is specified in seconds.

Setting

The SE_DTEDIT struct is used for setting the date/time editor.

```
typedef struct
    {
    UWORD flags;
    LONG value;
    LONG low;
    LONG high;
    } SE_DTEDIT;
```

The property to be set is indicated by ORing one or more of the following flags into the flags field of the above struct.

SE_DTEDIT_VALUE	indicates that the current value is to be set.
SE_DTEDIT_LOW	indicates that the lower limit is to be set.
SE_DTEDIT_HIGH	indicates that the upper limit is to be set.

The following code fragment demonstrates the setting of a date/time editor:

```
SE_DTEDIT set;

set.flags=SE_DTEDIT_VALUE;
set.value=p_date();
p_send4(DatDialogPtr,O_WN_SET,2,&set);
```

Sensing

A pointer to a SE_DTEDIT is used for sensing the current value. The low and high values can be sensed. The following code fragment demonstrates the sensing of a date/time editor:

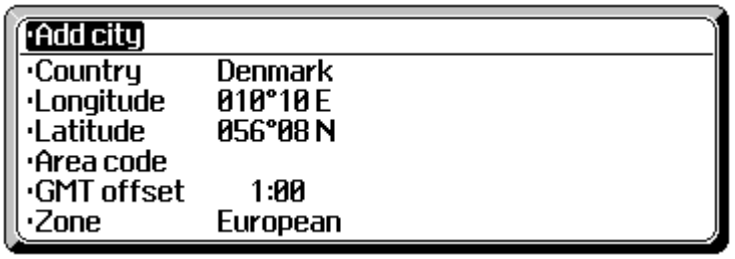
```
SE_DTEDIT sense;

p_send4(DatDialogPtr,O_WN_SENSE,2,&sense);
```

The Latitude/Longitude editor

The *lledit.g* header file should be included when using a latitude/longitude editor control.

A latitude/logitude editor presents either an editable latitude or an editable longitude. Both the latitude and the longitude are expressed in degrees and minutes followed by a single character specifying the cardinal point i.e. N, W, S or E.

In the following example the upper latitude/longitude editor is showing a longitude and the lower editor is showing a latitude.



A latitude editor control that initially displayed 56° 8 N could be defined using the following resource:

```
RESOURCE CONTROL
    {
    class=C_LLEDIT;
    flags=IN_LLEDIT_LATITUDE;
    prompt="Latitude";
    info=LLEDIT
        {
        value=3368;
        };
    }
```

Initialisation

The initial content and appearance of a latitude/longitude editor are specified by means of an `LLEDIT` resource struct:

```
STRUCT LLEDIT    /* lat long editor */
    {
    WORD flags;         /* Latitude or longitude */
    WORD value=0;              /* The default value */
    }
```

where the latitude/longitude `value` is in minutes of arc: a positive sign indicating a latitude/longitude in the northern /western hemisphere, and a negative sign indicating a latitude/longitude in the southern /eastern hemisphere. Thus -604 corresponds to a latitude of 10° 4 S or a longitude of 10° 4 E.

The behaviour of a latitude/longitude editor is specified by assigning one of the following flags to the flags member.

`IN_LLEDIT_LATITUDE` display `value` as a latitude.

`IN_LLEDIT_LONGITUDE` display `value` as a longitude.

Setting

The `SE_LLEDIT` struct is used for setting the latitude/longitude editor.

```
typedef struct
    {
    WORD value;
    } SE_LLEDIT;
```

The following code fragment demonstrates the setting of a date/time editor:

```
SE_LLEDIT set;

set.value=3368;
p_send4(DatDialogPtr,O_WN_SET,2,&set);
```

Sensing

A pointer to an `SE_LLEDIT` struct is used for sensing the current `value`. The following code fragment demonstrates the sensing of a date/time editor:

```
SE_LLEDIT sense;

p_send4(DatDialogPtr,O_WN_SENSE,2,&sense);
```

File name editor

The *files.g* header file should be included when using a file name editor control.

A file name editor present the user with a file name that may be edited with the keyboard or with the pop-out file selector (via the Tab key). A wide range of options is provided for customising the behaviour of file name editors: these options are described in the resources section below.

A file name editor is always supplied with a pack selector, that is placed immediately below, as in the following example: the selector is specified by `OR`ing the `DLGBOX_NEEDS_PACK` flag into the `flags` member of the `CONTROL` struct (see below).

A file name editor is commonly used when saving an edited file, for example a document created with the Word application. It is not usually used for opening an already existing file for which a file name choice list is better suited.

A file name editor control could be defined using the following resource:

```
RESOURCE CONTROL
    {
    class=C_FNEDIT;
    flags=DLGBOX_ITEM_NEEDS_PACK;
    prompt="";
    info=FNEDIT
        {
        flags=IN_FNEDIT_STANDARD;
        fname="easel.pic";
        };
    }
```

Initialisation

The initial content and appearance of a file name editor are specified by means of an `FNEDIT` resource struct:

```
STRUCT FNEDIT   /* filename editor */
    {
    BYTE flags=0;
    TEXT fname="";
    }
```

The behaviour of the file name editor is specified by `OR`ing a suitable combination of the following flags into the `flags` member of the above struct.

`IN_FNEDIT_STANDARD`	set `fname` to be the file name specified by the `DatUsedPathNamePtr` reserved static: typically points to the application's current file.
`IN_FNEDIT_ALLOW_DIRS`	allow the name of a directory with the file name.
`IN_FNEDIT_JUST_DIRS`	allow only the name of a directory and no file name.
`IN_FNEDIT_FORCE_NXIST`	disallow existing files.
`IN_FNEDIT_NO_AUTOQUERY`	do not query on an existing file (by default the control prompts the user before accepting an already existing file - this helps prevent accidental deletion/overwriting).
`IN_FNEDIT_ACCEPT_NULL`	allow the null string.
`IN_FNEDIT_SET_DEFEXT`	set the default file extension to that of the file specified in the `FNEDIT` resource struct.
`IN_FNEDIT_CAN_WILDCARD`	allow wildcards.

Setting

The file name may be set by passing a pointer to a character string to the `wn_set` method. The character string should contain the file name terminated by a zero character.

The default extension may be set by passing a pointer to a character string to the `wn_set` method. The first character must be `0x01`. The following characters can either be a file name, or a file extension preceded by a full stop character. Only the extension is significant.

The following code fragment demonstrates the setting of the default extension:

```
TEXT buf[P_FNAMESIZE];

buf[0]=1;
p_scpy(&buf[1],".DBF");
p_send4(DatDialogPtr,O_WN_SET,3,&buf[0]);
```

Sensing

A buffer, of size at least `P_FNAMESIZE` bytes, is used for sensing the current full file specification from a file name editor, as illustrated by the follwing code fragment:

```
TEXT buf[P_FNAMESIZE];

p_send4(DatDialogPtr,O_WN_SENSE,3,&buf[0]);
```

File name choice list

The *files.g* header file should be included when using a file name choice list control.

A file name choice list presents the user with a choice of files with the selection being made with the keyboard arrow keys, via letter matching or by pressing Tab to display the pop-out file selector. A wide range of options is provided for customising the behaviour of a file name choice list: these options are described in the resources section below.

A file name choice list is always supplied with a pack selector that is placed immediately below as in the following example: the selector is specified by `OR`ing the `DLGBOX_NEEDS_PACK` flag into the `flags` member of the `CONTROL` struct (see below).

A file name choice list is commonly used when opening an existing file. It is not usually used for saving an edited/modified file for which a file name editor is better suited.

A file name choice list control could be defined using the following resource:

```
RESOURCE CONTROL
    {
    class=C_FNSELWN;
    flags=DLGBOX_ITEM_NEEDS_PACK;
    info=FNSELWN
        {
        fname="";
        };
    }
```

Initialisation

The initial content and behaviour of a file name choice list control are defined by an `FNSELWN` resource struct as follows:

```
STRUCT FNSELWN    /* filename selector */
    {
    BYTE flags=0;
    TEXT fname="";
    }
```

The behaviour of a file name choice list control is specified by `OR`ing one or more of the following flags into the `flags` member of the above struct

`IN_FNSELWN_STANDARD`	select the file specified by the `DatUsedPathNamePtr` reserved static: typically points to the application's current file.
`IN_FNSELWN_SHOW_DIRS`	show directory names.
`IN_FNSELWN_HIDE_FILES`	hide directory names.
`IN_FNSELWN_RESTRICT_LIST`	display only files with extension matching the default.
`IN_FNSELWN_CAN_TAG`	allow file tagging: file tagging is carried out with the file name choice list - pressing the '+' and '-' keys tags and untags a file respectively.

IN_FNSELWN_ACCEPT_NULL	accept the null string.
IN_FNSELWN_SET_DEFEXT	set the default extension to that of the file specified in the FNSELWN struct.
IN_FNSELWN_CAN_WILDCARD	allow wildcards.

Setting

The file name may be set by passing a pointer to a character string to the wn_set method. The character string should contain the file name terminated by a zero character. Wildcards in the file name are allowed.

The default extension may be set by passing a pointer to a character string to the wn_set method. The first character must be 0x01. The following characters can either be a file name, or a file extension preceded by a full stop character. Only the extension is significant.

The following code fragment demonstrates the setting of the file name:

```
TEXT buf[P_FNAMESIZE];

p_scpy(&buf[0],"DATABASE.DBF");
p_send4(DatDialogPtr,O_WN_SET,3,&buf[0]);
```

Sensing

A buffer, of size at least P_FNAMESIZE bytes, is used for sensing the full file specification of the currently selected file.

The following code fragment demonstrates the sensing a file name choice list:

```
TEXT buf[P_FNAMESIZE];

p_send4(DatDialogPtr,O_WN_SENSE,3,&buf[0]);
```

CHAPTER 9

ACTIVE OBJECTS

An active object represents an event source and is, by definition, an instance of any class that has `ACTIVE` as an ancestor in its inheritance chain. This chapter provides a simple introduction to the use of active objects: for further information on the precise mechanisms involved, see *The APPMAN Application Manager Class*, *The Active Class and Active Objects*, and following chapters in the *OLIB Reference* manual.

All HWIM applications contain at least one active object, which is a subclass of the HWIM `WSERV` window server active object class. This active object represents the source of events directed to the application by the window server process.

An application may create additional active objects to represent other event sources. A simple example would be to implement a timer, so that the application will, from time to time, receive timer expiry events.

A prioritised queue of an application's active objects is maintained by the application manager. A significant part of the application manager's function is in its event loop, which manages this queue, associating the occurrence of an event with the appropriate active object and sending it a message.

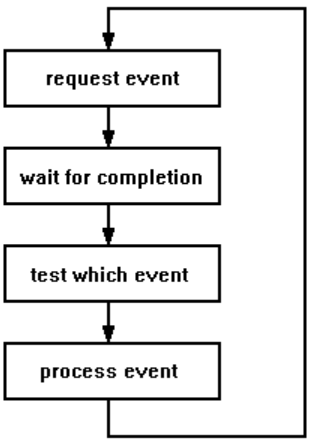
An active object must be added to this queue when it is created. This may be done explicitly by the code that creates the active object, or it may be included in the initialisation (that is, in the `ao_init` method) of the active object itself.

Active objects and asynchronous requests

Before an event can occur it must be requested by a program. The way to do this is to make an asynchronous request, as explained in the *Asynchronous Requests and Semaphores* chapter of the *PLIB Reference* manual. In consequence there is a strong connection between the making of asynchronous requests and active objects.

An active object is, in fact, the standard way of making and processing asynchronous requests in HWIM applications. Since the application manager contains a mechanism with the express purpose of scheduling the events marking the completion of asynchronous requests that are encapsulated in active objects, you are recommended to use active objects for all asynchronous operations in an HWIM application.

The diagram opposite illustrates the general form of any event loop. Testing for the event that has completed is usually done by polling the status words of all the possible requests.



The application manager's event loop follows this general pattern, except that the code that requests an event and the code that processes its completion are provided by the `ao_queue` and `ao_run` methods respectively of one or more active objects. On completion of a request, the application manager polls the active objects in its queue to determine which one can process the event, and explicitly sends it an `AO_RUN` message. In contrast, the application manager's event loop contains no explicit code to request an event and relies on its active objects to do this. Since an HWIM application always has at least one active object - its window server active object - there will always be something to make such a request.

Note that there is an implied restriction that there should be only a single

asynchronous request in an ao_queue method, and there should be no such request made in the ao_run method.

All ao_run methods must return the value RUN_ACTIVE_USED (defined in *appman.g*) to signify that the event has been consumed.

An active object has a status word (active.status) in its property and it is this status word that the application manager polls to determine which active object is to be sent an AO_RUN message. In order to assist the polling mechanism, an active object has a further item of property, active.isactive, which must be set to a TRUE value when an asynchronous request is made. It is automatically cleared when the active object is sent an AO_RUN message.

Active object priorities

When an active object is created, it must be given a priority (by setting its active.priority property) before it is added to the application manager's queue. If, at any time, the requests from two or more active objects have completed, the one with highest priority will be the first to be sent an AO_RUN message.

A priority is a signed byte and therefore must lie in the range +127 (highest priority) to -128 (lowest priority). A number of standard priorities are defined in *appman.g* and some of the more significant of these are explained below.

PRIORITY_ACTIVE_IPCS	+80 - for inter-process communication
PRIORITY_ACTIVE_WSERV	+60 - the priority of the window server active object
PRIORITY_ACTIVE_SERIAL	+20 - for serial port communications
PRIORITY_ACTIVE_FILES	-20 - for reading or writing files
PRIORITY_ACTIVE_REPEATER	-40 - for timers, animation, etc.
PRIORITY_ACTIVE_PRINT	-60 - for communication with a printer
PRIORITY_ACTIVE_COMPUTE	-100 - for background computation

These values are supplied as guidelines and you are not compelled to follow them precisely. However, in order to ensure the application's responsiveness to redraws and the keyboard, most active objects should be given priorities that do not exceed that of the window server active object.

Application responsiveness

An active object can not be given a chance to run (by being sent an AO_RUN message) until the execution of a previous AO_RUN message (by this, or any other active object) has terminated. Thus an active object, even one of low priority, that performs extensive processing in its ao_run method will reduce the application's responsiveness to other events.

It is the programmer's responsibility to ensure that the processing done in any one call to an active object's ao_run method is restricted to a reasonable amount. An active object that, for example, is being used to write a file to an SSD should not write the whole file in one operation, but should divide the writing into a number of relatively small sections.

One way of doing this is to construct a buffer containing a section of the file and write the contents of this buffer to the file by means of an asynchronous write request in the ao_queue method. On receipt of an AO_RUN message, signifying that the write has completed, the process can be repeated, provided there is still part of the file that has not been written.

An alternative approach would be to use a technique similar to the background processing mechanism, described below. In this case writing to the file would be performed synchronously, from within the ao_run method. A disadvantage to this second technique is that, if writing to a remote device, the write could take an extended time to complete and thus may compromise the responsiveness of the application.

Background processing

An active object is ideally suited to breaking down a long computation into a sequence of small sections and may be used for this purpose, even if the process does not involve making asynchronous requests. Examples of where this technique may be useful are the formatting of a large amount of text, or the recalculation of all the cells of a spreadsheet.

The AIDLE class is supplied in the OLIB library as a basis for this type of use. The supplied class subclasses ACTIVE to replace the ao_init method with code that sets a priority of PRIORITY_ACTIVE_COMPUTE and adds itself to the application manager's active object queue. It uses the default (ACTIVE class) ao_queue method, which simply sets active.isactive to TRUE and generates an event by calling p_iosignal.

The AIDLE class must be subclassed to replace the ao_run method with one to perform a unit of processing and, if processing is not complete, send itself an AO_QUEUE message.

Errors

Apart from errors during initialisation (for, example, failure to open a channel to a device) errors that result in p_leave being called are expected only to occur in the ao_run method. Thus, all operations that could potentially fail, such as the allocation of memory from the heap, should be performed from within this method, with a call to p_leave if an error occurs. Where possible, it is more effective to use the f_xxx functions, such as f_alloc, f_new or f_send.

If failure is possible in any asynchronous request that is made in the ao_queue method, the value of active.status should be checked for an error value in the ao_run method. Any such error should result in p_leave being called, passing the error value.

Any call to p_leave from within the ao_run method is handled by system code to perform standard error handling, as described in the *Error Handling and Error Recovery* chapter. Part of this standard mechanism is to send the active object an AO_ABRUN message.

The ao_abrun method supplied by the ACTIVE class provides fail-safe reporting of the error and so, by default, no active object needs to take any explicit action to report errors to the user. The ao_abrun method is intended to be replaced by subclassers to provide any application-specific error recovery (again, see the *Error Handling and Error Recovery* chapter). In most cases the replacement method will, in addition to any other action, supersend the AO_ABRUN message to report the error. Depending on the specific circumstances, the replacement ao_abrun method may, as its final action, send the active object a DESTROY message. This would normally be appropriate if the active object was created by a command manager method, called as a result of the user's selection of a command menu option.

A simple timer

The Timer demonstration application is installed into the *\sibosdk\oopdemo* directory. It may be built from that directory by typing:

```
make timer
```

It is a simple example that uses a timer to print an information message every two seconds, but clearly illustrates the way in which an active object is set up and used.

The category file, *timer.cat* contains, in addition to classes that will be familiar from the "Hello World" example, the class definition of a timer:

```
CLASS mytimer active
    {
    REPLACE ao_init
    REPLACE ao_queue
    REPLACE ao_run
    PROPERTY
        {
        UWORD count;
        }
    }
```

Note that, in order to show clearly the active object mechanisms, the MYTIMER class subclasses ACTIVE. In a real case it would be more efficient to subclass the OLIB TIMER class, which supplies some of the functionality that is duplicated in MYTIMER.

During the initialisation of the application's window server active object, an instance of MYTIMER is created and initialised as illustrated below:

```
METHOD VOID timerws_ws_dyn_init(PR_TIMERWS *self)
    {
    self->wserv.cli=f_new(CAT_TIMER_TIMER,C_TIMERBW);
    p_send2(self->wserv.cli,O_WN_INIT);
    f_newsend(CAT_TIMER_TIMER,C_MYTIMER,O_AO_INIT,"TIM:",-1);
    }
```

The `ao_init` method first supersends the `AO_INIT` message to be processed by the `ACTIVE` `ao_init` method, which opens a channel to the `TIM:` device. It then sets itself to a reasonably low priority and adds itself to the application manager's active object queue. Its final action is to send an `AO_QUEUE` message.

```
METHOD VOID mytimer_ao_init(PR_MYTIMER *self,TEXT *devname,INT mode)
    {
    p_supersend4(self,O_AO_INIT,devname,mode);
    self->active.priority=PRIORITY_ACTIVE_REPEATER;
    p_send3(w_am,O_AM_ADD_TASK,self);
    p_send2(self,O_AO_QUEUE);
    }
```

The `ao_queue` method, listed below, simply makes an asynchronous request on the timer, using `active.stat` as its status word. Also, as required, it sets `active.isactive` to `TRUE`. An event will be signalled after the requested two-second delay.

```
METHOD VOID mytimer_ao_queue(PR_MYTIMER *self)
    {
    LONG delay;

    delay=20;
    p_ioc4(self->active.pcb,P_FREAD,&self->active.stat,&delay);
    self->active.isactive=TRUE;
    }
```

When the event has occurred `MYTIMER` will be sent an `AO_RUN` message to signify that the requested event has completed. The `ao_run` method increments a counter in its property and uses this value to display an information message at the bottom right hand corner of the screen. It then sends an `AO_QUEUE` message to restart the timer before returning `RUN_ACTIVE_USED`.

```
METHOD INT mytimer_ao_run(PR_MYTIMER *self)
    {
    self->mytimer.count+=1;
    hInfoPrint(TIMER_INFO,self->mytimer.count);
    p_send2(self,O_AO_QUEUE);
    return(RUN_ACTIVE_USED);
    }
```

The timer continues to run for the lifetime of the application.

In a real application an active object could be created elsewhere in the program - frequently from a command manager method, executed in response to the selection of a menu option. The `ao_run` method would include a test for the completion of processing and, when complete would terminate itself by not sending an `AO_QUEUE` message (it might send itself a `DESTROY` message instead).

CHAPTER 10

ERROR HANDLING AND ERROR RECOVERY

Psion's Object Oriented programming system and the associated libraries provide considerable support for reporting and recovering from error conditions. In consequence, a typical HWIM application contains very little explicit error-handling code.

The cornerstones of error reporting and recovery are:

- the use of `p_enter` and `p_leave` to centralise the handling of errors
- using `PROPERTY n` statements in category files, for the automatic destruction of component objects
- the OLIB `CLEANUP` class, to provide for the freeing of temporary resources
- standard error reporting in the `ao_abrun` method of all active objects

This chapter gives a brief summary of the range of techniques that are available.

Errors during initialisation

One simple precaution makes the handling errors during the initialisation of an application very simple:

> *The application should be written so that all resources that the application needs in order to draw its initial view are created from within the `ws_dyn_init` method of its subclass of `WSERV`.*

If this condition is met then, assuming there are no coding errors that cause `p_panic` to be called, the application can not fail between the return from the `ws_dyn_init` method and its appearance on the screen.

All that the application has to do in the event of a failure to create one of its start-up resources from anywhere in the `ws_dyn_init` method, or any other functions or methods that this calls, is to call `p_leave` with a suitable error number. System code handles the recovery and the reporting that the application has failed to start.

The most common cause of failure during initialisation is that there is insufficient memory available. In many applications this is the only error that could occur during start-up initialisation. An error of this nature can normally be precipitated by setting the application's initial heap size requirement to be sufficiently large. This is done by setting the `heapsize` variable in the applications *.pr* file, as described in the *An HWIM Application - Hello World* chapter. Should insufficient memory be available, the application will then fail at a very early stage, before any application-specific code is executed, and this failure will be handled by system code.

To avoid excessive memory use by an application you should avoid allowing for the worst case. You should not, for example, set an initial heap size for a file-based application so that it is guaranteed to be able to load an exceptionally large file.

This technique may not prevent application code from running if, for example, the out-of-memory failure occurs when the window server is creating server-side resources for the application, or if a file-based application fails while opening a large file. It will, however, cause the application to fail earlier rather than later in the majority of cases.

General error recovery

One of the most important aspects of the handling of errors is to understand that virtually all application-specific code is executed from within the ao_run method of some active object or other. For example, all code that is executed in response to the receipt of a window server event is executed from within the ao_run method of the application's subclass of the WSERV object. This includes all system-generated redrawing and all keypress processing, which itself includes both the receipt of a WN_KEY message by any window and, more indirectly, the processing of a command by means of a command manager method.

There are two main areas of code that are exceptions to this general case:

- the start-up initialisation of an application
- code executed in response to an error, such as an active object's ao_abrun method.

Errors occurring in the first of these two areas are handled as described earlier, while the code executed in response to an error should be written so that it can never, itself, generate errors.

In consequence an error can always be reported simply by calling p_leave. When called within an active object's ao_run method, this will be caught by the application manager's event handler and will, cause the application manager to be sent an AM_CLEAN_UP message. This provides standard resource clean-up and also makes a call-back to the active object's own ao_abrun method, to provide standard error reporting (which is, itself, designed so that it will not fail). For further details, see the description of APPMAN's am_start method, and the further topics that it references, in the *APPMAN Application Manager Clas*s chapter of the *OLIB Reference* manual.

Note that an application that has one or more global actions to execute on the occurrence of all types of error can subclass the am_clean_up method to provide these actions as well as the method's standard functionality. The Record application, which is described in the *Application Design* chapter, and whose source code is supplied with the SDK, provides an example of the use of this technique.

The roll-back principle

All operations that can fail should be written so that failure causes roll-back to a previously safe state. Failure to ensure adequate roll-back is one of the most common defects in application software and usually evidenced by a monotonically increasing use of memory as some operation is repeated. Such an occurrence can generally be detected faily easily, for example, by the use of *spy.app* (described in the *Series 3/3a Programming Guide*).

A simple example is the insertion of records into a variable array (see, for example, the va_insertm method of the VAFLAT class, described in the *Variable Arrays* chapter of the *OLIB Reference* manual). If an error occurs before the insertion is complete, any partially inserted data is removed before the error is propagated by calling p_leave.

In general, any operation that consists of a sequence of stages, where any stage could fail, must be written to release any resources that have been created in earlier successful stages. The general model is illustrated in the following code:

```
VOID multi_stage()
    {
    INT error;

    stage_one(); /* this creates a resource, calling p_leave on failure */
    error=p_enter(stage_two); /* catch any error in the second stage */
    if (error)
        {
        undo_stage_one(); /* release the resource created in stage one */
        p_leave(error);   /* propagate the error to system code */
        }
    }
```

If possible, such code should be written so that the roll-back is as simple as possible. For example, suppose a large amount of data has to be inserted into a buffer, and that the data has to be read in segments. Recovery may be complex if each segment is read and inserted separately, since a failure will require the deletion of all previously inserted segments. A simpler approach, which requires no explicit error recovery code, is to pre-allocate space for the entire insertion in a single operation. If this fails (presumably by calling p_leave) no further action is necessary. If it succeeds, the data can be written into the allocated space, segment by segment, with no risk of subsequent failure. If there could be a failure in reading the data

segments then this approach still simplifies matters since there is always only a fixed size of allocated memory to be released, regardless of how many segments have been copied into it.

The following sections give a number of different means of ensuring that roll-back recovers resources that have been created before an error occurs. In a real case it is likely that a mixture of these techniques will be used.

Roll-back for component objects

The creation of an object that contains a number of components could fail during the creation of the object itself or during the creation or initialisation of any of its components. Any failure should result in the destruction of the object and any partially created components. This situation is particularly simple since it can make use of the built-in mechanisms for component destruction.

Suppose, for example, that a MYAPP category contains the `MYCLASS` class, with component classes `COMPONENT1` and `COMPONENT2`. The class definition for `MYCLASS` could be as follows:

```
CLASS myclass root
    {
    ADD init
    PROPERTY 2
        {
        VOID *comp1;
        VOID *comp2;
        }
    }
```

where its `init` method function might be:

```
VOID myclass_init(PR_MYCLASS *self)
    {
    self->myclass.comp1=f_new(CAT_MYAPP_MYAPP,C_COMPONENT1);
    self->myclass.comp2=f_new(CAT_MYAPP_MYAPP,C_COMPONENT2);
    }
```

If a `MYCLASS` instance is created with:

```
VOID *hand;

hand=f_newsend(CAT_MYAPP_MYAPP,C_MYCLASS,O_INIT);
```

then a failure (by means of a call to `p_leave`) at any stage will cause the object and any created components to be sent a `DESTROY` message, and the call to `p_leave` is then propagated.

Note the use of `f_new` and `f_newsend`, to guarantee a call to `p_leave` on failure.

The principle may be applied to the creation of components of the components, and so on.

Other resources in an object's property

Resources that are not component objects, but whose handles are stored in an object's property must be explicitly released in the object's `destroy` method. This is illustrated in the following example, which extends the one given above.

In this case, `MYCLASS` has a cell of allocated memory, with its handle stored in its property, according to the class definition:

```
CLASS myclass root
    {
    REPLACE destroy
    ADD init
    CONSTANTS
        {
        ALLOC_SIZE 100
        }
    PROPERTY 2
        {
        VOID *comp1;
        VOID *comp2;
        BYTE *alloc;
        }
    }
```

Its `init` method function could then be:

```
VOID myclass_init(PR_MYCLASS *self)
    {
    self->myclass.comp1=f_new(CAT_MYAPP_MYAPP,C_COMPONENT1);
    self->myclass.comp2=f_new(CAT_MYAPP_MYAPP,C_COMPONENT2);
    self->myclass.alloc=f_alloc(ALLOC_SIZE);
    }
```

Again, note the use of the `f_xxx` functions, to guarantee a call to `p_leave` on failure.

The `destroy` method function would be:

```
VOID myclass_destroy(PR_MYCLASS *self)
    {
    if (self->myclass.alloc)
        {
        p_free(self->myclass.alloc)
        self->myclass.alloc=NULL; /* not strictly necessary in this case */
        }
    p_supersend2(self,O_DESTROY);
    }
```

Again, creating an instance of `MYCLASS` with:

```
VOID *hand;

hand=f_newsend(CAT_MYAPP_MYAPP,C_MYCLASS,O_INIT);
```

will result in total roll-back (and an error report) in the event of any failure.

Note that it is good practice to zero the property corresponding to the handle of a resource when that resource is released. Although not strictly necessary in the above example, in general it is useful as it prevents an attempt being made to release a resource that does not exist.

Using the CLEANUP list

An application may create temporary resources whose handles are stored on the stack, rather than in an object's property. Alternatively, even if the handles are stored in property, the resources may not be created or destroyed at the same time as the 'owning' object, and the roll-back on failure to create the resource may not need an object to be destroyed.

In such cases, roll-back can be performed by use of the `CLEANUP` object that is present in every HWIM application. This object is described in *The CLEANUP Class*, in the *OLIB Reference* manual.

Suppose that, the file *afile.txt* needs to be opened temporarily, together with the temporary creation of two allocated memory cells. On failure to create all of these three resources, any of them that have been created must be released and the error reported. Suitable code would be as follows:

```
VOID CreateResources(VOID)
    {
    VOID *fcb;
    UBYTE *p1,p2;
    INT clean1,clean2;

    f_open(&fcb,"AFILE.TXT");    /* just leave on error */
    clean1=cl_add_iochan(fcb);   /* add file handle to cleanup list */
    p1=f_alloc(100);             /* create first cell */
    clean2=cl_add_alloc(p1);     /* add first cell to cleanup list */
    p2=f_alloc(100);             /* if this succeeds, all resources are
created */
    cl_remove(clean1);           /* so we can remove both items... */
    cl_remove(clean2);           /* ... from the cleanup list */
    ...
    ...                          /* processing that can not fail */
    ...
    p_close(fcb);
    p_free(p1);
    p_free(p2);
    }
```

Remember that resources on the cleanup list will be removed by system code if `p_leave` is called in the `ao_run` method of any active object.

Interactions with system code

Care should be taken when errors arise in application-specific code when system code also needs to perform error recovery. A typical case is during the initialisation of a dialog, since it is system code that controls the roll-back from any partially complete creation of dialog objects.

In general this is not a problem since the application code does not normally need to take any specific action on an error condition. The application code can just call `p_leave` and leave the system code to perform all necessary error recovery.

This might not be the case if, exceptionally, the application-specific code needs to perform some specific action on detection of an error, such as reporting the error in a non-standard way, or sending some form of notification to another process. In such a situation the application code will normally trap errors by calling `p_enter` and perform local error handling when this call returns an error.

The preferred solution in such a case is to propagate the error to system code by calling `p_leave` (with the same error number as was returned from the call to `p_enter`) after the local error handling is complete. This will allow system code to perform its own error recovery, including reporting the error in a standard way. If the error has already been reported by application code, the error can be propagated by calling `p_leave(-1)`. This will enable any required error recovery in system code but will disable the standard error reporting.

A difficulty arises in the (fortunately rare) case where it is essential, for some reason, that the application code does not call `p_leave`. If, in such a case, system-owned resources may need to be released. then the application's error recovery code should at least send the application manager a `CL_CLEAN_LEVEL` message, which will normally be sufficient. There is, however, no guarantee that this will always be totally successful: such a situation should be avoided if at all possible.

CHAPTER 11

FILE-BASED APPLICATIONS

The general aspects of file-based applications for the Series 3 range of machines are described in the *Communicating with the System Screen* chapter of the *Series 3 Programming Guide*. This chapter assumes a basic familiarity with that material and concentrates on those aspects that are of significance to an object oriented application. The three main topics that are discussed are:

- start-up initialisation
- opening and creating files
- saving files

To maintain consistency with the built-in applications, all file-based applications should obey the general guidelines for such applications. They should, for example, store their files in a suitable subdirectory and applications that use record-based files should write their records in a flash-friendly manner (see, for example, the *Database Files* chapter of the *PLIB Reference* manual). It is a general rule that an application must keep its current file open, even if it is not actually reading from or writing to the file.

Start-up initialisation

As is described in the *Series 3 Programming Guide*, the command line that is passed to a file-based application when it is started contains the name of a file to be opened or created, the default file extension and any 'alias' information. System initialisation code analyses the command line and writes the information that it contains to a number of standard locations. Thus, by the time the application receives a `WS_DYN_INIT` message, to perform application-specific initialisation, the data is set up as follows:

- the full path name of the file to be opened or created is pointed to by the magic static `DatUsedPathNamePtr`
- whether the file is to be created or opened is determined by the `UBYTE` accessed by `w_am->hwimman.command`, which will contain either `H_COMMAND_CREATE_FILE` or `H_COMMAND_OPEN_FILE` (defined in *hwimman.g*)
- the default file name extension for the application's files is pointed to by an item of the application manager's property and is accessed by the `TEXT` pointer `w_am->hwimman.defext`
- the alias information, if required, is accessed via the `TEXT` pointer `w_am->hwimman.aliasinfo`

A convenient way of opening or creating the required file from within the application-specific initialisation code is to send the command manager a `COM_FILE_CHANGE` message of the form:

```
p_send4(w_ws->wserv.com,O_COM_FILE_CHANGE,w_am-
>hwimman.command,DatUsedPathNamePtr);
```

The required behaviour of this method is described in the following section.

Opening and creating files

A file-based application will normally have New and Open command menu options, to create a new file and to open an existing file. The corresponding command manager method functions would present suitable dialogs to specify a file name and any other relevant parameters.

On successful completion of the dialog, the opening of an existing file or the creation of a new file could conveniently be performed by calling a replacement of the `com_file_change` method of the application's subclass of the `COMMAN` command manager. A typical replacement would have the form indicated by the following code:

```
GLDEF_D TEXT filename[P_FNAMESIZE];

METHOD INT mycman_com_file_change(PR_MYCMAN *self,INT command,TEXT *pname)
    {
    SaveCurrentFile(self);
    p_scpy(&filename[0],pname);
    switch (command)
        {
    case H_COMMAND_CREATE_FILE:
        hEnsurePath(&filename[0]);
        CreateNewFile(self,&filename[0]);
        break;
    case H_COMMAND_OPEN_FILE:
        OpenExistingFile(self,&filename[0]);
        break;
        }
    p_send3(w_am,O_AM_NEW_FILENAME,&filename[0]);
    return(0); /* there has been no call to p_leave */
    }
```

where `SaveCurrentFile`, `CreateNewFile` and `OpenExistingFile` represent application-specific code to perform the corresponding actions.

The following points must be noted regarding this code:

- the method is also called by system code, under the protection of `p_enter`, and must return zero on successful completion. There is thus an implicit assumption that any failure within the method should result in `p_leave` being called. Application code may, if desired, take advantage of this, to trap and explicitly handle errors, by sending the message by means of `p_entersend`. An application will normally have to handle a failure to open or create a file by attempting to reopen the previously open file.

- it is standard practice to call the utility function `hEnsurePath` at any point where it is possible that the directory specified by a file specification might not exist. Although this is not guaranteed to succeed, and does not report an error on failure, it reduces the likelihood that the following operation (in this case, the creation of a file) could fail, merely because a directory has not yet been created.

- whenever an application switches to a new file it must, on successful completion of the operation, send the application manager an `AM_NEW_FILENAME` message, passing a pointer to a *permanent* buffer containing the full file specification of the new file. System code sets `DatUsedPathnamePtr` to point to this name, as required for correct operation of the System Screen. Note that, in the example, this buffer is, for clarity, implemented as static data. In a real application it would normally be part of the property of some object that remained in existence for the whole time that this file is the application's current file. In a simple application this object could be the command manager itself, but would normally be an object that represents the current file.

A common alternative scheme is illustrated by the Record application (whose code is supplied and is discussed in the *Application Design* chapter). In this case, the application's command manager has, for example, a `com_open_file` method that does not use the `com_file_change` method. Instead, both methods call common application-specific code.

Switchfiles messages

As discussed in the *Series 3 Programming Guide*, the System Screen can, at any time, send a *Switchfiles* message to a file-based application. System code within the application converts such a message to a

COM_FILE_CHANGE message, sent to the application's command manager. The receipt of this message may be handled in exactly the same way as described above.

An application that is temporarily unable to process a *Switchfiles* message may set the magic static DatLocked to a non-zero value, clearing it when it is again able to process such a message.

If opening a file takes an extended time, it would be sensible for an application to set DatLocked for the duration of this operation. In this case, the application must ensure that DatLocked is cleared on termination, even if the operation terminates on an error (which will generally result in p_leave being called).

Saving files

Saving a file is subject to many of the considerations already discussed in the previous section. An application will generally support at least Save and Save as menu options, with only the second of these requiring a dialog to select a file name.

If, after saving the file with a specific name, the current file takes the new file name, this must be reported by sending the application manager an AM_NEW_FILENAME message, as described above.

Saving the file may be an extended operation and should similarly be protected against *Switchfiles* messages by setting DatLocked.

Application termination

On termination of a file-based application by means of an Exit menu option, the command manager's com_exit method should save any outstanding changes to the current file automatically, without any notification to the user. Should an error occur during any such saving, the application should come to the foreground (see below). The user should then be notified of the error and offered the option of cancelling the Exit. Once the file is successfully saved (or, on failure, the user has elected to terminate the application) the com_exit method should either supersend the COM_EXIT message or, equivalently, call p_exit(0).

The following code illustrates a possible replacement com_exit method:

```
METHOD VOID mycman_com_exit(PR_MYCMAN *self)
    {
    INT error;

    error=p_enter2(SaveChanges,self);
    if (error)
        {
        wClientPosition(0,0); /* come to foreground */
        hErrorDialog(error,0);
        if (h2LineConfirm(-SYS_LOSING_CHANGES,-SYS_CONFIRM_CONTINUE))
            return;
        }
    p_supersend2(self,O_COM_EXIT);
    }
```

The code assumes that the application-specific function SaveChanges returns zero if no changes have been made or if saving the changed file was successful. Any p_leave caused by an error while saving the file is trapped by calling SaveChanges under the protection of p_enter and is explicitly reported (by use of the hErrorDialog utility function).

Shutdown messages

The System screen may, at any time, send an application a *Shutdown* message. System code within the application converts such a message to a COM_EXIT message, sent to the application's command manager. The receipt of this message may be handled in the same way as described above.

An application may receive a *Shutdown* message while it is a background process. To ensure that the user can see and respond to any error notification, the process must therefore come to foreground, as mentioned above.

Note that setting DatLocked disables *Shutdown* messages as well as *Switchfiles* messages.

CHAPTER 12

EDIT WINDOWS

This chapter explains how to use the HWIM EDWIN class to create edit windows that (to name but a few features)

- handle all standard cursor movement, selection, typing, and deletion keys
- can be either single-line or multi-line
- automatically scroll vertically and/or horizontally, whenever required
- automatically word-wrap, whenever required
- provide common editing functionality such as *Copy*, *Insert*, *Bring*, *Evaluate*, *Find*, and *Replace*.

All the features of Hwif edit boxes, available through the Hwif hEBxxx functions, are also available to HWIM programmers using EDWIN directly. (In fact, as can be confirmed by consulting the module *ehwif.c* in the optional *\sibosdk\hwifsrc* directory, the hEBxxx functions are just thin layers over calls to various methods of EDWIN.) However, programming directly at the EDWIN level opens up many additional possibilities. Some of these additional possibilities are:

- more efficient handling of larger amounts of text
- edit windows (and edit-*like* windows) which support "labels" in the left-margin
- edit windows with tabs and variable tabstops
- edit windows with multiple fonts and font styles.

In fact, the main editing window in the Word Processor application built into the Series 3 is a subclass of EDWIN - as are the main windows of the Database and Program Editor applications.

In order to achieve effects like this, programmers need to become acquainted with some of the *component* objects utilised by EDWIN - for example the EPDOC document object, the SCRIMG screen image object, and the SCRLAY screen layout object. Later sections of this chapter provide an introduction to these additional objects (which are all instances of classes in FORM). However, many of the aspects of EDWIN can be accessed without any knowledge of the internal structure of the class. These aspects are described in the earlier sections of this chapter.

Introduction to EDWIN

Dialogs and edit windows contrasted

The first use a programmer normally makes of EDWIN is by having an edit box in a dialog. (See the chapter *Dialog Controls* for information on how to program edit boxes in dialogs.)

In this case, the initialisation of the edit window is taken care of by system code. System code likewise ensures that

- keypresses are passed to the edit box at the right time
- the edit box is always displayed in the correct "emphasis" state (ie with its cursor flashing or not, as the case may be, and with any select region highlighted when appropriate).

The responsibility of the programmer in this case is merely to

- choose which initialisation flags to define
- set text into the edit box when needed
- sense the contents of the edit box, after the user has edited them.

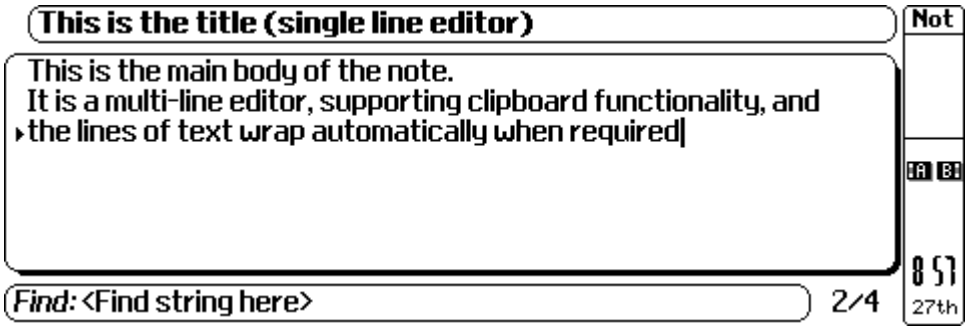
In contrast, when an application has an edit window *outside* a dialog box, the programmer has to accept the following additional responsibilities

- creating the edit window to start with - by filling in fields in the `IN_EDWIN` data structure (and, optionally, in the `IN_EDWIN_X` auxiliary data structure)
- deciding when the edit window should receive keys - and passing these keys onto the edit window
- deciding when the edit window should be emphasised.

The programmer also has to draw any *border* required for the edit window; the `EDWIN` class has, itself, no notion of a border.

The NOTES example program

The examples in the first half of this chapter are mainly based on the example application *Notes.app*. The source code of this application is placed in the directory *\sibosdk\notes* if the optional OOPDEMO component of the SIBO C SDK is installed. This application may be recognised as an HWIM version of one of the Hwif example programs. It creates *three* different edit windows, as can be seen in the following screen shot:



In this screen shot, the emphasis is currently with the middle editor - as can be seen from the flashing cursor at the end of the third line, and also by the "margin cursor" in the left margin. These visual indications as to which editor has the emphasis will of course disappear when the emphasis (sometimes also called the "keyboard focus") is moved elsewhere.

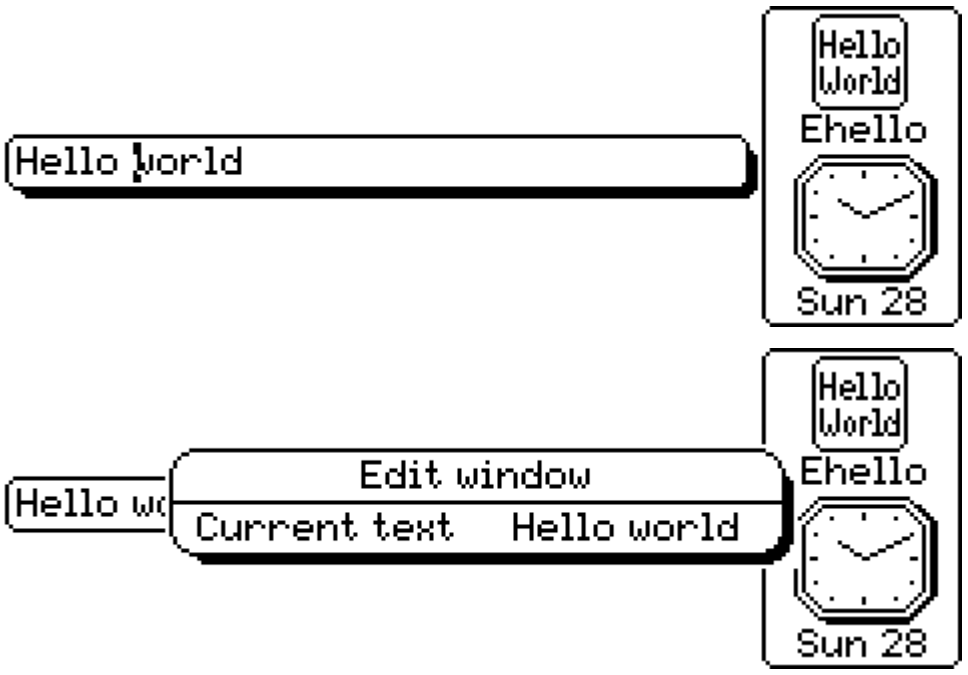
The three editors are each drawn within their own border - which is provided by creating the editor within an instance of the `BWIN` bordered window class. As suggested above, the `EDWIN` class itself does not make any call to any variant of `gBorder`: its concern is purely with the text inside the border.

The screen shot is of this application running on a Series 3a. The application also runs on a Series 3, and in this case, there are fewer lines in the middle editor. In fact, as well as illustrating use of the `EDWIN` class, the application provides an example of how to write a fully-resizeable application, that can run on either the Series 3 or the Series 3a (exactly the same image program runs on the two different models, and would also run intelligently on machines with intermediate screen sizes). However, this feature of this application is incidental to the main theme of this chapter and will not be mentioned again.

The "Hello World" program for edit windows

Because *Notes* is a fairly well developed example program, the basic architecture of programming an edit window may to some extent be hidden, in its source code, by the lots of other concerns that have to be taken care of by that program. For this reason, installing the optional OOPDEMO component of the SDK also places the source code for a much simpler example, *ehello.img*, into the *\sibosdk\ehello* directory.

This program simply draws a one-line edit window in the middle of the screen, and diverts most incoming keypresses to that window. The only exception is the ENTER keypress, which throws up a simple dialog confirming the text currently in the editor. The following two screen shots demonstrate, respectively, the main state of the application, and the confirmation dialog:



The EHELLO category file

The category file for the *ehello* application, *ehello.cat*, defines only three classes:

```
IMAGE ehello

EXTERNAL olib
EXTERNAL hwim

INCLUDE hwimman.g
INCLUDE edwin.g

CLASS ehwserv wserv
    {
    REPLACE ws_dyn_init
    }

CLASS ehbwin bwin
    {
    REPLACE wn_init
    REPLACE wn_emphasise
    REPLACE wn_key
    REPLACE wn_draw
    PROPERTY
        {
        PR_EDWIN *edwin;
        }
    }

CLASS ehdlg dlgbox
    {
    REPLACE dl_dyn_init
    }
```

The bulk of what little code there is in the application resides in the `EHBWIN` class. As can seen, `EHBWIN` is a custom-subclass of `BWIN`, and owns a component `EDWIN` object. In technical terms - elaborated below - `EHBWIN` is the “landlord” for the edit window in this application.

Initialisation code in EHELLO

The code in `main` has the standard form:

```
GLDEF_C VOID main(VOID)
    {
    IN_HWIMMAN app;
    IN_WSERV wserv;

    p_linklib(0);
    app.flags=FLG_APPMAN_RSCFILE|FLG_APPMAN_SRSCFILE|FLG_APPMAN_CLEAN;
    app.wserv_cat=p_getlibh(CAT_EHELLO_EHELLO);
    app.wserv_class=C_EHWSERV;
    wserv.com_cat=p_getlibh(CAT_EHELLO_HWIM);
    wserv.com_class=C_COMMAN;
    p_send4(p_new(CAT_EHELLO_HWIM,C_HWIMMAN),O_AM_INIT,&app,&wserv);
    }
```

After `main`, application code is next called in the `ws_dyn_init` method of the `EHWSERV` class:

```
METHOD VOID ehwserv_ws_dyn_init(PR_EHWSERV *self)
    {
    wsEnable();
    self->wserv.cli=f_newsend(CAT_EHELLO_EHELLO,C_EHBWIN,O_WN_INIT);
    }
```

Evidently, this creates and initialises the client window for the application - an instance of `EHBWIN`. In turn, the `wn_init` method of `EHBWIN` is as follows:

```
METHOD VOID ehbwin_wn_init(PR_EHBWIN *self)
    {
    W_WINDATA wd;
    IN_EDWIN_X initx;
    struct
        {
        IN_EDWIN e;
        TEXT rest[21];
        } init;

    wd.extent.width=240-50; /* extent calculation presupposes SERIES 3 screen
*/
    wd.extent.height=3+10+5;    /* matches flags set for bwin below */
    wd.extent.tl.x=0;
    wd.extent.tl.y=31;      /* centred vertically */
    p_send5(self,O_WN_CONNECT,NULL,W_WIN_EXTENT,&wd);
    hLoadResBuf(EHSTR_INIT,&init.e.contents[0]);
    init.e.vulen=240-50-3-1-5-1;    /* one extra pixel clearance each end */
    init.e.maxlen=50;
    init.e.flags=IN_EDWIN_VULEN_PIXELS|IN_EDWIN_POSITION_SUPPLIED;
    initx.pos.x=3+1;
    initx.pos.y=3;
    self->ehbwin.edwin=f_newsend(CAT_EHELLO_HWIM,C_EDWIN,O_WN_INIT,
                                                    &init,self,&initx);
    self->win.flags=IN_BWIN_SHADOW_2|IN_BWIN_CUSHION;
    p_send3(self,O_WN_EMPHASISE,TRUE);
    hInitVis(self);
    }
```

Without going into details at the moment, the basic form of this method can still be pointed out:

- connect the window to the Window Server (by sending `self` a `wn_connect` message)
- create and initialise the `EDWIN` component object
- make this window tree visible.

Other code in EHELLO

The other three methods of `EHBWIN` mainly just delegate responsibility appropriately, between the `EDWIN` component and the `BWIN` superclass (see later for a fuller explanation of what is going on here):

```
METHOD VOID ehbwin_wn_emphasise(PR_EHBWIN *self,INT flag)
    {
    p_supersend3(self,O_WN_EMPHASISE,flag);
    p_send3(self->ehbwin.edwin,O_WN_EMPHASISE,flag);
    }
```

```
METHOD VOID ehbwin_wn_draw(PR_EHBWIN *self)
    {
    p_supersend2(self,O_WN_DRAW);
    p_send2(self->ehbwin.edwin,O_WN_DRAW);
    }

METHOD VOID ehbwin_wn_key(PR_EHBWIN *self,INT keycode,INT mods)
    {
    SE_EDWIN sense;

    if (keycode!=W_KEY_RETURN)
        p_send4(self->ehbwin.edwin,O_WN_KEY,keycode,mods);
    else
        {
        p_send3(self->ehbwin.edwin,O_WN_SENSE,&sense);
        LaunchDialog(C_EHDLG,EHDLG,sense.buf);
        }
    }
```

The utility routine `LaunchDialog` has the standard form

```
LOCAL_C VOID LaunchDialog(INT class,INT resid,VOID *rbuf)
    {
    DL_DATA dld;

    dld.id=resid;
    dld.rbuf=rbuf;
    dld.pdlg=NULL;
    hLaunchDial(CAT_EHELLO_EHELLO,class,&dld);
    }
```

and, in turn, the `dl_dyn_init` method of the `EHDLG` dialog box class merely sets the text of the edit window into a text window in the dialog:

```
METHOD VOID ehdlg_dl_dyn_init(PR_DLGBOX *self)
    {
    hDlgSetText(1,self->dlgbox.rbuf);
    }
```

Simple use of EDWIN

This section explains: how to initialise an edit window, how to pass keys to it, how to set text into it and sense text out of it, how to pass `wn_draw` and `wn_emphasise` messages onto it, and the basic format of the text stored in it.

Initialising an instance of EDWIN

Often, the hardest aspect of incorporating an edit window in an application is initialising it correctly. Once the edit window has been set up properly, it handles most features automatically.

In order for an edit window to be drawn on the screen, the following steps are required:

1. an instance of the `EDWIN` class (or a subclass thereof) has to be created
2. a `wn_init` message must be sent to the object, with suitable parameters (see below)
3. the window tree of which the editor is part must be made visible, usually via a call to the utility function `hInitVis`.

These steps may take place in code such as the following:

```
IN_EDWIN init;
IN_EDWIN_X initx;
...
edwin=f_newsend(CAT_NOTES_HWIM,C_EDWIN,O_WN_INIT,&init,landlord,&initx);
...
hInitVis(landlord);
```

The landlord of the edit window

In the above code fragment, the variable `landlord` is the handle of an instance of (a subclass of) the HWIM `WIN` class which has connected to the Window Server. In the language of the *Windows* chapter in this manual (to which the reader is referred for background information on such concepts as "lodger windows"), the `landlord` object contains a notional `WSWIN` component. In other words, a `wn_connect` message has been sent to the `landlord` object, and the field `landlord->win.id` has been filled in as a result.

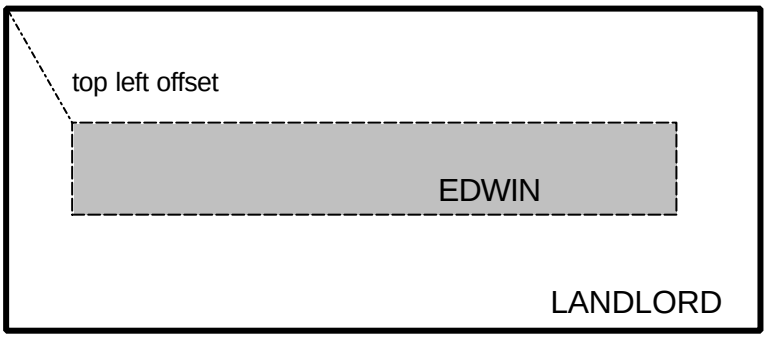
Note that the value of `landlord->win.id` must be filled in *before* the `wn_init` message is sent to the `EDWIN` object.

Frequently, the `landlord` object is an instance of a (subclass of) the HWIM `bwin` class.

Incidentally, whereas in the *Notes* example application, each of the three editors has its own unique landlord window, there is no general requirement for a given landlord window to contain only one editor. For example, if there are three editors in one dialog, that dialog window (an instance of `dlgbox`) is the common landlord of all three editors.

As with all lodger windows, an `EDWIN` object requires to know

- the top left offset within the landlord, to the rectangle occupied by the lodger
- the width of the rectangle occupied
- the height of this rectangle:



In the case of `EDWIN`:

- the height is worked out, inside the `wn_init` method, from a knowledge of the number of lines that are to be visible at one time (this defaults to one), the font in which the text in the editor is to be displayed, and the vertical leading to be applied with this font
- the width is also worked out, by one of a variety of different means, depending on which flags are set on initialisation
- the top-left offset always has to be supplied explicitly, in pixels - although as is explained below, it is possible to *defer* providing this value until later in the overal initialisation process.

The IN_EDWIN and IN_EDWIN_X data structs

The `IN_EDWIN` and `IN_EDWIN_X` data structs are defined as follows in *edwin.cl*:

```
typedef struct
    {
    UWORD vulen;        viewing length or width
    UWORD flags;        autoselect etc
    UWORD maxlen;       maximum number of characters allowed
    TEXT contents[1];   rest of initial contents follows in line
    } IN_EDWIN;

typedef struct
    {
    WORD total;
    WORD top;
    } EDWIN_LEADING;
```

```
typedef struct
    {
    UWORD vislines;             number of lines visible in window
    P_POINT pos;                top left offset relative to landlord
    WORD font;                  font id
    UWORD style;                font style
    EDWIN_LEADING leading;      total and top-only vertical leadings
    VOID *doc;                  document object to use
    VOID *clip;                 possible clipboard to use
    } IN_EDWIN_X;   never used in edwins in dialogs
```

Whilst the address of an `IN_EDWIN` struct must *always* be passed to the `wn_init` method of `EDWIN`, it is optional whether to pass the address of an `IN_EDWIN_X` struct. This is because, depending on various bit values that can be set in the `flags` field in the `IN_EDWIN` struct, default values are assumed for the fields that can be supplied in the `IN_EDWIN_X` struct. More precisely:

- unless `IN_EDWIN_VISLINES_SUPPLIED` is set in `flags`, the value of `vislines` is taken as `1`
- the value of `pos` is ignored unless `IN_EDWIN_POSITION_SUPPLIED` is set (if this bit is clear, the value of `pos` must be supplied by a subsequent call to `lg_set_id_pos` - see below)
- unless `IN_EDWIN_FONT_SUPPLIED` is set, the value of `WS_FONT_SYSTEM` is assumed for `font`, and the value `G_STY_NORMAL` is assumed for `style`
- unless `IN_EDWIN_LEADING_SUPPLIED` is set, values of `2` and `1` are assumed for `leading.total` and `leading.top`
- unless `IN_EDWIN_DOC_SUPPLIED` is set, any supplied value of `doc` is ignored, and the edit window automatically creates a suitable document object (see later in this chapter for further discussion of document objects)
- the value of `clip` is ignored unless `IN_EDWIN_CLIPBOARD` is set (specifying that the editor is to support clipboard functionality).

As can be appreciated, edit boxes in dialogs are never passed an address of an `IN_EDWIN_X` struct; all the corresponding flags are clear. This reflects the fact that the `EDWIN` resource struct, defined in *hwim.rh*, corresponds just to the `IN_EDWIN` struct, and does not have any fields matching the `IN_EDWIN_X` struct.

The lg_set_id_pos method

In case it is impossible (or particularly inconvenient) to give the value of `pos`, the top-left offset of the editor within its landlord, at the time when the `wn_init` method has to be called (this is the case when editors are created within dialogs, as the dimensions of a dialog cannot be determined until all the items in the dialog have been initialised), this can be given later, by sending the editor an `lg_set_id_pos` message as follows:

```
P_POINT pos;

...
p_send5(edwin,O_LG_SET_ID_POS,landlord->win.id,&pos,width);
```

where `width` is the width of the region the editor is to display itself upon.

Other edit window initialisation flags

In addition to the six `IN_EDWIN_XXX` flags mentioned above, there are four other groups of possible values that the `flags` field in the `IN_EDWIN` struct can contain:

- values governing the interpretation of the `vulen` field, if set
- values influencing the behaviour of the `wn_key` method of the editor (ie influencing the way the editor responds to various keypresses passed to it)
- values governing the initial cursor position and initial highlighting (more precisely, these control the cursor position and the highlighting following initialisation *and* following any call to `wn_set` to set text into the editor)
- miscellaneous other values governing attributes the edit window may or may not possess.

In detail, these ten additional values are as follows:

`IN_EDWIN_VULEN_CHARACTERS`	if this is set in `flags`, the value of `vulen` is multiplied by the `max_width` value for the font to be used by the editor, and the result is used for the width of the editor
`IN_EDWIN_VULEN_PIXELS`	if this is set in `flags`, the value of `vulen` is taken directly as the width to be used by the editor (if *neither* this flag or the previous one is set, then the width is set to the product of the `max_width` value of the font and the `maxlen` value from the `IN_EDWIN` struct)
`IN_EDWIN_ACCEPT_TABS`	unless this is set, the `wn_key` method of the editor will reject the TAB key
`IN_EDWIN_ACCEPT_SOFT_HYPHENS`	unless this is set, the `wn_key` method will reject the CONTROL-hyphen key (which would otherwise insert a so-called “soft” hyphen - which would be invisible in most cases)
`IN_EDWIN_DIALLABLE`	unless this is set, the `wn_key` method will reject the SHIFT-DIAL and the CONTROL-SHIFT-DIAL keys (if this *is* set, the `wn_key` method will, respectively, insert an `0x05` telephone symbol, or run the system ‘Append country markup’ dialog)
`IN_EDWIN_NO_AUTOSELECT`	if this is set, the initial cursor position is set to the start of the text, and there is no initial select region
`IN_EDWIN_AUTO_CUR_END`	this has the same effect as the previous flag, except that the initial cursor position is set to the end of the text (if *neither* this flag or the preceding one is set, the cursor is placed at the end of the text and the entirety of the text is selected)
`IN_EDWIN_LEFT_CURSOR`	if this is set, a triangular pointing cursor is displayed down a left-hand margin (as in the middle of the three editors in the *Notes* example application), using the same font as the main application but with all bits of the font style cleared apart from the `G_STY_DOUBLE` bit (if set)
`IN_EDWIN_TEXT_SEGMENTED`	if set, this means that the editor will create an `EPSEG` document object whenever needed (as opposed to the `EPFLAT` object that is created by default - see later in this chapter for more discussion of document objects)
`IN_EDWIN_PAGINATABLE`	(for advanced use only - see later).

A note on the CONTENTS field in the IN_EDWIN struct

Evidently, the `IN_EDWIN` struct only defines one element in a possible `contents[]` array. In order to specify initial contents different from just the null string (`""` - specified when `init.contents[0]` is zero), the application needs to make a declaration such as

```
struct
    {
    IN_EDWIN e;
    TEXT rest[LENGTH-1+1];
    } init;
...
init.e.maxlen=...;
...
p_scpy(&init.e.contents[0],pInitString);
...
edwin=f_newsend(C_APP_HWIM,C_EDWIN,O_WN_INIT,&init,...);
```

Note that the array `contents[]` is *ignored* altogether if the flag `IN_EDWIN_DOC_SUPPLIED` is set in `flags`. Apart from this, `&init.contents[0]` is always interpreted as a pointer to a zero terminated string.

Alternative means of defining initial text for an editor (these methods can circumvent the limitation to setting text that is only one paragraph long) include:

- using the `wn_set` or other methods of the `EDWIN` class, possibly repeatedly, before the editor is made visible
- setting the text directly into the document object.

Values of special characters in the text

The following values are used to represent special characters within an editor:

`7 (SCRLAY_SYM_HARD_HYPHEN)`	a minus sign (sometimes called a “nonbreaking hyphen”) that does not count as a word delimiter - normally entered at the keyboard using SHIFT-CONTROL-hyphen
`8 (SCRLAY_SYM_SOFT_HYPHEN)`	a soft (sometimes called an “optional”) hyphen - normally invisible, but will transform into a visible hyphen to allow a word to wrap over two lines at the point, if required - normally entered at the keyboard using CONTROL-hyphen
`15 (SCRLAY_SYM_HARD_SPACE)`	a space that does not count as a word delimiter - normally entered at the keyboard using SHIFT-CONTROL-SPACE
`0 ('\0')`	a paragraph end - normally entered at the keyboard using ENTER
`10 ('\n')`	a forced line break - normally entered at the keyboard using SHIFT-ENTER
`9 ('\t')`	a tab character - normally entered at the keyboard using the TAB key
`5 (WS_SYMBOL_PHONE)`	a telephone character - normally entered at the keyboard using SHIFT-DIAL.

Apart from the above values, characters with values less than `32` should in general *not* be set into edit windows. Note in particular that characters with values less than `4` are all treated, by low-lying code (in the OLIB library) as *paragraph delimiters* - with potentially bizarre results, given that the formatting code in FORM only recognises the character value `0` as a paragraph delimiter.

Note that no characters are written to the buffer to denote the location of line ends caused merely by word-wrap. These locations (sometimes called “soft carriage returns”) have no fundamental significance:

- they change whenever the window size changes (eg when the status window is altered) or the display font is “zoomed”
- they are calculated dynamically when needed, and are held in a different part of property of the edit window (actually in the `SCRLAY` screen layout component).

A note on the MAXLEN field in the IN_EDWIN struct

For clarity, it should be emphasised that the `maxlen` field in the `IN_EDWIN` data struct specifies the maximum number of characters the editor can contain, *not counting any final terminating character*.

For example, an editor with `maxlen` set to `6` could contain the string `"abcdef"` (where the editor would in fact also store a terminating zero at the end of the string.

However, if the editor was multi-line, it could *not* store the text `"ab\0cdef"` (representing a paragraph of two characters followed by one of four characters): that would require a `maxlen` of (at least) `7`.

The wn_sense method

For many uses of edit windows, the following model is sufficient to explain the storage of text within the editor: the text is stored as a zero terminated string, and the `wn_sense` method of `EDWIN` provides access to this buffer, as follows:

```
typedef struct
    {
    TEXT *buf;
    UWORD len;
    } SE_EDWIN;

SE_EDWIN sense;
...
p_send3(edwin,O_WN_SENSE,&sense);
p_scpy(&store[0],sense.buf);
```

Note however that the buffer whose address (`sense.buf`) is obtained in this way *must* always be regarded as read only. In order to change the contents of this buffer, the various methods of `EDWIN` (or methods of *components* of `EDWIN`) have to be used.

Second, note that the buffer used by the editor may *move* as more text is added. This is because the buffer is resized according to how much text it contains. Therefore, it would be a grave mistake to hold onto the address of this buffer, and assume that this will still be valid after more text could have been added into the editor.

Third, as mentioned above, paragraph ends (in multi-line editors) are internally represented as zeros. However, code such as

```
p_send3(edwin,O_WN_SENSE,&sense);
p_scpy(&store[0],sense.buf);
```

will only succeed in copying out text as far as the first embedded zero. For this reason, the *Notes* example application essentially uses the following code instead:

```
SE_EDWIN sense;
...
p_send3(edwin,O_WN_SENSE,&sense);
p_bcpy(&store[0],sense.buf,sense.len);
```

Finally, note that this still assumes that the text is stored in a *flat* buffer. This applies by default, but the initialisation flag `IN_EDWIN_TEXT_SEGMENTED` can be used to specify that the text is stored in a *segmented* buffer. (Roughly speaking, the larger the quantity of the text, the more pressing the need to store it in a segmented buffer - to cut down on the amount of data that needs to be shuffled along each time a single character is typed into the middle of the document.) If the text *is* stored segmented, the result of the `wn_sense` method is undefined, and other means are required to sense the contents of the editor.

The wn_set method

The `wn_set` method can be used to completely replace the contents of an `EDWIN` object.

It uses the same `SE_EDWIN` struct as does the `wn_sense` method:

```
SE_EDWIN set;
...
p_send3(edwin,O_WN_SET,&set);
```

After this method, the contents of the editor are the `set.len` characters in the buffer pointed to by `set.buf`. (The operation of the method involves copying these characters into the internal storage buffers of the edit window.) All previous contents are discarded. The highlight and the cursor position are adjusted according to the `IN_EDWIN_AUTO_CUR_END` and `IN_EDWIN_NO_AUTOSELECT` flags specified on initialisation.

The wn_key method

The way to pass keypresses onto an edit window is to send a `wn_key` message as follows

```
p_send4(edwin,O_WN_KEY,keycode,modifiers);
```

The method actually returns one of the following two values:

- `WN_KEY_CHANGED` - the contents of the edit window changed as a result of the keypress
- `WN_KEY_NO_CHANGE` (which is the same as `FALSE`) - the contents of the edit window did *not* change as a result of the keypress.

In most cases, however, this value will be ignored by application code that passes the key to the window.

Something else that is automatically suitable in most uses of `EDWIN` is the behaviour of the `wn_key` method when the keypress passed to the editor

- causes an out-of-memory error, or
- would cause the maximum capacity of the editor to be exceeded.

See the discussion of the `ew_leave` method below for more information on how `EDWIN` copes with these two cases.

The wn_emphasise method

Although the need for the `wn_key` method is clear, the need for the `wn_emphasise` method may be less so. However, as mentioned several times in this manual, there is a definite need for applications to track the "emphasis" as it moves around an application:

- into the menu bar and out again
- into the Help subsystem and out again, or into dialogs and out again
- around various parts of the main viewing screen of the application.

The parameters to the `wn_emphasise` method of `EDWIN` are the same as those for any other window class within HWIM:

```
p_send3(edwin,O_WN_EMPHASISE,flag);
```

where `flag` is either `FALSE`, to indicate that emphasis is moving away from the edit window, or `TRUE`, to indicate that emphasis is moving *to* the edit window.

The wn_draw method

The `wn_draw` method shares with `wn_key` and `wn_emphasise` the feature that landlord windows for edit windows invariably have to pass these messages onto their `EDWIN` components.

As far as `EDWIN` is concerned, there are no additional parameters to the `wn_draw` method (in particular, the entire visible portion of the editor has to be redrawn every time).

Additional EDWIN methods

This section explains some additional methods of `EDWIN`:

- methods for inserting text, finding text, and replacing text
- the copy and insert ("paste") clipboard methods
- the `ew_evaluate` method
- additional setting and sensing methods
- how to detect if the contents of an edit box has been changed
- edit boxes set to be "read only"
- the `ew_leave` method for notifying run-time errors.

The ew_insert method

The following code will result in the `blen` characters at `*buf` being inserted into the editor with handle `edwin`:

```
p_send4(edwin,O_EW_INSERT,buf,blen);
```

The characters are inserted at the cursor position (any selection being cancelled first), and the cursor is advanced to the end of the characters inserted.

The `ew_insert` method calls `ew_leave` if any error occurs.

The ew_find method

It is possible to request an editor to search for given text within itself. The following code is used

```
val=p_send4(edwin,O_EW_FIND,pstr,flags);
```

where `pstr` points to a zero-terminated string of the text to match, and possible bit values in `flags` are:

- `EWF_BACKWARDS` to search backwards from the cursor position (the default is to search forwards from the cursor position)
- `EWF_CASESENS` to make the search case sensitive (the default is for the search to be case insensitive).

The `ew_find` method returns `FALSE` if no match was found, and otherwise `TRUE` - in which case the matched text is highlighted as the new select region. The `ew_find` method is intelligent enough to code with repeated calls to `ew_find` without finding the same text repeatedly.

Note that matches across paragraph boundaries are not possible (this is consistent with the search string being zero terminated: it cannot contain an embedded zero).

The ew_replace method

The `ew_replace` method is in some ways similar to the `ew_insert` method, but it is designed primarily to implement a 'Replace' menu command (in conjunction with the `ew_find` method, which is designed to implement a 'Find' menu command). Whereas `ew_insert` *cancels* any selection before inserting the specified characters, `ew_replace` starts by *deleting* any selection. Another difference is that `ew_insert` takes its insertion text in the `(buf,len)` form, whereas `ew_replace` expects a zero terminated string. Finally, whereas `ew_insert` always leaves cursor at the bottom end of the selection region consisting of the text just inserted, `ew_replace` allows the cursor to be positioned at either end of this selection - in order to support repeated forward or backward text replacement, without entering an infinite recursion:

```
p_send4(edwin,O_EW_REPLACE,replace,backwards);
```

The replacement string is specified by the zero-terminated string `*replace`, and the flag `backwards` specifies whether the cursor should be placed at the top end of the selection (if `backwards` is `TRUE`) or at the bottom end.

The method attempts to insert the replacement text first, before deleting the existing selection (if any), so that any out-of-memory error is handled automatically without the loss of any text.

The `ew_replace` method calls `ew_leave` if any error occurs.

The ew_replace_clip method

The `ew_replace_clip` method is provided to implement a 'Copy text' menu command, in conjunction with the `ew_paste_clip` method (discussed next), which is provided to implement an 'Insert text' menu command (sometimes called a 'Paste' menu command).

Both these methods presuppose that the flag `IN_EDWIN_CLIPBOARD` was set on initialisation - otherwise the editor will panic (panic `55`). Note however that the `IN_EDWIN_CLIPBOARD` flag should *not* be set unnecessarily (ie if no calls to `ew_replace_clip` or `ew_paste_clip` are to be made), since this entails an additional memory overhead.

If `IN_EDWIN_CLIPBOARD` is set on the initialisation of the editor, the value of the `clip` field in the `IN_EDWIN_X` initialisation struct becomes significant:

- if this is non-`NULL`, it is taken as the handle of a suitable clipboard object to be used by the editor
- otherwise, the editor creates a clipboard object for its own use, which is in fact an instance of the OLIB `EPFLAT` class (unless the flag `IN_EDWIN_TEXT_SEGMENTED` was set on initialisation, in which case an instance of the OLIB `EPSEG` class is used).

In most cases, setting `clip` to `NULL` will be perfectly sufficient. The main exception is if the clipboard object has to persist beyond the lifetime of the editor (or if the clipboard is to be shared between two editors that both exist at the same time). Note here that the `destroy` method of `EDWIN` sends a `destroy` message in turn to any clipboard object that the editor itself created - whereas clipboard objects specified via a non-`NULL` value of the `clip` field of the `IN_EDWIN_X` struct passed at initialisation do *not* get destroyed in this way (that responsibility falls to the owner of the editor).

An application that wishes to create a clipboard for external purposes can use code such as

```
clip=f_newsend(CAT_APP_OLIB,C_EPFLAT,O_EP_INIT,maxlen+1);
```

where the reason why `1` is added to `maxlen` (the value used to initialise the editor itself) is that space has to be reserved, in the document object, for the final paragraph delimiter as well. In general, *any* realisable subclass of the OLIB class `EPROOT` can be used as the clipboard.

Note that there is no `EDWIN` method corresponding directly to any 'Cut' menu command. There is of course no such menu command on the ROM-resident Series 3 applications; the way that text is "cut" into the clipboard is that the user highlights the text and simply presses DELETE. This keypress is received by the `EDWIN` code in the `wn_key` method, and this is the location of code to copy the deleted text into the clipboard (if present).

Having said all that, in practice use of the `ew_replace_clip` method is simplicity itself. For example, the corresponding code in the *Notes* example application is just

```
METHOD VOID nocomman_ncoe_copy(PR_NOCOMMAN *self)
    {
    CheckEditing(self);
    if (!(p_send2(DatApp3,O_EW_REPLACE_CLIP)))
        hInfoPrint(NOSTR_NO_TEXT_COPIED);
    else
        hInfoPrint(NOSTR_TEXT_COPIED);
    }
```

In this example, `DatApp3` holds the handle of the editor.

The `ew_replace_clip` method in fact returns the length of the current selection - which is therefore zero if there is nothing to copy.

The ew_paste_clip method

Use of the `ew_paste_clip` method is just as simple. The corresponding code in the *Notes* application is

```
METHOD VOID nocomman_ncoe_insert(PR_NOCOMMAN *self)
    {
    CheckEditing(self);
    if (!(p_send2(DatApp3,O_EW_PASTE_CLIP)))
        hInfoPrint(NOSTR_NO_TEXT_INSERTED);
    }
```

The `ew_paste_clip` method returns the length of the text in the clipboard - which is zero if there is nothing to insert.

The ew_evaluate method

The `ew_evaluate` method can usefully be discussed alongside `ew_replace_clip` and `ew_paste_clip` because

- an 'Evaluate' menu command would normally be found alongside those for 'Copy text' and 'Insert text'
- the usage of this method is, in practice, equally as straightforward (although, in all three cases, a great deal happens behind the scenes).

The `ew_evaluate` method can, however, be used without the editor having been initialised with `IN_EDWIN_CLIPBOARD`.

The code in the *Notes* application that implements the 'Evaluate' menu command there is

```
METHOD VOID nocomman_ncoe_evaluate(PR_NOCOMMAN *self)
    {
    CheckEditing(self);
    p_send2(DatApp3,O_EW_EVALUATE);
    }
```

Note that whereas it is the responsibility of the application to detect "errors" (such as *Nothing to insert*) in the case of `ew_replace_clip` and `ew_paste_clip`, syntactical errors within the string to be evaluated are signalled by code within the `ew_evaluate` method - with the cursor being positioned to the error and an appropriate `hInfoPrint` being executed.

The ew_set method

For some purposes, the additional control provided by the `ew_set` method (as compared to the `wn_set` method) may be helpful:

```
typedef struct
    {
    UWORD flags;
    SE_EDWIN txt;
    UWORD cursor;  cursor (moving point of select)
    UWORD anchor;  anchor point (fixed end of select)
    } SET_EDWIN;

SET_EDWIN set;
...
p_send3(edwin,O_EW_SET,&set);
```

As well as containing an `SE_EDWIN` struct within itself, the `SET_EDWIN` struct evidently also allows the cursor and select region to be defined more precisely. This is governed by the possible values in `flags`:

- if `SET_EDWIN_EMPTY` is set, this is merely a convenient way to empty the contents of the editor
- if `SET_EDWIN_TXT` is set, a call to the `wn_set` method of the editor is effectively made
- if `SET_EDWIN_SEL_ALL` is set, the entire contents of the editor are selected
- if `SET_EDWIN_CUR_END` is set, the cursor is set to the end of the document
- if `SET_EDWIN_ANCHOR` is set, the anchor point of the selection (the non-moving end) is set to document offset `set.anchor`
- if `SET_EDWIN_CURSOR` is set, the cursor (which is also the moving end of the selection, if one exists) is set to document offset `set.cursor`.

The ew_sense method

Whereas the `ew_set` method allows more control, compared to `wn_set`, over features of an edit window that can be *set*, the `ew_sense` allows additional editing details (namely, the two ends of the select region) to be *sensed*:

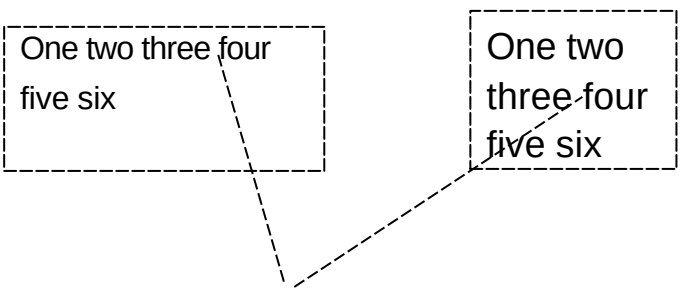
```
typedef struct
    {
    UWORD cursor;  cursor (moving point of select)
    UWORD anchor;  anchor point (may equal cursor)
    } SENSE_EDWIN;

SENSE_EDWIN sense;
...
p_send3(edwin,O_WN_SENSE,&sense);
```

In fact, this method always returns the top end of the select region (if any) in `sense.anchor`, and the bottom end in `sense.cursor`. If the length of the select region is presently zero, both the two fields in the `SENSE_EDWIN` return the cursor position.

The concept of document offset

Both the `ew_set` and `ew_sense` methods make use of the concept of *document offset*. In fact, this is a fundamental notion for the `EDWIN` class as a whole. The idea is that although features like *line number* and *line offset* vary according to the zoom state and status window setting, the document offset of a given location with the editor remains constant, regardless of how the contents are viewed.

For example, suppose an editor has its width decreased and its display font zoomed larger, causing the word-wrap to change. Then the possible cursor location just in front of, say, the ‘f’ of “four”



is on the first line in one case but on the second line in the second case. The line offset of this location also changes, but the *document offset* remains constant: the location has document offset `14` in both cases.

Allowed values of document offset

If there are `n` characters in a document - not counting the final terminating character zero - then there are precisely `n+1` allowed character offsets, ranging in value from `0` to `n`.

Note that the cursor cannot be positioned *beyond* the final terminator (ie at document offset `n+1`). Nor in fact can the cursor ever be positioned beyond the final character on any line - it always repositions automatically in these cases to the very beginning of the following line.

The EDWIN.CHANGE property

Although many parts of the property of `EDWIN` should be regarded as private, the `change` field is open to direct read-write manipulation by application code.

For example, the following routine in the *Notes* application checks whether any change has been made to the editor in a given window, and if so

- senses the new text, and records it
- resets the value of `edwin.change` to `FALSE`:

```
LOCAL_C VOID RecordChange(PR_NOWIN *win,LENBUF *lb)
    {
    PR_EDWIN *edwin;
    SE_EDWIN sense;

    edwin=win->nowin.edwin;
    if (!(edwin->edwin.change))
        return;
    p_send3(edwin,O_WN_SENSE,&sense);
    lb->buf=f_realloc(lb->buf,sense.len);
    lb->len=sense.len;
    p_bcpy(lb->buf,sense.buf,sense.len);
    edwin->edwin.change=FALSE;
    }
```

In fact there are *two* different `EW_CHANGE_XXX` bit flags defined in *edwin.cl*: `EW_CHANGE_SINCE_SAVED` and `EW_CHANGE_SINCE_PAGINATE`, with the values `0x01` and `0x02` respectively. Further, whenever any code inside a method of `EDWIN` changes the contents of the editor, *all* the bits (`0xffff`, symbolically `EW_CHANGE`) are set in `edwin.change`. This allows an application greater control over monitoring the extent to which changes may or may not have taken place.

"Read-only" edit boxes and the ew_readonly method

Before `EDWIN` code ever allows a change to be made, by the user, to the contents of the edit box, the value of the `PR_EDWIN_READONLY` bit in the `edwin.flags` field in property is tested. If this is set, by default a beep is emitted and the thread of execution is terminated - as can be seen from the following utility routine called frequently internal to `EDWIN` code:

```
LOCAL_C VOID CheckNotReadOnly(PR_EDWIN *self)
    {
    if (self->edwin.flags&PR_EDWIN_READONLY && p_send2(self,O_EW_READONLY))
        {
        hBeep();
        p_leave(RUN_ACTIVE_USED);
        }
    }
```

Note that there is no formal mechanism whereby the `PR_EDWIN_READONLY` bit in `edwin.flags` can be set or cleared, other than by the application directly manipulating this bit. (Care must be taken, however, to leave all other bits in this field well alone.) Note further that this bit is *cleared* at the end of `edwin_wn_init`, so that application code should only ever attempt to set it after the return of the `wn_init` message to the editor.

The `ew_readonly` method is declared in *edwin.cl* to be equal to `p_true` - in other words, it always returns `TRUE`. An application can subclass this to make some additional tests. For example, code shared between the Program Editor and Word Processor applications supplies the following replacement:

```
METHOD INT oplwin_ew_readonly(PR_OPLWIN *self)
    {
    if (IsOutlined())
        {
        KillOutline(self);
        return(FALSE);
        }
    return(TRUE);
    }
```

where the effect is to cancel any outline state before allowing any change to take place in the document, whereas other reasons for the document being read-only continue to result in TRUE being returned.

The ew_leave method

Whenever there is a run-time failure following user action that causes characters to be added to the document (eg inside the wn_key, ew_insert, or ew_paste_clip methods), a call is made to ew_leave.

For example, the code for edwin_ew_paste_clip is as follows:

```
METHOD INT edwin_ew_paste_clip(PR_EDWIN *self)
    {
    UWORD len;
    INT err;

    CheckNotReadOnly(self);
    len=p_send2(self->edwin.clip,O_EP_SENSE_LEN);
    if (len)
        {
        if ((err=p_entersend4(self->edwin.doc,O_EP_PASTE,
                              &self->edwin.cpos,self->edwin.clip))!=0)
            p_send3(self,O_EW_LEAVE,err);
        self->edwin.clen+=len;
        EdwinFwdChange(self);
        SetEdwinSelect(self,self->edwin.cpos-len,len);
        }
    return(len);
    }
```

The significance of this is so that one particular error message can be trapped - namely the “Overflow” error, E_GEN_OVER, that document objects such as EPFLAT generate when an attempt is made to insert more than the allowed maximum number of characters. Code in the ew_leave method translates this particular error into something more meaningful to the user:

```
METHOD VOID edwin_ew_leave(PR_EDWIN *self,INT err)
    {
    if (err==E_GEN_OVER)
        {
        hBeep();
        if (self->edwin.flags&PR_EDWIN_NOTIFY_OVERFLOW)
            hInfoPrint(-SYS_EDIT_NCHARS);
        err=RUN_ACTIVE_CLEANUP_NONOTIFY;
        }
    f_leave(err);
    }
```

(If desired, an application could modify this behaviour by subclassing this method.)

In English, the text for the system resource -SYS_EDIT_NCHARS is “Maximum number of characters reached”. As can be seen, this message is displayed only if the PR_EDWIN_NOTIFY_OVERFLOW flag is set in edwin.flags. By default, this is set for all editors which set either of the IN_EDWIN_VULEN_XXX flags on initialisation.

Controlling the layout and formatting

The mechanisms described in this section require various measures of knowledge of the SCRIMG and SCRLAY components of EDWIN. In fact, on the whole, these mechanisms are not methods of EDWIN itself, but involve sending messages to the SCRIMG and/or SCRLAY objects.

Amongst other things, these mechanisms allow:

- changes in which kinds of hidden symbols are displayed
- setting the width of the cursor (eg to turn it off completely - if desired)
- changes in the font used to display the text
- changes in the size of window
- changes in the margins applying to paragraphs (including the ability to disable word-wrapping)
- setting tabstops.

An introduction to SCRLAY

As mentioned already, the *character content* of an edit window is stored within a so-called "document object", which is an instance of a subclass of the FORM `EPDOC` class. The document object can, for the most part, be thought of as simply an extended buffer (possibly segmented), containing all the characters in the document, together with paragraph delimiters, tab characters, and forced line breaks stored in line.

By contrast, the document object has *no* knowledge of

- tab stop positions
- whether various special characters (spaces, tabs, carriage returns, etc) are shown or hidden
- left, right, and first-line margins applying to paragraphs
- interline and interparagraph spacing
- the "keep with next", "keep together", and "start new page" attributes of paragraphs
- the fonts to be used to display (and/or print) characters.

These attributes of an editor are supervised by a `SCRLAY` ("screen layout") sub-component.

More precisely, from the point of view of `SCRLAY`, screen layout consists of

- a linked list of paragraphs, each of which consist of
- a linked list of lines, each of which consists of
- a linked list of so-called *tboxes*.

There are three reasons why a line can be split into different tboxes:

- *physical segmentation* - the characters making up the line happen to be stored in two different physical buffers at that point (this can only ever apply, for editors, if the flag `IN_EDWIN_TEXT_SEGMENTED` is set on initialisation)
- *stylistic segmentation* - where there is a change of font, font-style, or character visibility
- *enforced segmentation* - where a limit of `236` characters per tbox is applied, to simplify visual display of the text of a tbox on the screen by means of the Window Server function `gPrintBoxText` (the value `236` is symbolically known as `WS_PRINT_BOX_TEXT_MAX_LEN`).

SCRLAY structure definitions

The screen layout is held in property of `SCRLAY` using the following structures (defined in *scrlay.cl*);

```
typedef struct que_tbox
    {
    struct scrlay_tbox *next;
    struct scrlay_tbox *prev;
    } QUE_TBOX;
```

```
typedef struct scrlay_tbox
    {
    QUE_TBOX hd;
    WORD width;     /* width of box in pixels */
    UWORD tlen;     /* number of doc positions, with mask info */
    } SCRLAY_TBOX;

typedef struct que_line
    {
    struct scrlay_line *next;
    struct scrlay_line *prev;
    } QUE_LINE;

typedef struct scrlay_line
    {
    QUE_LINE hd;
    QUE_TBOX tboxs;
    WORD indent;       x pixel position of left of 1st tbox
    UWORD len;         number of addressible content positions
    UBYTE islast;      TRUE if last line in paragraph
    UBYTE new_page;    TRUE if start of page
    } SCRLAY_LINE;

typedef struct que_para
    {
    struct scrlay_para *next;
    struct scrlay_para *prev;
    } QUE_PARA;

typedef struct scrlay_para
    {
    QUE_PARA hd;
    QUE_LINE lines;
    } SCRLAY_PARA;
```

An important efficiency measure is that SCRLAY only contains the layout information *for the visible portion of the data* - ie the data currently visible on the screen (though it turns out simpler also to maintain the data for any portion of the first visible paragraph that is off the top of the screen).

Thus SCRLAY property contains the document offset of the top of the layout structure it currently possesses. From this, it is possible to calculate the document offset of the start of any line, or the start of any tbox, within the screen layout.

As well as containing the screen layout *data*, SCRLAY contains the logic for re-calculating screen layout, according to changes in, for example, document content or paragraph styling. Finally - and this is of particular concern to users of edit boxes - SCRLAY property contains a SCRLAY_STYLE "global style definition" data structure:

```
typedef struct
    {
    UWORD fid;          font id for wserv or typeface for printer
    UWORD style;        font style (eg bold)
    UWORD height;       height of printer font in decipoints
    } SCRLAY_FONT;

typedef struct
    {
    UWORD left;         Left margin
    UWORD right;        Right margin
    UWORD indent;       Left margin of first line in para
    UWORD align;        Alignment (left, right, centre or justified)
    } SCRLAY_MARGINS;

typedef struct
    {
    UWORD line;         Space between paragraph lines
    UWORD above;        Space above paragraph
    UWORD below;        Space below paragraph
    UWORD flags;        Keep together/next and start new page
    } SCRLAY_SPACING;
```

```
typedef struct
    {
    UWORD x;           tab position
    UWORD type;        tab type (left, right, centre or repeated)
    } SCRLAY_TABSTOP;

typedef struct
    {
    UWORD ntab;         number of tabs
    SCRLAY_TABSTOP tab[SCRLAY_NTABS_MAX];
    } SCRLAY_TABS;

typedef struct
    {
    SCRLAY_MARGINS *margins;     Paragraph margins
    SCRLAY_TABS *tabs;           Paragraph tabs
    SCRLAY_SPACING *spacing;     Paragraph spacing
    } SCRLAY_PDATA;

typedef struct
    {
    UBYTE options;         Layout options
    UBYTE printer;         TRUE if printer layout
    SCRLAY_PDATA pd;       Global margins, tabs, spacing
    SCRLAY_FONT *font;     Global font id and style
    UBYTE *fwtab;          Global font table or NULL
    UWORD scrpwidth;       Width of screen in printer units
    SCRLAY_FONT *sfont;    Global screen font id and style
    } SCRLAY_STYLE;
```

Example: changing visibility of special characters

The following code shows an example of interaction with the SCRLAY_STYLE data structure inside SCRLAY. The code either hides or shows specified so-called “special characters”:

```
LOCAL_C VOID ShowSymbols(PR_EDWIN *edwin,INT symbols)
    {
    SCRLAY_STYLE style;

    p_send3(edwin->edwin.scrlay,O_SL_SENSE,&style);
    style.options=symbols;
    p_send3(edwin->edwin.scrlay,O_SL_SET,&style);
    p_send3(edwin->edwin.scrimg,O_SI_STYLE_CHANGED,SCRIMG_STCHNG_DOC);
    }
```

The basic pattern here is: sense the global layout data, make the required changes, set the new values, and then notify SCRIMG of the change (see later for further discussion of SCRIMG).

The given routine is complete in its own right, but for it to be used, the allowed values of the options field in SCRLAY_STYLE property have to be known. These can be found out from *scrlay.cl*:

```
SCRLAY_SHOW_TABS     0x01
SCRLAY_SHOW_SPACES   0x02
SCRLAY_SHOW_CRS      0x04
SCRLAY_SHOW_HYPHENS  0x08    Show optional hyphens
SCRLAY_SHOW_LFS      0x10
SCRLAY_WIDOW_ORPHAN  0x20    Set to enable widow & orphan suppression
```

Default values of SCRLAY_STYLE in edit windows

Note that many of the fields in SCRLAY_STYLE are stored by indirection. By default, the indirected data exists in suitable slots within EDWIN property - which contains a SCRLAY_MARGINS margins field and a SCRLAY_FONT font field.

The following extract from edwin_wn_init shows how this works:

```
SCRLAY_STYLE style;
SCRLAY_DOC doc;
...
style.options=SCRLAY_SHOW_TABS;
style.printer=FALSE;
style.pd.margins=(&self->edwin.margins);
style.sfont=style.font=(&self->edwin.font);
style.pd.tabs=(SCRLAY_TABS *)(&self->edwin.font.height);    /* ntab = 0 */
style.fwtab=NULL;
style.scrpwidth=0;
p_send3(self,O_EW_INIT_STYLE,&style);   /* chance for subclassers */
p_send4(self->edwin.scrlay=h_fnew(C_SCRLAY),O_SL_INIT,&doc,&style);
```

(see later in this chapter for a discussion of the SCRLAY_DOC structure).

Note in particular that, by default, no tabstops are set up. (There is a minor piece of trickery here, relying on the fact that, for screen display purposes, the font.height field is always zero.)

Changing from the default layout style

As can be seen above, one way to change from the default layout style is by a call to sl_sense followed by one to sl_set.

Another approach is to subclass the ew_init_style method of EDWIN - since a call to this is made just prior to the SCRLAY object is actually created and initialised (see the code fragment given earlier). By default, the ew_init_style method equals p_dummy, and does nothing.

Finally, all the above concerns so-called *global style* for the editor - style applying by default to all portions. However, the formatting code within SCRLAY is open to the possibility of *local variations in style*. This is discussed further in the section on document objects below.

An introduction to SCRIMG

As noted above, changing the layout style by means of a call to sl_set is not, by itself, sufficient to cause an actual change in the formatted layout of the editor. In addition, the screen image SCRIMG object has to be notified - hence the si_style_changed message in the example given.

In general, SCRIMG caters for the concepts of

- cursor position and select region
- emphasis on or off
- knowledge of the (lodger) window to be drawn to
- knowledge of how this window region may break down into a possible left gutter (“labels”) region, and a possible “line cursor” margin, as well as the main drawing area
- width of the text cursor, when displayed, as well as the style of any margin line cursor
- parameters affecting the way horizontal scrolling takes place
- the state of *background reformatting* (ie which parts of the layout are up-to-date, and which need to be re-evaluated as soon as time allows).

Additionally, SCRIMG contains the logic for the actual displaying (drawing and redrawing) of the edit window, for recalculating layout information (ie for driving the SCRLAY object), for freeing layout structures no longer required (since the visible portion of the document has altered), and for smoothly scrolling the display vertically whenever appropriate.

In fact, it may well appear that SCRIMG contains the core logic for EDWIN itself, and there is much to be said for this view. Several methods of EDWIN simply delegate responsibility to SCRIMG by passing on an appropriate message. However, it may be worth pointing out a few of the general differences between the overall EDWIN object and its SCRIMG component:

- SCRIMG (and SCRLAY and indeed *any* class in FORM) is completely independent of any of the concepts in HWIM, and can be utilised eg on the MC range of computers, where the front-line user interface library (WIMP) is significantly different from the HWIM library
- SCRIMG can be utilised independently of EDWIN, to provide so-called “edit-*like* windows”

- EDWIN may be viewed as an *organiser* of the cooperation between a SCRIMG, a SCRLAY, and an EPDOC; the wn_init method of EDWIN involves a substantial amount of "form filling", in which these sub-components are properly initialised in a suitable relationship to one another
- EDWIN contains an extensive wn_key method, which is actually one of the longest methods in the whole of the HWIM library
- EDWIN adds on significant clipboard functionality, evaluation functionality, link-paste functionality, and find and replace functionality
- amazingly (as discussed in more detail later in this chapter), SCRIMG has no *direct* knowledge *whatsoever* about the document object.

SCRIMG structure definitions

The "window" or "drawing environment" aspects of a SCRIMG object are stored in property in an SCRIMG_WIN data structure:

```
typedef struct
    {
    UWORD wid;      window ID
    P_POINT tl;     top left corner of area being drawn to
    WORD nlines;    number of text lines
    UBYTE lheight;  line height in pixels
    UBYTE lascent;  distance from top of line to text base line
    WORD width;     total width in pixels (margin,line cursor,text)
    WORD margin;    width of label margin in pixels
    WORD lcfont;    line cursor font (or zero for no line cursor)
    UBYTE cwidth;   text cursor width
    UBYTE lcstyle;  line cursor style
    UBYTE lccode;   line cursor character code
    UBYTE hscrlx;   horizontal scroll x jump
    UBYTE hscrlm;   horizontal scroll margin
    UBYTE drawplabs;   draw trailing para labels after formatting if set
    } SCRIMG_WIN;
```

Just as the SCRLAY_STYLE data held by a SCRLAY object can be sensed via an sl_sense method and set via an sl_set method, so also is there an si_sense method to sense the SCRIMG_WIN data held by a SCRIMG object, and an si_set method to set this data (see later for examples of these calls).

In fact, as the code for scrimg_si_set makes clear, the si_set method takes *two* parameters - the first being (if non-NULL) the pointer to a SCRIMG_WIN struct, and the second being (if non-NULL) the handle of the associated SCRLAY object:

```
METHOD INT scrimg_si_set(PR_SCRIMG *self,SCRIMG_WIN *win,VOID *lay)
/*
Optionally set the window and the layout object (win and lay may be NULL)
and return the width (in pixels) of the text area.
Also waits for an background formatting to die down.
*/
    {
    (PR_SCRIMG *)DatScrimg=self;
    CompleteFormatting();
    if (lay)
        self->scrimg.lay=lay;
    if (win!=NULL)
        {
        self->scrimg.gc.font=WS_FONT_BASE;
        self->scrimg.gc.style=G_STY_NORMAL;
        self->scrimg.win=*win;
        if (self->scrimg.win.lcfont)
            self->scrimg.lcwidth=gTextWidth(self->scrimg.win.lcfont,
                self->scrimg.win.lcstyle,&self->scrimg.win.lccode,1)+2;
        self->scrimg.mrwidth=self->scrimg.win.margin+self->scrimg.lcwidth;
        self->scrimg.xo=self->scrimg.mrwidth+self->scrimg.win.tl.x;
        self->scrimg.txwidth=self->scrimg.win.width-self->scrimg.mrwidth;
        }
    if (self->scrimg.crs.line>self->scrimg.win.nlines-1)
        self->scrimg.crs.line=self->scrimg.win.nlines-1;
    if (self->scrimg.lay)
        p_send3(self->scrimg.lay,O_SL_SET_LINES,self->scrimg.win.nlines);
    return(self->scrimg.txwidth);
    }
```

During initialisation, `EDWIN` takes care of setting up appropriate values for the `SCRIMG_WIN` data structure - based (as can be imagined) on the data in the `IN_EDWIN` and `IN_EDWIN_X` structures.

Example: changing the width of the text cursor

The following code shows an example of interaction with the `SCRIMG_WIN` data structure inside `SCRIMG`. The code adjusts the width that the flashing text cursor will have, when shown (eg, it could be used to set the width to zero - effectively to hide the cursor altogether):

```
LOCAL_C VOID SetCursorWidth(PR_EDWIN *ebH,INT cwidth)
    {
    SCRIMG_WIN win;

    p_send3(edwin->edwin.scrimg,O_SI_SENSE,&win);
    win.cwidth=cwidth;
    p_send4(edwin->edwin.scrimg,O_SI_SET,&win,NULL);
    }
```

Changing the font used by an editor

The `ew_set_font` method of `EDWIN` can be used to change the font used for the display. The code follows:

```
METHOD VOID edwin_ew_set_font(PR_EDWIN *self,INT font,UINT style,EDWIN_LEADING
*leading)
/*
Expected to be accompanied by call to ew_set_size
*/
    {
    SCRIMG_WIN win;
    G_FONT_INFO finfo;

    self->edwin.font.fid=font;
    self->edwin.font.style=style;
    gFontInfo(self->edwin.font.fid,self->edwin.font.style,&finfo);
    p_send3(self->edwin.scrimg,O_SI_SENSE,&win);
    win.lheight=finfo.height+leading->total;
    win.lascent=finfo.ascent+leading->top;
    p_send4(self->edwin.scrimg,O_SI_SET,&win,NULL);
    }
```

Note however that, as the comment in the code states, this call by itself will generally be insufficient to effect the font change. Additionally:

- in many cases, the size of the window region may change (quite likely when the font change has been triggered by a 'Zoom' menu command)
- a suitable request message will have to be passed in due course to SCRIMG to request it to recalculate the screen image - which will involve invalidating some or all of the formatting information maintained by SCRLAY.

See later for more information about notification and request messages to SCRIMG. The main point here is that adjusting the SCRIMG_WIN data does not, by itself, trigger a recalculation; rather, this is delayed until *all* necessary adjustments have been made, to avoid needless *repeated* re-calculations.

Note incidentally that the new font details do not have to be passed on explicitly to SCRLAY. Recall that the various SCRLAY_FONT data structures required by SCRLAY are referenced *indirectly*: the SCRLAY_STYLE data structure actually contains, by default, pointers to the edwin.font structure inside EDWIN property.

Note moreover that there is no compulsion to use the ew_set_font method, in order to change the font used by an editor. Rather, the code given above can be used as a template (in conjunction with more code to be listed shortly) for application-specific code to achieve a similar result.

Note finally that editors with *local* variations in font - ie with some portions of text being displayed in one style, and with other portions being displayed in another style - require alternative document objects to be used - as is discussed later in this chapter.

The ew_sense_size and ew_set_size methods

Code that can be used to resize an editor - for example in response to the status window changing, or in response to a 'Zoom' menu command - includes the ew_sense_size and ew_set_size methods. These are normally used as a pair, for obvious reasons:

```
METHOD VOID edwin_ew_sense_size(PR_EDWIN *self,P_EXTENT *pext)
    {
    SCRIMG_WIN win;

    p_send3(self->edwin.scrimg,O_SI_SENSE,&win);
    pext->tl=win.tl;
    pext->width=win.width;
    pext->height=win.nlines;
    }

METHOD VOID edwin_ew_set_size(PR_EDWIN *self,P_EXTENT *pext,VOID *hand,INT
method)
    {
    SCRIMG_WIN win;
    INT twid;

    p_send3(self->edwin.scrimg,O_SI_SENSE,&win);
    self->lodger.offset=win.tl=pext->tl;
    self->lodger.width=win.width=pext->width;
    win.nlines=pext->height;
    twid=p_send4(self->edwin.scrimg,O_SI_SET,&win,NULL);
    if (hand)
        p_send3(hand,method,twid);
    p_send3(self->edwin.scrimg,O_SI_STYLE_CHANGED,SCRIMG_STCHNG_DOC);
    }
```

Note that the height field in the P_EXTENT data accessed by both these methods refers to the number of *lines* in the screen window - in contrast to the values in each of the other three fields in the P_EXTENT data, which all represent numbers of *pixels*. Evidently, SCRIMG always assumes that there are a whole number of lines visible.

Next, note that the si_set method returns the number of pixels in the width of the text area of the editor, after the resize. This value may or may not be useful - see below for its use within the wn_init method of EDWIN itself.

Finally, note that the ew_set_size method supports a possible "soft" call-back - if the parameter hand is non-NULL - before making the si_style_changed request to SCRIMG to trigger a layout recalculation.

Changing the paragraph margins

The following example could be used to set the “right” margin to the arbitrary large value of `4096` - and thereby to disable word-wrap, in effect (as in the Program Editor).

Alternatively, the example could be extended to adjust the other paragraph margins used by paragraphs - ie the “left” and “first line” margins:

```
LOCAL_C VOID SetRightMargin(PR_EDWIN *edwin,UINT right)
    {
    edwin->edwin.margins.right=right;
    p_send3(edwin->edwin.scrimg,O_SI_STYLE_CHANGED,SCRIMG_STCHNG_DOC);
    }
```

Notifying SCRIMG of a change in style

The `si_style_changed` method causes `SCRIMG` to recalculate some or all of the layout in the associated `SCRLAY` object, and to redraw the screen (intelligently - ie minimising the amount of redrawing that actually is done). The cursor position and any select region are maintained, and as far as possible, the cursor is drawn on the same line number of the screen (eg the second line down from the top of the visible portion) as before.

Exactly how much work is carried out depends on the `SCRIMG_STCHNG_XXX` parameter passed (but note that this parameter is ignored on the Series 3, with notice of it being taken only on the Series 3a and on the MC):

- if the parameter is `SCRIMG_STCHNG_DOC`, the entire layout is rebuilt and redrawn (subject, as always, to only building as much layout as is required to cover the visible portion of the document)
- if the parameter is `SCRIMG_STCHNG_PARA`, layout above the top of the paragraph containing the cursor (or the top of any selected region) is *not* recalculated or redrawn, and any layout below the bottom of the paragraph containing the cursor (or the foot of any selected region) is merely scrolled vertically (if required) - this is the action appropriate when, for example, paragraph styling is applied in the word processor by a key sequence such as CONTROL-BT
- if the parameter is `SCRIMG_STCHNG_LINE`, a similar optimisation is made, appropriate this time to the application of *phrase style* (sometimes called “emphasis”) in the Word Processor - this can result in even less work being carried out, if for example the cursor is in the third or subsequent line in a paragraph.

Initialising the SCRIMG_WIN data structure

For general background interest, here is an extract from `edwin_wn_init`, containing the code that creates and initialises the `SCRIMG` subcomponent:

```
METHOD VOID edwin_wn_init(PR_EDWIN *self,IN_EDWIN *init,PR_WIN
*whand,IN_EDWIN_X *initx)
    {
    SCRIMG_WIN win;
    SCRLAY_DOC doc;
    SCRLAY_STYLE style;
    G_FONT_INFO finfo;
    EDWIN_LEADING leading;

    self->lodger.landlord=whand;
    ...
    win.lcfont=0;           /* no left cursor by default */
    if (init->flags&IN_EDWIN_LEFT_CURSOR)
        {
        win.lcfont=WS_FONT_BASE;
        win.lcstyle=self->edwin.font.style&G_STY_DOUBLE;
        win.lccode=WS_SYMBOL_MARGIN_CURSOR;
        }
    leading.total=2;  /* by default, one pixel leading at top and at bottom */
    leading.top=1;
    if (init->flags&IN_EDWIN_LEADING_SUPPLIED)
        leading=initx->leading;
    win.lheight=finfo.height+leading.total;
    win.lascent=finfo.ascent+leading.top;
    win.wid=whand->win.id;
    win.tl=initx->pos;      /* may be garbage but no harm done */
    win.nlines=(init->flags&IN_EDWIN_VISLINES_SUPPLIED? initx->vislines: 1);
    win.width=self->lodger.width;
    win.margin=0;
    win.cwidth=2;
    if (win.nlines==1)
        {
        win.hscrlx=0;
        win.hscrlm=win.width>>2;
        }
    else
        win.hscrlx=win.hscrlm=30;
    self->edwin.scrimg=h_fnew(C_SCRIMG);
    self->edwin.margins.right=p_send4(self->edwin.scrimg,O_SI_SET,
                                &win,self->edwin.scrlay)-finfo.max_width;
    if (win.nlines==1)
        self->edwin.margins.right=4096;     /* any large value will do */
    if (init->flags&IN_EDWIN_POSITION_SUPPLIED)
        {       /* else defer until following lg_set_id_pos */
        self->lodger.offset=win.tl;
        self->win.id=win.wid;
        p_send4(self->edwin.scrimg,O_SI_INIT,0,0);
        SelectAllOrCurEnd(self);
        }
    }
```

Direct interaction with document objects

A given EDWIN object interacts with up to two *document objects*: the document object where its own character content is stored, and (optionally) the clipboard document object.

The clipboard document object is usually an instance of either EPFLAT or EPSEG. See the *OLIB Reference* manual for a description of these two classes - which are each a subclass of EPROOT.

The main document object for an EDWIN has to be a subclass of the FORM EPDOC class. This is a specialised subclass of EPROOT, and like EPROOT, it supports both “flat” and “segmented” concrete subclasses, namely EPFLAT and EPSEG. Methods of EPROOT all carry over to EPDOC - although the implementation may differ, in places.

Although many of the methods of EDWIN manipulate the document objects on behalf of the application, there can be occasions where it is more appropriate for the application to interact *directly* with these objects. Afterwards, of course, the editor has to be informed that a change has taken place.

Another reason for wishing to understand document objects more deeply is in order to support local variations in style data. Yet another is in order to supply "labels" for paragraphs. All these topics are discussed in the following subsections of this chapter.

Setting text directly into the document object

Suppose for example that an application wishes to set a large amount of text into an edit window - text that can be read in stages from a file. The following steps could be taken.

First, the handle of the document object needs to be obtained. This can be read out of the `edwin.doc` property field of the `EDWIN`, or the document object may have been created separately - in which case the handle will already be known.

Next, it may be appropriate (especially in the case of a flat document object) to set the capacity of the document object in advance - if the overall size is known. The `ep_capacity` method (refer to the *OLIB Reference* manual for details) can be used to this end. (The `ep_capacity` method is left as `p_dummy` for all segmented document objects.)

Following that, repeated calls of the following sort can be made:

```
p_send5(doc,O_EP_INSERT,pos,buf,len);
```

having the effect, each time, of inserting the `len` characters at `*buf` into the document, at document offset `pos`. It would be usual to track the value of `pos` as this operation proceeds, with `len` being added to it each time `len` more characters are inserted.

Finally, code of the following sort is required:

```
LOCAL_C VOID NotifyDocChanged(PR_EDWIN *edwin)
    {
    UINT doclen;

    doclen=p_send2(edwin->edwin.doc,O_EP_SENSE_LEN);
    edwin->edwin.clen=doclen+1;
    p_send3(edwin->edwin.scrimg,O_SI_DOC_CHANGED,doclen);
    }
```

It is this last routine that stands most in need of comment here (the earlier steps, after all, only require knowledge of the `EPROOT` class). In general

- `SCRIMG` level data has to be adjusted - by means of, for example, the `si_doc_changed` method
- *and*, at the same time, `EDWIN` level data has to be adjusted - usually by direct manipulation of the relevant property fields.

There are actually two key `EDWIN` property fields in this context:

- `edwin.clen`, giving the total "character length" of the document (*including* the final paragraph delimiter)
- `edwin.cpos`, giving the cursor position, as a document offset.

For simplicity, the above routine, `NotifyDocChanged`, omits making any change in the cursor position - which may be appropriate in some cases, but it will *not* be appropriate in other cases, and will even result in program crashes in yet other cases (eg if there are fewer characters in the document after the change than before the change).

Dual variables at the EDWIN and SCRIMG levels

As can be seen, copies of, effectively, the total character length of the document are held by both `SCRIMG` and `EDWIN`. Likewise, dual copies are also kept of the cursor position. There is also a `select` field within `EDWIN` property (which is `TRUE` if there is a non-`NULL` select region, otherwise `FALSE`), which must, once again, be kept in synchronisation with the status of the select region as known to `SCRIMG`.

The reason for this duplication of data storage is to increase the speed at which various critical manoeuvres within edit boxes can be executed. However, it should be pointed out that failure to keep these dual variables appropriately in harmony is a common cause of bugs in programming edit windows.

Adjusting the cursor position

One way that the cursor position of an edit window (and, with it, the select region) can be adjusted is via the ew_set method documented earlier in this chapter.

For many purposes, however, the SCRIMG method si_move_cursor may prove more suitable. In fact, si_move_cursor is used frequently within EDWIN code (for example, within the ew_set method), often via the following utility routine:

```
LOCAL_C VOID MoveCursor(PR_EDWIN *self,INT shift,INT type)
    {
    self->edwin.select=p_send5(self->edwin.scrimg,O_SI_MOVE_CURSOR,
                                          shift,type,&self->edwin.cpos);
    }
```

Subclasses of EDWIN often contain a duplicate of this utility function - such is its use.

The meaning of the shift parameter to si_move_cursor is as follows:

- if the shift parameter is non-zero, it means to extend (or create) a select region, with the movement specified by type being applied to the moving end of the selection
- if shift is zero, it means to cancel any existing select region, and to move the cursor as specified by type.

The return value from si_move_cursor is TRUE if there is a non-NULL select region after the movement, and otherwise FALSE.

The possible meanings of type are as follows:

SCRIMG_LINEDN	move the cursor down one line
SCRIMG_LINEUP	move the cursor up one line
SCRIMG_PAGEDN	move the cursor down one page
SCRIMG_PAGEUP	move the cursor up one page
SCRIMG_LINBEG	move the cursor to the beginning of the current line
SCRIMG_LINEND	move the cursor to the end of the current line
SCRIMG_SETPOS	move the cursor to the document offset specified by the final parameter.

In all cases, the *final* position of the cursor, as a document offset, is written to the address specified by the final parameter to the si_move_cursor call.

Logical cursor movement and physical cursor movement

Most of the type values in the above table cater for so-called “physical” cursor movement - where the actual movement of the cursor is determined by reference to the current layout. (In order to work out where to position the cursor, SCRIMG interrogates the data structures maintained by the associated SCRLAY object.)

In many other cases - for example, in response to the CONTROL-LEFT key, which moves the cursor back to the next beginning of a word - the movement of the cursor is instead determined by reference to document content, and results in a so-called “logical” cursor movement. Methods of EPROOT, such as ep_scan_word, may be of use in this case. Once the required document offset is known, a call to si_move_cursor is required, specifying SCRIMG_SETPOS as the type. For this reason, subclasses of EDWIN often contain a routine such as

```
LOCAL_C VOID SetCursor(PR_EDWIN *self)
    {
    MoveCursor(self,0,SCRIMG_SETPOS);
    }
```

which, evidently, layers over the MoveCursor utility routine described earlier.

On this subject, yet another routine that may be worth duplicating is the following, whose effect is to set up a select region with given ends (this routine is called, in effect, from inside edwin_ew_set):

```
LOCAL_C VOID SetEdwinSelect(PR_EDWIN *self,UINT ancpos,INT sellen)
    {
    self->edwin.cpos=ancpos;
    MoveCursor(self,0,SCRIMG_SETPOS);
    self->edwin.cpos+=sellen;
    MoveCursor(self,TRUE,SCRIMG_SETPOS);
    }
```

Notifying SCRIMG of a change in document content

The SCRIMG class supports in all *four* different methods for reporting to it that there has been a change in the document. These methods differ primarily in how much reformatting is required - to avoid incurring unnecessary work re-evaluating layout data that cannot possibly have changed. (That is a very important consideration when the user is typing in the middle of a sizeable paragraph.)

The si_doc_reset method can be called as follows:

```
p_send5(scrimg,O_SI_DOC_RESET,doclen,cpos,line);
```

with the following effect:

- SCRIMG is notified that the document has totally changed, and now has document length doclen
- any existing select region should be discarded
- the existing layout data should be discarded
- the layout should be rebuilt so that the curspor is put at document offset pos and is displayed at line number line on the visible screen (subject to that line being reachable).

Note that line can be set to the value -1 in order to pick up the current line number (so that the cursor remains, if possible, on the same line of the screen as before).

Note again that the doclen value *includes* the final terminating paragraph delimiter in the document.

A simple example of the use of si_doc_reset is in the following utility routine called inside edwin_wn_set:

```
LOCAL_C VOID EdwinSet(PR_EDWIN *self,SE_EDWIN *set)
    {
    p_send4(self->edwin.doc,O_EP_SET_TEXT,set->buf,set->len);
    self->edwin.cpos=0;
    self->edwin.clen=set->len+1;
    p_send5(self->edwin.scrimg,O_SI_DOC_RESET,self->edwin.clen,0,0);
    }
```

(the code in edwin_wn_set goes on to set the cursor position and select region depending on the value of the initialisation IN_EDWIN_XXX flags).

Similar in effect to si_doc_reset, the si_doc_changed method differs on in that the cursor position is maintained the same as before the change was made. Accordingly, whilst a doclen parameter is still needed, this method has no cpos or line parameters. The way to call si_doc_changed in general is

```
p_send3(scrimg,O_SI_DOC_CHANGED,doclen);
```

For example, edwin_ew_replace calls si_doc_changed, indirectly, as follows (this code also illustrates use of the SCRIMG method si_sense_select):

```
LOCAL_C VOID EdwinDocChanged(PR_EDWIN *self)
    {
    self->edwin.change=EW_CHANGE;
    p_send3(self->edwin.scrimg,O_SI_DOC_CHANGED,self->edwin.clen);
    }
```

```
METHOD INT edwin_ew_replace(PR_EDWIN *self,TEXT *replace,INT backwards)
    {
    UWORD replen;
    UWORD sellen;
    UWORD pos;
    INT err;

    CheckNotReadOnly(self);
    replen=p_slen(replace);
    sellen=p_send3(self->edwin.scrimg,O_SI_GET_SELECT,&pos);
    if ((err=p_entersend5(self->edwin.doc,O_EP_INSERT,pos,replace,replen))!=0)
        p_send3(self,O_EW_LEAVE,err);
    self->edwin.cpos=pos+replen;
    if (backwards)
        self->edwin.cpos=pos;
    SetCursor(self);
    p_send4(self->edwin.doc,O_EP_DELETE,pos+replen,pos+replen+sellen);
    self->edwin.clen+=(replen-sellen);
    EdwinDocChanged(self);
    return(0);      /* confirm success */
    }
```

Notifying SCRIMG of a *local* change in document content

The `si_fwd_change` method of `SCRIMG` is called as follows:

```
p_send3(scrimg,O_SI_FWD_CHANGE,doclen);
```

This takes exactly the same parameters as `si_doc_changed`, and as in that case, the method has the effect of

- cancelling any select region
- keeping the cursor at the same document offset as before.

However, for `si_fwd_change`, `SCRIMG` makes the assumption that the layout cannot change in paragraphs earlier in the document than that containing the cursor; nor can it change in lines in the current paragraph more than one above that containing the cursor. Briefly (though, as can be appreciated, not completely accurately), only layout forward from the cursor can have changed.

For example, `edwin_ew_paste_clip` calls `si_fwd_changed`, indirectly, as follows:

```
LOCAL_C VOID EdwinFwdChange(PR_EDWIN *self)
    {
    self->edwin.change=EW_CHANGE;
    p_send3(self->edwin.scrimg,O_SI_FWD_CHANGE,self->edwin.clen);
    }

METHOD INT edwin_ew_paste_clip(PR_EDWIN *self)
    {
    UWORD len;
    INT err;

    CheckNotReadOnly(self);
    len=p_send2(self->edwin.clip,O_EP_SENSE_LEN);
    if (len)
        {
        if ((err=p_entersend4(self->edwin.doc,O_EP_PASTE,
                              &self->edwin.cpos,self->edwin.clip))!=0)
            p_send3(self,O_EW_LEAVE,err);
        self->edwin.clen+=len;
        EdwinFwdChange(self);
        SetEdwinSelect(self,self->edwin.cpos-len,len);
        }
    return(len);
    }
```

The `si_para_changed` method takes stages one step further by restricting the extent of possible layout change to the current paragraph (whereas a change notified by `si_fwd_change` can effect regions on the screen arbitrarily far below the current paragraph). More precisely, `si_para_changed` assumes that the only possible change in layout for paragraphs below the current paragraph is vertical scrolling.

Another change between si_fwd_change and si_para_changed is in the form of the parameters passed. In general, a call to si_para_changed has the form

```
SCRLAY_PLX old;
...
p_send4(scrimg,O_SI_PARA_CHANGED,keycode,&old);
```

The precise description of the method is that the paragraph containing the cursor has changed at the cursor position as a result of one of:

- a single character insertion of a content character, where keycode is either a character code (which is assumed to be printable) or zero (paragraph end) or '\t' (W_KEY_TAB) or '\n'
- a left delete, where the character code is '\b' (W_KEY_DELETE_LEFT)
- a right delete, where the character is 127 (W_KEY_DELETE_RIGHT).

The method immediately redraws the current line to reflect the input, and completes the production and the drawing of the layout as a background task. The final parameter, which is a pointer to a SCRLAY_PLX struct, has significance only when keycode is W_KEY_DELETE_LEFT. This is required in order for SCRIMG code to be able to make a safe judgement about whether the deletion has effects that extend over more than one line (the problem being that SCRIMG cannot in this calculate the old cursor position *after* being notified of the change in the document).

An example of code that sets up the appropriate SCRLAY_PLX structure, prior to calling si_para_changed, is in the following extract from edwin_wn_key, for the case when a W_KEY_DELETE_LEFT key has been received:

```
case W_KEY_DELETE_LEFT:
    CheckNotReadOnly(self);
    if (self->edwin.select)
        goto delsel;
    if (self->edwin.cpos==0)
        break;
    p_send3(self->edwin.scrimg,O_SI_DELPREP,&plx);
    self->edwin.cpos-=1;
    p_send4(self->edwin.doc,O_EP_DELETE,self->edwin.cpos,self->edwin.cpos+1);
    self->edwin.clen-=1;
    p_send4(self->edwin.scrimg,O_SI_PARA_CHANGED,keycode,&plx);
```

As can be seen, there is no need to pass a doclen parameter to si_para_changed.

When there is a change of content *and* a change in cursor position

The above few sections have touched (and hinted) at one potential problem when orchestrating notification to SCRIMG that the document has altered, whilst at the same time trying to take advantage of incremental updates in the layout information (for speed purposes).

The problem is that SCRIMG cannot be left with a cursor position that no longer exists in the document. More precisely, recall that SCRLAY ultimately views the document as consisting of a series of so-called tboxes. For example, a given tbox may refer to n characters starting at document offset doff. As mentioned earlier, these n characters will all be stored contiguously within a buffer inside the associated document object. But suppose, as a result of a change in the document, these characters are no longer stored contiguously. Then any subsequent attempt to access these characters - by reference - will fail. But this is precisely the kind of thing that SCRIMG and SCRLAY will, between them, attempt to do - so long as they believe that part of the layout structure is still valid.

Without going into any more details, the moral is clear: position the SCRIMG cursor to the *beginning* of any region where change is about to occur, before making that actual change. Various aspects of the EDWIN code given above can be seen, upon inspection, to be obeying this principle.

The SCRLAY_DOC data structure

The sl_init method, described earlier, actually requires the address of a SCRLAY_DOC data structure, as well as the address of a SCRLAY_STYLE data structure. The SCRLAY_DOC structure informs SCRLAY about some very important aspects of the associated document object:

```
typedef struct
    {
    UWORD len;          Length of doc (one greater than max position)
    VOID *content;      Object containing document content
    WORD sensechars;    Method to sense character segments
    WORD sensepdata;    Method to sense paragraph layout data
    WORD senseplabel;   Method to sense paragraph label
    WORD toparst;       Method to scan start of paragraph
    WORD enqpage;       Method to enquire for a page break
    } SCRLAY_DOC;
```

Here is how these values are filled in during `edwin_wn_init`:

```
SCRLAY_DOC doc;
...
doc.enqpage=0;
if (init->flags&IN_EDWIN_TEXT_SEGMENTED)
    {
    doc.sensechars=O_EPDOC_SENSE_CHARS;
    doc.toparst=O_EPDOC_PARA_START;
    if (init->flags&IN_EDWIN_PAGINATABLE)
        doc.enqpage=O_EPDOC_ENQ_PAGE;
    }
else
    {
    doc.sensechars=O_EPFDOC_SENSE_CHARS;
    doc.toparst=O_EPFDOC_PARA_START;
    }
if (init->flags&IN_EDWIN_DOC_SUPPLIED)
    self->edwin.doc=initx->doc;
else
    {
    self->edwin.doc=NewEdwinDoc(CAT_HWIM_FORM,C_EPFDOC,init);
    p_send4(self->edwin.doc,O_EP_SET_TEXT,&init->contents[0],p_slen(&init-
>contents[0]));
    }
doc.sensepdata=0;
doc.senseplabel=0;
doc.content=self->edwin.doc;
doc.len=self->edwin.clen=p_send2(self->edwin.doc,O_EP_SENSE_LEN)+1;
```

Subclasses of `EDWIN` (or other window objects providing edit-*like* windows) may wish to alter some of the values of these fields, to achieve affects such as paragraphs with associated labels.

The five soft method numbers in SCRLAY_DOC

It is time to point out one key design decision embodied in the relationship between `SCRIMG`, `SCRLAY`, and `EPDOC`. In fact, `SCRLAY` only ever communicates with the document object via the five soft methods (also known as "call-backs") whose numbers are contained within the `SCRLAY_DOC` data structure. And `SCRIMG` *never* communicates directly with the document object. (For example, when `SCRIMG` needs to display text on the screen - eg in response to a redraw request - it *reads* the characters it has to draw, by sending the associated `SCRLAY` object an `sl_read` message.)

This allows for diverse powerful objects to be built up, using `SCRLAY` and `SCRIMG` as components. In some cases, the associated document object will continue to be a subclass of `EPDOC` - as is always the case when `EDWIN` is involved. But there is no fundamental requirement for this to be the case. Instead, the only requirement is to provide methods for *some* of the slots in `SCRLAY_DOC`.

The SENSECHARS call-back

The protocol of the `sensechars` call-back can be seen from the following excerpt from `SCRLAY` code:

```
GLDEF_C VOID SenseChars(SCRLAY_SENSECHARS *ps,SCRLAY_FONT **pf,UBYTE **pfw)
    {
    p_send5(DatScrlay->scrlay.doc.content,DatScrlay-
>scrlay.doc.sensechars,ps,pf,pfw);
    }
```

This uses the `SCRLAY_SENSECHARS` struct which is defined as follows:

```
typedef struct
    {
    UWORD pos;          document position to sense
    WORD printer;       TRUE for printer data else screen data
    TEXT *buf;          address of character block
    WORD blen;          length of character block
    } SCRLAY_SENSECHARS;
```

One concrete realisation of the `sensechars` call-back is provided by the `epdoc_sense_chars` method of the `EPDOC` class:

```
METHOD VOID epdoc_epdoc_sense_chars(PR_EPDOC *self,SCRLAY_SENSECHARS *sense)
    {
    sense->blen=p_send5(self,O_EP_SENSE_CHARS,
                    &sense->buf,sense->pos,WS_MAX_PRINT_BOX_TEXT_LEN);
    }
```

It will be immediately apparent that this realisation of `sensechars` has ignored the final two parameters, `pf` and `pfw`. That is entirely deliberate. The reason for this is that, prior to `SCRLAY` sending the `sensechars` message, it pre-loads the `pf` and `pfw` variables to point to the corresponding *global* style fields, inside the `SCRLAY_STYLE` data structure. Accordingly, for editors in which there is no *local* variation in style, there is no need for the `sensechars` call-back to write to the passed `pf` and `pfw` variables.

For interest, here is an extract from the `sensechars` call-back provided by the Word Processor document class (known simply as `doc`):

```
METHOD INT doc_epdoc_sense_chars(PR_DOC *self,SCRLAY_SENSECHARS *pss,
                                            SCRLAY_FONT **pf,UBYTE **pfw)
    {
    TAGLIST_ITEM *pi;
    LOG_POSITION *plog;
    UINT len;
    UBYTE *pw;

    pi=doc_dc_sense_position(self,pss->pos);
    plog=(&self->doc.t->taglist.lpos);
    if (pf) /* if pss->printer is TRUE, get printer data */
        {
        if (pss->printer)
            {
            SensePrnFont(pi->par,pi->phr,&self->doc.pf);
            *pf=&self->doc.pf;
            }
        else
            {
            pw=SenseScrFont(self,pi->par,pi->phr,&self->doc.sf);
            *pf=&self->doc.sf;
            }
        }
    if (pfw) /* if pss->printer is TRUE, get printer data */
        {
        if (pss->printer)
            *pfw=SensePrnWidthTable(self,pi->par,pi->phr);
        else
            *pfw=pw;
        }
    len=pi->link-plog->offset;
    pss->blen=p_send5(self,O_EP_SENSE_CHARS,&pss->buf,pss->pos,len);
    return(pss->blen==len);
    }
```

Without going into details, some points can be noted:

- the significance of the `printer` field in the `SCRLAY_SENSECHARS` struct is that, if it is `TRUE`, printer font width tables and printer font details should be provided - otherwise data for screen display
- a test should be made on `pf` and `pfw`, in case they are `NULL`, which means that *no data* should be written to them in that case.

Structure of SCRLAY font width tables

Reference has been made above to font width tables. As can be seen, the default value of the font width table pointer `fwtab` - as set up in `edwin_wn_init` - is actually `NULL`. This clarifies that an explicit font width table is not always required. In this case, widths of pieces of text are calculated by calling the Window Server function `gTextWidth`. But for some purposes, having a font width table on the client side significantly increases the speed at which formatting can take place. Furthermore, for printing purposes, having font width tables loaded is *essential*. See the *Printing* chapter in this manual for more information about font width tables.

The TOPARST call-back

The protocol of the `toparst` call-back can be seen from the following excerpt from `SCRLAY` code:

```
LOCAL_C VOID ToParStart(UWORD *pos)
    {
    p_send3(DatScrlay->scrlay.doc.content,DatScrlay->scrlay.doc.toparst,pos);
    }
```

In contrast to the `sensechars` call-back, this is a completely straightforward routine, with only one parameter, which is a pointer to a document offset that needs to be converted to that for the start of the paragraph containing it.

The implementation of `toparst` by `EPDOC` is as follows:

```
METHOD VOID epdoc_epdoc_para_start(PR_EPDOC *self,UWORD *ppos)
/*
Convert *ppos to point to the beginning of the paragraph containing *ppos
*/
    {

p_send4(self,O_EP_SCAN_PARA,ppos,EP_SCAN_BACKWARDS|EP_SCAN_STAY|EP_SCAN_TO_BEG
IN);
    }
```

The ENQPAGE call-back

The `enqpage` call-back, if non-zero, is used by `SCRLAY` to determine whether a page-break is to occur at a given line in a paragraph. This fact is indicated by setting the `new_page` field `TRUE`, in the corresponding `SCRLAY_LINE` data structure (see earlier in this chapter for the definition of this structure).

By default, this call-back is left as zero, in edit windows, but `EDWIN` sets it to `epdoc_enq_page` if the initialisation flag `IN_EDWIN_PAGINATABLE` is set.

A full discussion of the operation of `epdoc_enq_page` requires detailed knowledge of the `EPDOC` class.

The SENSEPDATA call-back

The `sensepdata` call-back plays a role analogous to `sensechars` call-back: it caters for local variations in *paragraph* styling (whereas `sensechars` caters for local variations in *phrase* emphasis).

The protocol for the `sensepdata` call-back can be seen from the following extract from `SCRLAY` code:

```
if (DatScrlay->scrlay.doc.sensepdata)
    p_send5(DatScrlay->scrlay.doc.content,DatScrlay->scrlay.doc.sensepdata,
                        pos1,DatScrlay->scrlay.st.printer,&DatScrlay-
>scrlay.st.pd);
```

In other words, if the call-back is non-zero, the document object has to provide the `SCRLAY_PDATA` data for the paragraph containing the document offset `pos1` (and paying due respect to whether the `printer` parameter is `TRUE` or `FALSE`).

For interest, here is how the Word Processor `doc` class provides the `sensepdata` call-back:

```
METHOD VOID doc_dc_sense_pdata(PR_DOC *self,UINT pos,INT printer,SCRLAY_PDATA
*pd)
/*
Sense the margin, tab and spacing data for the current paragraph.
If printer is TRUE, return printer values.
*/
    {
    PAR_STYLE *par;

    par=doc_dc_sense_position(self,pos)->par;
    if (!printer)
        {
        SenseScrTabs(self,par,&self->doc.tabs);
        pd->margins=&par->scr_mar;
        }
    else
        {
        SensePrnTabs(self,par,&self->doc.tabs);
        pd->margins=&par->prn_mar;
        pd->spacing=&par->prn_spc;
        }
    pd->tabs=&self->doc.tabs;
    }
```

The SENSEPLABEL call-back

If non-zero, the `senseplabel` call-back is assumed to provide information about *labels* to be drawn alongside the starts of paragraphs, in a left margin area. Two examples of paragraph labels are

- field names in the Database application
- style short-codes in the Word Processor application.

The protocol of the call-back can be seen from the Word Processor implementation of the method:

```
METHOD VOID doc_dc_sense_plabel(PR_DOC *self,UINT cpos,INT
printer,SCRLAY_PLABEL **ppl)
    {
    PAR_STYLE *par;
    XWP_LAY *pxlay;

    pxlay=SenseLay(self);
    *ppl=&pxlay->label;
    par=doc_dc_sense_position(self,cpos)->par;
    pxlay->label.buf=&par->ph.sc[0];
    pxlay->label.blen=2;
    }
```

As with many of these call-backs, two of the parameters passed are

- the document offset, `cpos`, of the position of interest
- a flag, `printer`, specifying whether the information is required for screen-display purposes or for printing purposes.

The final parameter involves the `SCRLAY_PLABEL` struct, whose definition is as follows:

```
typedef struct
    {
    SCRLAY_FONT font;   font ID, height (printer only) and style
    UWORD align;        alignment
    TEXT *buf;          address of character block
    WORD blen;          length of character block
    } SCRLAY_PLABEL;
```

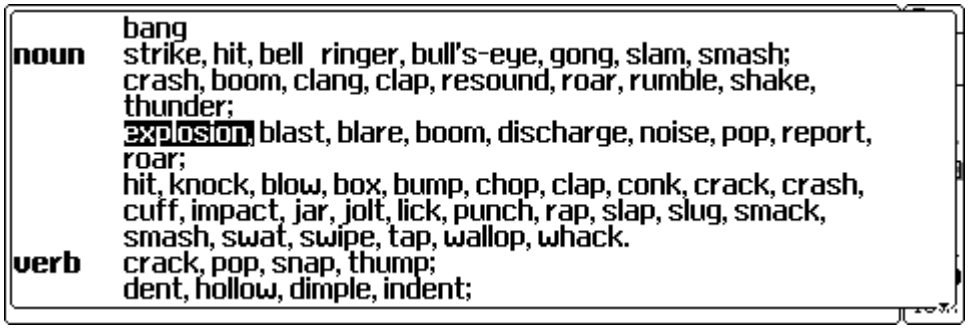
Some examples of edit-*like* windows

Admittedly, there is a formidable learning curve to gaining full familiarity with the scope and power of the `SCRLAY` and `SCRIMG` classes. In particular, despite its length, this chapter has only touched peripherally on

such topics as page breaks and special support for printing (ie using approaches *other* than the LPRINTER and XPRINTER classes).

However, it turns out in practice that many displays can be programmed more quickly and more efficiently using SCRIMG and SCRLAY, than using any alternative mechanisms. Furthermore, in many cases only a small amount of the rather detailed full interface to SCRIMG and SCRLAY needs to be appreciated.

One example is the "Synonyms" screen in the Spellchecker application. This consists of a list of words which can be navigated by standard keypresses. Thus if the word "Bang" is looked up, the screen can look like:

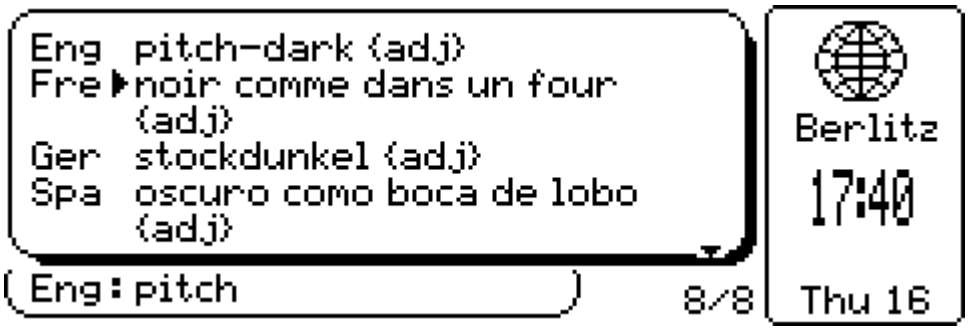


In this example, ROM code handles:

- automatically wrapping the lists of words at the edge of the window
- automatically smoothly scrolling the display vertically when needed
- navigating by word - using the ep_scan_word method
- displaying the relevant "labels" at the side of each group of words (in this example, "noun" and "verb").

ROM code even supports the notion of a "hard space", eg between "bell" and "ringer" on the second line in the above screen, to prevent paired groups of words being split over a line.

Another example worth mentioning is the main display window of the "Berlitz" five-language translator application:



General comments on creating edit-like windows

In all examples like this, a significant proportion of the work is in the initialisation. The various initialisation structs - which are quite long - have to be filled in appropriately, and in the correct order. But once the system of objects is in place, it largely runs itself.

The main extra responsibility of the programmer is to pass messages onto the SCRIMG object (and, less frequently, to the SCRLAY object) at appropriate times. Many of these messages have already been discussed in the course of this chapter, but a few remain to be covered.

The si_redraw method

The entire contents of the wn_draw method of EDWIN is to pass on a corresponding message to SCRIMG:

```
METHOD VOID edwin_wn_draw(PR_EDWIN *self)
    {
    p_send3(self->edwin.scrimg,O_SI_REDRAW,NULL);
    }
```

The final NULL parameter means to redraw the entirety of the region.

On the SERIES 3a, the final parameter can meaningfully be other than NULL (in fact, on the Series 3, the final parameter is *ignored*), in which case it should point to a P_RECT structure indicating the portion of the region that needs to be redrawn.

The si_emphasize method

The entire contents of the wn_emphasise method of EDWIN is, again, to pass on a corresponding message to SCRIMG:

```
METHOD VOID edwin_wn_emphasise(PR_EDWIN *self,INT flag)
    {
    p_send3(self->edwin.scrimg,O_SI_EMPHASIZE,flag);
    }
```

Any application that makes direct use of SCRIMG (ie without the benefit of an intermediate EDWIN object) would have to possess a corresponding message.

The si_pan method

The si_pan method provides a means for a window display to be scrolled *horizontally*, even though it is not displaying any visible cursor.

An example of this behaviour is when the user presses RIGHT or LEFT when viewing the “Found” screen in the “Wrap off” state of the Database application.

Code calling the si_pan method will look like

```
p_send4(scrimg,O_SI_PAN,func,par);
```

The si_pan method actually combines three different functions, depending on the value of func:

- if func is SCRIMG_PAN_SETNOPAN, the nopan property value of SCRIMG is set to par (which should be either TRUE or FALSE)
- if func is SCRIMG_PAN_DELTA, the effect is to scroll the display horizontally by par pixels (where par can be either positive or negative)
- if func is SCRIMG_PAN_ABS, the effect is the scroll such that par is at the left of the view.

Any scrolling is contrained to reasonable limits. For example, the display will not scroll past the right hand end of the longest visible line.

The purpose of the nopan mode is to control whether, any time the view is scrolled vertically, it also tries to scroll horizontally (ie to “pan”) in order to keep the cursor position visible.

CHAPTER 13

PRINTING

This chapter describes the basics of access to the WDR print system from HWIM applications.

See the chapter *WDR Printing* in the *Additional System Information* manual for background information about the scope of the SIBO WDR print system.

Although partial access to the WDR print system is available to Hwif programmers, full access is only possible via object oriented techniques, such as are explained in this chapter.

As far as HWIM applications that wish to print are concerned, three different levels of approach can be identified, in increasing order of sophistication (and difficulty):

1. applications create and use a subclass of the HWIM `LPRINTER` class, and `REPLACE` only the one method `lpr_sense_text`
2. applications create and use a subclass of `LPRINTER`, and `REPLACE` *other* methods, such as `lpr_read`
3. applications avoid using `LPRINTER`, and instead interface more directly with classes such as `PAGES` in the FORM library (the interaction with `PAGES` is one of the things that `LPRINTER` handles automatically, for less demanding applications).

For many applications, the first of these three levels is perfectly sufficient, and in this case, only a limited acquaintance with the material in this chapter is required.

Print preview

On the Series 3a, it is relatively straightforward for applications using the WDR print system to provide 'Print preview' menu commands, in addition to 'Print' commands.

The basic step involved, in most cases, is to subclass the `XPRINTER` class, rather to subclass `LPRINTER`. (The `XPRINTER` class is in the XADD library and is not available on the Series 3.) The interface to `XPRINTER` is very similar to that of `LPRINTER` (`XPRINTER` is actually a subclass of `LPRINTER`). The vast majority of application-level print preview code is *exactly* the same as the application code that implements print.

The basic model of WDR printing

Regardless of whether an application implements print preview as well as print, and regardless of the level of sophistication the application brings to printing, the first basic requirement, in use of the WDR print system, is an understanding of the `WDR_PRINT` struct. This is defined as follows in *prdrv.cl*:

```
typedef struct
    {
    WORD flags;          WDR_PRINT_XXX
    WORD typf;           Typeface number for WDR_PRINT_FONT
    WORD fheight;        Font height for WDR_PRINT_FONT (twips)
    WORD style;          Font style for WDR_PRINT_FONT
    WORD down;           Line down for WDR_PRINT_LINE
    WORD indent;         Line indent for WDR_PRINT_LINE
    WORD height;         Line height for WDR_PRINT_LINE
    WORD right;          Right movement for WDR_PRINT_RIGHT
    TEXT *buf;           Text to print for WDR_PRINT_TEXT
    UWORD blen;          Length of data at buf for WDR_PRINT_TEXT
    } WDR_PRINT;
```

(Hwif programmers will recognise this as the same as the `H_PRINT` struct.)

Essentially, the basic model of WDR printing can be summarised as follows:

- the application creates and initialises a suitable top-level printing object
- in due course, system code repeatedly calls the application back, passing, each time, the address of a `WDR_PRINT` struct
- the application fills in various fields in this struct, each time, to describe the next “print element”, and lets the flow of execution return into system code
- eventually, printing comes to an end - either because it has finished normally, or because the user has cancelled it, or because an error condition has arisen
- the top-level printing object gets destroyed.

Thus printing can be viewed as a sequence of “print elements”, each described by a `WDR_PRINT` struct. The possible *types* of print elements can be seen from the list of possible bits that the application can set, each time, in the `flags` field of the `WDR_PRINT` struct:

`WDR_PRINT_PAGE`	this print element is to go on a new page (ie a page break will be forced, if not already on a new page)
`WDR_PRINT_LINE`	this print element is to go on a new line (ie a line break will be forced, if not already at the start of a line)
`WDR_PRINT_FONT`	this print element is to be in a specified font and font style
`WDR_PRINT_RIGHT`	this print element consists, in part, of moving the print position right by a specified amount
`WDR_PRINT_TEXT`	this print element consists, in part, of text to be printed (in cases where `WDR_PRINT_RIGHT` is set as well, the movement right takes place before the printing of the text)
`WDR_PRINT_END`	this print element terminates the print process
`WDR_PRINT_KEEP`	the line containing this print element is to be kept, if possible, on the same page as the following print element
`WDR_PRINT_IDLE`	(for advanced use only - see later).

Calculation of page breaks

Note that applications in general have no need to calculate the positions of page breaks, as they print. System code, inside the `PAGES` class in FORM, automatically calculates when page break instructions should be emitted to the printer. This calculation takes all the following into account:

- Any explicit instructions from the application, on account of print elements with the bits `WDR_PRINT_PAGE` and/or `WDR_PRINT_KEEP` set
- The length of the page, as supplied by the user in the ‘Print setup’ dialog for the application
- The height of each line, and (in effect) the *spacing* between each line, as supplied in the `height` and `down` fields in print elements with `WDR_PRINT_LINE` set.

System code (in PAGES) also takes care of printing relevant *headers* and *footers* at the top and bottom of each page - with the *contents* of the header and footer being supplied by the user in the 'Print setup' dialog for the application.

Calculation of line breaks

The situation as regards calculation of *line* breaks is rather different from that of the calculation of *page* breaks. Whereas the PAGES class handles "vertical" formatting (such as pagination), it is up to other software (eg that in LPRINTER) to handle "horizontal" formatting (such as word-wrap).

Thus the only time PAGES repositions the print position back to the beginning of a line is when WDR_PRINT_LINE is explicitly set in the flags field of a print element.

Moreover, PAGES never prints any text, when moving horizontally, other than that supplied to it in a print element, which contrasts (again) with the case when moving vertically - since headers and footers are added in automatically, by PAGES, whenever they are required.

Printer units

Values set in the down, indent, height, and right fields in a print element must be supplied in *printer units*. These units vary from printer to printer (corresponding to the different degrees of resolution these printers support). Now it might at first seem that having to deal with printer units conflicts with the general philosophy of WDR printing, in which application code doesn't have to worry about which printer has been selected by the user. However

- there are system services, such as the lpr_sense_buf_width method of LPRINTER, to calculate the width, in current printer units, of a specified buffer of text
- there are other system services, such as the wdr_twips_to_xy method of the WDR class (see later for more details of this class), to assist with the transformation of printer independent units (ie "twips", where there are 1440 twips to an inch) into printer dependent ones
- simple uses of LPRINTER can avoid the need to specify units altogether, since they can accept the default values of the down, indent, height, and right fields in any print element.

Note that the reason why word wrap and pagination calculations must take place in printer units is to avoid unnecessary rounding errors in the use of any other units.

The difference between INDENT and RIGHT, and between DOWN and HEIGHT

Superficially, the indent and right fields in the WDR_PRINT struct may seem to serve the same purpose. However, the indent field is only relevant when the WDR_PRINT_LINE bit is set in flags, and the right field is only relevant when the WDR_PRINT_RIGHT bit is set in flags:

- when WDR_PRINT_LINE is set, the current print position is moved back to the beginning of the line, then moved down by the height of one line, *and then moved right* by indent
- when WDR_PRINT_RIGHT is set, the current print position is simply moved right by right (if a print element contains *both* a WDR_PRINT_LINE and a WDR_PRINT_RIGHT, the former is executed before the latter).

The indent field is intended for use, as its name implies, as the "left indent" (or "first line indent") parameter of a paragraph of formatted text, whereas the right field is intended for use as the spacing between two columns of data, or (more simply) as the representation of tab characters.

Note that the action of any right movement *adds on to the current horizontal offset* of the print position - as established by earlier indents, rights, and text printing on that line.

The difference between the down and height fields may also require some attention. Both fields are relevant only when WDR_PRINT_LINE is set. Normally, the two values are simply added together, and the sum is taken as the amount, in printer units, to advance the print position vertically, before starting to print the given line of text. However, code in PAGES *automatically zeros the supplied value* of down if the given line of text would be the first on a page.

For example, applications that wish to implement some measure of inter-paragraph spacing *additional to inter-line spacing* (or which wish, more generally, to separate related groups of printed lines by extra spacing in between these groups) should place this additional spacing in the down field. This will ensure that extra spacing is not printed, unnecessarily, *at the tops of pages*. (Blank spacing at page tops, of this

form, *would* be seen, on occasion, if extra spacing between groups of lines were implemented, at the application level, simply as blank lines.)

Margins and page size

There is no need for application code to attempt to set the `indent` value in such a way as to *include* the margin at the left hand edge of the page (as set, by the user, inside the 'Print setup' dialog for the application). Likewise, nor is there any need to try to set the first `height` or `down` values to try to include the margin at the top of the page.

Rather, these adjustments are automatically made by system code. In other words, zero is a suitable *default* value for both `indent` and `down`.

As mentioned before, there is no need, either, for applications to determine the length of the printing area of the page (ie the total page length, minus the sum of the top and bottom margins), since pagination is handled by code in `PAGES`. Nor, in general, does an application that uses `LPRINTER` need to investigate the *width* of the printing area of the page, since `LPRINTER` handles all (simple) word-wrap automatically.

The PRINTER class and storage of the 'Print setup' dialog settings

Whenever a standard 'Print setup' dialog is invoked in an application, the values displayed and edited in this dialog are stored within the property of an instance of the `PRINTER` class. The `PRINTER` class is in the FORM library and, in addition to allowing these data values to be stored, this class also contains general supervisory logic to do with printing. (The `PRINTER` class also encapsulates knowledge of read/write access to the `P$?` printing environment variables, described in the *WDR Printing* chapter of the *Additional System Information* manual.)

Some applications may choose to create a `PRINTER` instance as part of their standard initialisation. Other applications only create one such instance when they are about to:

- print (or print preview)
- run the print setup dialog.

HWIM code relies on the handle of any instance of `PRINTER` being written to the `wserv.printer` field within the property of `w_ws`. On the Series 3a, this handle is also written to the `appman.spare1` field of `w_am`, where it can be accessed by FORM code (eg low level print preview code).

However, HWIM applications have no need to write, themselves, the handle of the `PRINTER` object into either of these places. This is handled by system code, mainly inside the `ws_ens_print_context` method of `WSERV`. The contents of this method (which has no parameters) are, effectively,

```
METHOD VOID wserv_ws_ens_print_context(PR_WSERV *self)
    {
    if (!self->wserv.printer)
        self->wserv.printer=f_newsend(CAT_HWIM_FORM,C_PRINTER,O_PR_INIT);
    }
```

Note that HWIM code automatically sends `w_ws` a `ws_ens_print_context` message in the following cases:

- at the beginning of the `ws_edit_print_context` method of `WSERV` - the method which applications invoke (see below) to run the standard 'Print setup' dialog suite
- at the beginning of the `lpr_init` method of `LPRINTER` - the method which applications invoke (see below) in order to print or to print preview.

Consequently, most applications never need to call `ws_ens_print_context` directly. The exception is if an application wishes, for various reasons, to maintain an instance of `PRINTER` at other stages of its lifetime. For example, an application may store some of the print context (the property of the `PRINTER` class) to file, and restore that context when the file is opened again. In this case, typical action would be for the application to call `ws_ens_print_context` itself, during its file loading code, and then to set various parts of the in-memory print context, using methods of the `PRINTER` class, passing data from file as parameters. (An example of code to achieve this is given near the end of this chapter.)

Changing font or font style while printing

One of the aspects of the "print context" is the so-called "default printing font". Code in LPRINTER sets this font (and its associated style) by default into the typf, fheight, and style fields in the WDR_PRINT struct.

Here, note that a font, for printing purposes, is identified by a combination of its "typeface number" (the typf value) and its "font height" (the fheight value). For background information about fonts and typefaces, see the chapter *WDR Printing* in the *Additional System Information* manual.

Most applications will find it convenient not to adjust the typf or fheight values in any print element. However, applications may well wish to augment the style field, on occasion. For example, certain parts of the printed output might benefit from being emphasised in bold or in italics. In this case, note that the allowed values in the style field are bit combinations of WDR_STYLE_NORMAL, WDR_STYLE_UNDERLINE, WDR_STYLE_BOLD, WDR_STYLE_ITALIC, WDR_STYLE_SUPER, and WDR_STYLE_SUB - where all the meanings are obvious from their names.

Note also that variant style values should be in general be *orr*ed into the default ones supplied by system code, rather than completely overwriting these values. This is to preserve the freedom of the user, in the 'Print setup' dialog, to choose a "default printing font" with style other than normal.

Applications that support multiple fonts (or multiple font heights) should be aware of the set of fonts (and font heights) supported by the current printer model (eg Epson, HPIII, Postscript), as selected by the user in the 'Print setup' dialog. Thus if the application provides the user with a "palette" of possible "character styles" (or whatever), the set of fonts from which the user is allowed to choose ought to match the set known to the loaded printer model. There are methods of the WDR class that allow access to this information (see later in this chapter for more details).

The text referenced in a print element

There are a couple of potential errors that applications should watch out for, regarding the lifetime of the buffer referenced by the buf and blen fields of a print element. One is somewhat obvious; the other less so (but it only applies to applications that make explicit use of the WDR_PRINT_KEEP flag).

In the first place, this buffer must, obviously enough, *continue in existence after return of the application callback routine* (eg the lpr_sense_text method) that sets up the print element. Thus it would be a grave error to assemble a print buffer on the stack of this routine.

More subtly, consider a line of printed output with the WDR_PRINT_KEEP flag set. Clearly, system code cannot print this line straightaway, on the current page of output, since it must first process at least one additional line, to see whether the first line should instead be held over for a new page (eg if there is no room to place this line *and the following one* on the current page). For this reason, any print buffer associated with a line with WDR_PRINT_KEEP set must have a more permanent existence, and can only be re-used with due care.

Limitations with the WDR_PRINT_KEEP flag

Note incidentally that the WDR_PRINT_KEEP flag only has effect if it is set in the first print element in a line of output. Setting this flag for print elements other than those which also contain WDR_PRINT_LINE has no effect.

Applications which require more complicated pagination scenarios will need to perform a "pagination pass" prior to actually printing (this happens, in different ways, in both the Series 3 Word Processor and the Series 3 Spreadsheet) and will thereafter only set the WDR_PRINT_PAGE flag, and never the WDR_PRINT_KEEP flag.

The need to specify font and style for each line

One more potential problem should be pointed out (though this will be of concern only to programmers who interact more deeply with the PAGES class). On the face of things, if the font never changes, throughout the course of printing, there oughtn't to be any need to specify the typf, fheight, and fstyle values anew for every print element. However, it must be borne in mind that two print elements which the application regards as being contiguous may, in the actual course of printing, end up on two different pages, with a footer and a header separating them. Given that the user is, in general, free to specify the print fonts of the header and the footer to *differ* from that used in the body of the page, it can now be appreciated why, as far as PAGES is concerned, it is important for application-generated print elements to have their print font specified explicitly each time.

Programmers needn't worry about any particular inefficiency this might entail. Font change instructions are sent to the printer itself only when there is an actual change in between two adjacent print elements.

Use of WDR_PRINT_IDLE

A small proportion of applications may find themselves unable to generate print elements immediately (ie sufficiently quickly) in response to a system callback. For example, fetching and assembling the data to print may involve one or more application-specific active objects. This kind of application may wish to make use of print elements with `WDR_PRINT_IDLE` set.

If `PAGES` finds that `WDR_PRINT_IDLE` is set in any print element, the rest of that print element is disregarded *and the print subsystem is effectively placed into a state of suspension*. Thus no more application callbacks will take place until the print system is restarted by an explicit application call.

The way that the application lets system code know to restart the print subsystem is to send the `PAGES` object an `ao_queue` message. Applications using `LPRINTER` can find the handle of the `PAGES` object in the `lprinter.pages` field in its property.

Using LPRINTER for standard printing purposes

The printing requirements of most applications can be met by them defining an application-specific subclass of `LPRINTER`, in which they

- `REPLACE` the method `lpr_sense_text` (which is `DEFER`red at the `LPRINTER` level)
- declare sufficient property to be able to keep track of the progress of printing.

Then when the user invokes the ‘Print’ menu command in the application:

- the application may choose to present a dialog (a so-called ‘Print details’ dialog), to collect parameters describing which portions of its data are to be printed, and (possibly) with what options
- next, the application creates its subclass of `LPRINTER` and sends it an `lpr_init` message
- this message does not return until printing has completed (internally, a call to `am_start` is made); accordingly, the next lines of code can destroy the `LPRINTER` subclass object
- however, in the meantime, the `lpr_sense_text` message will have been called repeatedly.

Thus typical command manager code, in the print method, might look like

```
RunPrintDetailsDialog();
hDestroy(f_newsend(CAT_APP_APP,C_APP_LPRINTER,O_LPR_INIT));
```

Note: this code fragment assumes that the results of the “print details” dialog is available to the `LPRINTER` subclass via global data. An alternative approach would of course be to `REPLACE` the `lpr_init` method too, so that the command manager code would become something along the lines of

```
PRINT_DETAILS_RBUF rbuf;

RunPrintDetailsDialog(&rbuf);
hDestroy(f_newsend(CAT_APP_APP,C_APP_LPRINTER,O_LPR_INIT,&rbuf));
```

with the contents of the `lpr_init` method looking like

```
METHOD VOID app_lprinter_lpr_init(PR_APP_LPRINTER *self,PRINT_DETAILS_RBUF
*prbuf)
    {
    self->app_lprinter.prbuf=prbuf;
    p_supersend2(self,O_LPR_INIT);          /* calls am_start */
    }
```

The syntax of the LPR_SENSE_TEXT callback

(Hwif programmers may note that this is the same as the syntax of the `PrintLine` callback function, where `PrintLine` is the function name passed as the parameter to `hPrint`.)

This method passes the single parameter `pr`, which is a pointer to a `WDR_PRINT` data structure. Application code adjusts one or more or the fields in `*pr`, as described above.

The return value from `lpr_sense_text` has the following significance:

- a zero return value means that all print elements have already been defined, and that the print subsystem should now terminate
- a non-zero return value means that the application has not finished printing yet, and expects additional calls to `lpr_sense_text` to be made in due course.

Note that `LPRINTER` fills in many of the parts of `*pr` before sending `self` the `lpr_sense_text` message:

- the `flags` field is set to `WDR_PRINT_FONT|WDR_PRINT_LINE|WDR_PRINT_TEXT`
- the `typf`, `fheight`, `style`, and `height` fields are all set to values reflecting the default font and default font style (which the user can specify in the 'Print setup' dialog suite
- the `down` and `indent` fields are both set to zero.

Thus all that needs to be filled in, in most cases, are the `buf` and `blen` fields. On some occasions, the value of `flags` may need to be adjusted; likewise the `down`, `indent`, and `right` fields. Finally, a value will need to be provided for `right` in any case when `WDR_PRINT_RIGHT` is set in `flags`. (See below for some examples.)

LPRINTER and word-wrap

As mentioned above, `LPRINTER` takes care of word-wrap automatically on behalf of applications: if the text at `pr->buf` is too wide to fit on the space remaining to the end of the line, `LPRINTER` splits it up into two or more sections, and thus creates two or more print elements (without any intervention being required to this end from the application). For the print elements formed in this way:

- for new lines, the `indent` value is taken from the `subsqind` field within `LPRINTER` property (subclasses can write directly to this field within application code - see below for an example)
- unless the user has disabled widows/orphans control in the 'Print setup' dialog suite, the flag `WDR_PRINT_KEEP` is orred into all but the last of these print elements.

Note that the word-wrap calculation presupposes use of the default font and the default style throughout. If an application prints using multiple fonts or font-styles, it may need to subclass the `lpr_read` method of `LPRINTER`, instead of the `lpr_sense_text` method - see later for more details.

Working out widths

If, unusually (eg to support printing in columns), an application needs to know more about relative widths of various strings of text, the `lpr_sense_buf_width` method may be used. This takes two parameters:

- a `TEXT*` pointer to the buffer of text
- an `INT` giving the length of the buffer.

This method returns the width of the buffer, in current printer units.

Note that this method cannot be accessed until inside one of the `lpr_sense_text` callbacks - since it relies on fields in `LPRINTER` property that are not set up until just before the first such callback.

Note also that this calculation presupposes use of the default font and the default style. See later for how to calculate widths of text in other fonts and styles.

Another width value that may be of interest to applications is stored in the `width` property field of `LPRINTER`. This is the width, in printer units, of the printing area of the page (ie the complete page width less its left and right margins). Again, this value is not available prior to the first callback to `lpr_sense_text`.

Launching the print setup dialog suite

If (as normal) an application that supports print also supports a 'Print setup' menu command, the code that is required in the command manager method for print setup is usually just the single line:

```
p_send2(w_ws,O_WS_EDIT_PRINT_CONTEXT);
```

For interest's sake, the entire contents of the `ws_edit_print_context` method of `WSERV` is given here:

```
METHOD VOID wserv_ws_edit_print_context(PR_WSERV *self)
    {
    p_send2(self,O_WS_ENS_PRINT_CONTEXT);
    runHwimDialog(-SYS_PRINT_CONTROL_DL,C_PRNCTRL,NULL);
    }
```

Examples of use of LPRINTER

This section presents two related examples of the use of `LPRINTER`. Both are complete applications in their own right. Each example allows the user to specify a start date, and a number of days, and then prints the list of dates specified, in expanded form, eg as follows:

```
Monday, 18th October 1993
Tuesday, 19th October 1993
Wednesday, 20th October 1993
...
```

The user is also able, in each case, to specify a “separation” between months, which can be either “No gap”, “Half a line”, or “Complete line”. Also in each case, the user is given an opportunity, before the printing actually takes place, to alter the values in the ‘Print setup’ dialog.

The second example builds on the first, adding additional formatting to produce printed output in two columns, with effect as in:

```
Monday,       18th October 1993
Tuesday,      19th October 1993
Wednesday,    20th October 1993
...
```

On installation of the OOP component of the SDK, the source code for the second example is copied into a *\sibosdk\tprint* directory.

Framework of the example applications

Unusually, these applications have no menu bar or client window. This is possible because, throughout their lifetimes, a dialog is always presented:

- first, the ‘Print details’ dialog of the application
- next, the ‘Print setup’ dialog
- finally, the HWIM ‘Printing’ dialog, that keeps track of the progress of printing.

In implementation terms, all this happens inside the `ws_dyn_init` initialisation callback to the `WSERV` subclass of the application. Code never returns from here, since, after the printing is complete, a call to `p_exit` is made.

The entire contents of the category file, *demo.cat*, of the first example, are as follows:

```
IMAGE demo

EXTERNAL olib
EXTERNAL hwim

INCLUDE hwimman.g
INCLUDE dlgbox.g
INCLUDE lprinter.g
INCLUDE time.g

CLASS dewserv wserv
    {
    REPLACE ws_dyn_init
    }
```

```
CLASS dedlg dlgbox
    {
    REPLACE dl_key
    TYPES
        {
        typedef struct
            {
            UWORD dayno;
            UWORD ndays;
            UWORD gap;
            } RBUF_LP;
        }
    }

CLASS delp lprinter
    {
    REPLACE lpr_init
    REPLACE lpr_sense_text
    PROPERTY 1
        {
        VOID *time;
        RBUF_LP rbuf;
        TEXT buf[LN_TIME_DATE_STR];
        }
    }
```

From this, it can be seen that there are four key objects in the application:

- the `DEWSERV` object, which provides the `ws_dyn_init` method
- the `DEDLG` object, which supervises the 'Print details ' dialog of the application
- the `DELP` object, which is a subclass of `LPRINTER`
- a `TIME` object, which is used to obtain the textual representations of the dates printed.

The 'Print details' dialog

The following resources define the 'Print details' dialog of the application (in *demo.rss*):

```
RESOURCE MENU delp_gap_chlist
    {
    items =
        {
        CHOICE_ITEM { str="No gap";},
        CHOICE_ITEM { str="Half a line";},
        CHOICE_ITEM { str="Complete line";}
        };
    }
```

```
RESOURCE DIALOG delp_dlg
    {
    title="Print list of days";
    flags=DLGBOX_NOTIFY_ENTER|DLGBOX_RBUF_FILLED;
    controls=
        {
        CONTROL
            {
            class=C_DTEDIT;
            prompt="Start date";
            info=DTEDIT { flags=IN_DTEDIT_DDMMYYYY;};
            },
        CONTROL
            {
            class=C_NCEDIT;
            prompt="Number of days to print";
            info=NCEDIT
                {
                low=10;
                current=20;
                high=200;
                };
            },
        CONTROL
            {
            class=C_CHLIST;
            prompt="Space between months";
            info=CHLIST { rid=delp_gap_chlist; };
            }
        };
    }
```

When run, the dialog looks like

Code in the `dl_key` method of the `DLGBOX` subclass senses the values in the fields in this dialog into an `RBUF_LP` result buffer:

```
#include <demo.g>
#include <hwim.h>

#pragma METHOD_CALL

METHOD INT dedlg_dl_key(PR_DLGBOX *self)
    {
    RBUF_LP *prbuf;

    prbuf=self->dlgbox.rbuf;
    prbuf->dayno=hDlgSenseDtedit(1);
    prbuf->ndays=hDlgSenseNcedit(2);
    prbuf->gap=hDlgSenseChlist(3);
    return(WN_KEY_CHANGED);
    }
```

Startup code and WS_DYN_INIT code

This dialog code is invoked (indirectly) from the code in the `ws_dyn_init` method of the application:

```
#include <demo.g>
#include <demo.rsg>
#include <hwim.h>
```

```
LOCAL_C INT LaunchDialog(INT class,INT resid,VOID *rbuf)
    {
    DL_DATA dld;

    dld.id=resid;
    dld.rbuf=rbuf;
    dld.pdlg=NULL;
    return hLaunchDial(CAT_DEMO_DEMO,class,&dld);
    }

#pragma METHOD_CALL

METHOD VOID dewserv_ws_dyn_init(PR_DEWSERV *self)
    {
    RBUF_LP rbuf;

    if (LaunchDialog(C_DEDLG,DELP_DLG,&rbuf))
        {
        p_send2(self,O_WS_EDIT_PRINT_CONTEXT);
        hDestroy(f_newsend(CAT_DEMO_DEMO,C_DELP,O_LPR_INIT,&rbuf));
        }
    p_exit(0);
    }
```

In turn, this code is invoked (again indirectly) from that in `main`:

```
#include <demo.g>

GLDEF_C VOID main(VOID)
    {
    IN_HWIMMAN app;
    IN_WSERV wserv;

    p_linklib(0);
    app.flags=FLG_APPMAN_RSCFILE|FLG_APPMAN_SRSCFILE|FLG_APPMAN_CLEAN;
    app.wserv_cat=p_getlibh(CAT_DEMO_DEMO);
    app.wserv_class=C_DEWSERV;
    wserv.com_cat=p_getlibh(CAT_DEMO_HWIM);
    wserv.com_class=C_COMMAN;
    p_send4(p_new(CAT_DEMO_HWIM,C_HWIMMAN),O_AM_INIT,&app,&wserv);
   }
```

Note incidentally that the resource file for this application only contains three resources in all: the two listed above, which define the 'Print setup' dialog, plus a "dummy" `WSERV_INFO` resource defined as follows:

```
#include <hwim.rh>
#include <hwim.rg>

RESOURCE WSERV_INFO demo_accs
    {
    menbar_id=0;
    first_com=0;
    accel={'x'};
    }
```

(Although the application has no true menu bar - since it never returns to a "base state" without a dialog being present - it is a requirement of *all* HWIM applications that the first resource in their own resource file is a `WSERV_INFO`. This resource is loaded by system startup code, and must be present.)

The LPRINTER initialisation code (first example)

The `LPRINTER` subclass initialisation code has two parts:

- that which takes place *outside* of any `lpr_sense_text` callback
- that which takes place *during the first callback* to `lpr_sense_text` (by which time, all required system initialisation will be complete).

These take place, respectively, in the routines `delp_lpr_init` and `InitTimeObject`, with the former being called from code in the `ws_dyn_init` method (see above), and the latter from code in `lpr_sense_text` (see below):

```
#include <demo.g>

#define TIME_FMT_FLAGS
(PR_TIME_MONTH_NAME|PR_TIME_SUFFIX_NAME|PR_TIME_DAY_NAME)

LOCAL_C VOID InitTimeObject(PR_DELP *self)
    {
    P_DAYSEC ds;
    SE_TIME_FORMAT fmt;

    ds.day=self->delp.rbuf.dayno;
    ds.sec=0;
    self->delp.time=f_new(CAT_DEMO_OLIB,C_TIME);
    p_send4(self->delp.time,O_TO_SET,SET_TIME_DAYSEC,&ds);
    fmt.flags=TIME_FMT_FLAGS;
    p_send4(self->delp.time,O_TO_SET_FORMAT,&fmt,TIME_FMT_FLAGS);
    }

#pragma METHOD_CALL

METHOD VOID delp_lpr_init(PR_DELP *self,RBUF_LP *prbuf)
    {
    self->delp.rbuf=(*prbuf);
    p_supersend2(self,O_LPR_INIT);
    }
```

The LPR_SENSE_TEXT method (first example)

```
METHOD INT delp_lpr_sense_text(PR_DELP *self,WDR_PRINT *pr)
    {
    P_DATE dt;

    if (!self->delp.time)
        InitTimeObject(self);
    else if (!self->delp.rbuf.ndays)
        return(FALSE);
    else
        p_send4(self->delp.time,O_TO_ADD_DAYS,1,0);
    p_send4(self->delp.time,O_TO_SENSE,SENSE_TIME_DATESTR,&self->delp.buf[0]);
    pr->buf=(&self->delp.buf[0]);
    pr->blen=p_slen(pr->buf);
    if (self->delp.rbuf.gap)
        {
        p_send4(self->delp.time,O_TO_SENSE,SENSE_TIME_DATE,&dt);
        if (!dt.day)
            {
            pr->down=pr->height;
            if (self->delp.rbuf.gap==1)
                pr->down>>=1;
            }
        }
    self->delp.rbuf.ndays--;
    return(TRUE);
    }
```

Second example: additional initialisation code

To modify the example so that the printing takes place in columns, all that has to change is

- the definition of `DELP` in the category file
- the `DELP` code (of course)
- one new `STRING` resource is added to the resource file.

The additional initialisation determines the values of some new property fields in `DELP`. The new class definition of `DELP` becomes

```
CLASS delp lprinter
    {
    REPLACE lpr_init
    REPLACE lpr_sense_text
    CONSTANTS
        {
        DELP_STATE_COL1     0
        DELP_STATE_COL2     1
        DELP_STATE_COL3     2
        }
    PROPERTY 1
        {
        VOID *time;
        UWORD colwid;
        UWORD right;
        UWORD state;
        RBUF_LP rbuf;
        TEXT dname[E_MAX_DAY_NAME];
        TEXT buf[LN_TIME_DATE_STR];
        }
    }
```

The new initialisation routine `FindWidthFirstColumn`, called from within the first visit to `delp_lpr_sense_text` (just after `InitTimeObject` is called) calculates the required width, in printer units, for the first column of printed output. This works as follows:

```
LOCAL_C VOID FindWidthFirstColumn(PR_DELP *self)
    {
    INT i;
    TEXT buf[E_MAX_DAY_NAME];
    UWORD wid;

    for (i=0; i<7; i++)
        {
        p_nmday(&buf[0],i);
        wid=SenseBufWidth(self,&buf[0]);
        if (wid>self->delp.colwid)
            self->delp.colwid=wid;
        }
    self->delp.colwid+=SenseBufWidth(self,"  ");    /* two spaces */
    if (self->delp.colwid>(3*self->lprinter.width/4))
        {
        hInfoPrint(DELP_PAPER_NARROW);
        p_sleep(20);
        p_leave(RUN_ACTIVE_CLEANUP_NONOTIFY);
        }
    self->lprinter.subsqind=self->delp.colwid;
    }
```

The routine `SenseBufWidth`, used within `FindWidthFirstColumn`, is just a convenience layer over the `lpr_sense_buf_width` method of `LPRINTER`:

```
LOCAL_C UINT SenseBufWidth(PR_DELP *self,TEXT *pb)
    {
    return(p_send4(self,O_LPR_SENSE_BUF_WIDTH,pb,p_slen(pb)));
    }
```

Note one more feature of the code in `FindWidthFirstColumn` - the check that the width required for the first column does not leave too little room left for the remaining column. A more professional application may wish to take more sophisticated action in this kind of situation. (For example, the Series 3 Spreadsheet application performs a pre-printing "pagination" run that determines how many pages *horizontally* are required, to print a given range of spreadsheet columns.)

The three states in printing a two-column display

In this example, where there are only two columns per date to be printed, the process of printing each date string is split into *three* separate visits to the `lpr_sense_text` method, resulting in three different print elements:

- the first visit prints the text of the first column: since this starts a new line, the default `LPRINTER` flags of `WDR_PRINT_LINE` and `WDR_PRINT_TEXT` are left as they are; however, the width of the text *actually* printed (which will not exceed that of the column itself) is determined, by making

another call to SenseBufWidth, in order that the amount by which the print position should be moved right, in the next print element, can be known

- the second visit consists just of moving the print position forward from the end of the text printed in the first column to where the text in the second column should start; the default flags WDR_PRINT_LINE and WDR_PRINT_TEXT must be *removed* in this case, and the flag WDR_PRINT_RIGHT set instead
- finally, the third visit consists of printing the text for the second column; in this case, the flag WDR_PRINT_LINE has to be removed, but WDR_PRINT_TEXT remains.

(The way this mechanism would be extended to printing more than two columns should be clear enough.)

The LPRINTER subclass keeps track of what it has to do next, in any particular callback, by using the state property field, which rotates around the three possible DELP_STATE_XXX values.

The code for the entire lpr_sense_text method therefore becomes

```
METHOD INT delp_lpr_sense_text(PR_DELP *self,WDR_PRINT *pr)
    {
    P_DATE dt;
    P_DAYSEC ds;

    if (!self->delp.time)
        {
        InitTimeObject(self);
        FindWidthFirstColumn(self);
        }
    else if (!self->delp.rbuf.ndays)
        return(FALSE);
    switch (self->delp.state++)
        {
    case DELP_STATE_COL1:
        if (self->delp.rbuf.gap)
            {
            p_send4(self->delp.time,O_TO_SENSE,SENSE_TIME_DATE,&dt);
            if (!dt.day)
                {
                pr->down=pr->height;
                if (self->delp.rbuf.gap==1)
                    pr->down>>=1;
                }
            }
        p_send4(self->delp.time,O_TO_SENSE,SENSE_TIME_DAYSEC,&ds);
        p_nmday(&self->delp.dname[0],P_WEEK(ds.day));
        self->delp.right=self->delp.colwid-SenseBufWidth(self,&self-
>delp.dname[0]);
        pr->buf=(&self->delp.dname[0]);
        break;
    case DELP_STATE_COL2:
        pr->flags&=(~(WDR_PRINT_LINE|WDR_PRINT_TEXT));
        pr->flags|=WDR_PRINT_RIGHT;
        pr->right=self->delp.right;
        return(TRUE);
    case DELP_STATE_COL3:
        p_send4(self->delp.time,O_TO_SENSE,SENSE_TIME_DATESTR,&self-
>delp.buf[0]);
        pr->buf=(&self->delp.buf[0]);
        pr->flags&=(~WDR_PRINT_LINE);
        pr->indent=self->delp.colwid;
        self->delp.rbuf.ndays--;
        p_send4(self->delp.time,O_TO_ADD_DAYS,1,0);
        self->delp.state=DELP_STATE_COL1;
        }
    pr->blen=p_slen(pr->buf);
    return(TRUE);
    }
```

More details about printing in columns with LPRINTER

As mentioned above, LPRINTER checks all text passing through it, to ensure that it fits within the appropriate part of the page width. In the context of the above example of printing in columns, the text in the

first column is guaranteed always to fit (otherwise the check on the relative width of the first column and the page width would have failed). However, it is certainly possible that the text in the *second* column might end up being wrapped. It was with an eye on this possibility that the following two lines of code were included in `DELP` code above:

- `self->lprinter.subsqind=self->delp.colwid;` (in `FindWidthFirstColumn`)
- `pr->indent=self->delp.colwid;` (in the `DELP_STATE_COL3` case in the `lpr_sense_text` method).

The first of these lines of code tells the superclass `LPRINTER` code that, if text passed to it ever does need to be word-wrapped, the second line should be positioned with a "subsequent indent" (`subsqind` value) as specified. (Were this left at its default value of zero, any new line required would start off in the space intended to be reserved for the left hand column.)

The second of these lines of code is somewhat more obscure. The point is that `LPRINTER` code does not track the horizontal offset where the print position has reached. For this reason, in any word-wrap calculation, the only sensible width value for `LPRINTER` to wrap text within is equal to the expression

```
self->lprinter.width-pr->indent
```

Now in principle `LPRINTER` could be used to create printed displays of the "hanging bullet" variety, in which text in the "left-hand column" always fits on just one line, whereas text in the "right-hand column" is typically longer, and can flow over several lines. However, a couple of words of caution are appropriate here:

- support for this kind of "hanging bullet" printing by `LPRINTER` is only available, effectively, in the Series 3a version of the ROM code
- in any case, no "widows and orphans" control takes place, since, as mentioned earlier, the `WDR_PRINT_KEEP` flag is effective only when present *in the first print element* of a line.

In short, programmers requiring this kind of printed output would be best advised to dig deeper into the WDR print system possibilities - such as are featured in the remaining sections of this chapter.

Advanced uses of LPRINTER - and beyond

In this section, more features of the code in `LPRINTER` are gradually introduced - up to the point where it should be possible to see how to print *without any use* of `LPRINTER` - should that be desired.

This section can be skipped altogether by programmers whose needs are met by the information in the preceding sections.

The LPR_READ method of LPRINTER

Consider the code in the `lpr_read` method of `LPRINTER`:

```
METHOD VOID lprinter_lpr_read(PR_LPRINTER *self,INT x,WDR_PRINT *pr)
    {
    if (!self->lprinter.wdr)
        ReadWdrData(self);
    if (self->lprinter.defer.blen)
        {
        *pr=self->lprinter.defer;
        pr->indent=self->lprinter.subsqind;
        }
    else
        {
        pr->flags=WDR_PRINT_FONT|WDR_PRINT_LINE|WDR_PRINT_TEXT;
        pr->typf=self->lprinter.f.fid;
        pr->fheight=self->lprinter.f.height;
        pr->style=self->lprinter.f.style;
        pr->height=self->lprinter.lheight;
        pr->down=0;
        pr->indent=0;
        if (!p_send3(self,O_LPR_SENSE_TEXT,pr))
            {
            pr->flags=WDR_PRINT_END;
            return;
            }
        }
    self->lprinter.defer=(*pr);
    if (pr->indent>=self->lprinter.width)
        pr->indent=0;       /* guard against incorrect use */
    pr->blen=fit_line(self->lprinter.wtab,pr->buf,pr->blen,
                      self->lprinter.width-pr->indent,GetTextPrinterWidth);
    if (pr->blen==self->lprinter.defer.blen)
        self->lprinter.defer.blen=0;
    else
        {
        self->lprinter.defer.blen-=pr->blen;
        self->lprinter.defer.buf+=pr->blen;
        self->lprinter.defer.flags|=WDR_PRINT_LINE;   /* line added for Series
3a version */
        if (!((PRINTER_PARAMS *)
                p_send2(w_ws->wserv.printer,O_PR_GET_PARAMS))->d.wo_control)
            pr->flags|=WDR_PRINT_KEEP;
        }
    }
```

As can be seen, this is the routine which contains the call to `lpr_sense_text` - the deferred method that all `LPRINTER` subclasses have to provide. The role of `lpr_read` is to provide a layer in between print elements defined by application code and print elements as required by the `PAGES` class (see later for further discussion of `PAGES` - for the moment it suffices to explain that the `lpr_read` message is sent to `LPRINTER` by `PAGES`). The `lpr_read` method sets up defaults that are suitable for most `LPRINTER` subclasses, and also performs word-wrapping (via the call to the `fit_line` routine) that is, again, suitable for most subclasses.

However, in some cases, this behaviour is no longer so helpful, and has to be changed - in which case the `lpr_read` method will, itself, have to be `REPLACE`d.

LPRINTER property introduced

To make sense of the code in `lprinter_lpr_read`, reference will have to be made to various features of the full class definition of `LPRINTER`:

```
CLASS  lprinter root
    {
    REPLACE destroy
    ADD lpr_init
    ADD lpr_read
    ADD lpr_sense_buf_width
    DEFER lpr_sense_text
    PROPERTY
        {
        VOID *pages;            Copy for reference only
        VOID *wdr;              Copy for reference only
        SCRLAY_FONT f;          The default font
        WDR_PRINT defer;        In case previous data was too wide
        UBYTE *wtab;            Width table
        UWORD lheight;          Line height in printer units
        UWORD width;            Width of region to print to
        UWORD subsqind;         Indent for subsequent lines (if wrapped)
        }
    }
```

The `wdr` field in property is used, amongst other ways, as a flag as to whether this is the first call to the method (in the current print session). If it is (in which case the `wdr` property field is `NULL`), the initialisation routine `ReadWdrData` is called. See later for more details of the initialisation of the `LPRINTER` data.

Apart from testing the value of `wdr`, the code in `lprinter_lpr_read` splits into two cases:

- if `defer.blen` is non-zero, it means that the last text buffer specified by the subclass did not all fit on the space remaining on the previous printed line, and that some of this buffer remains to be processed; in this case, this call to `lpr_read` will *not* result in a call to `lpr_sense_text`
- otherwise, there is no “deferred” text remaining, and the subclass has to be asked, by means of sending `self` an `lpr_sense_text` message, to supply another `WDR_PRINT` “print element”.

In *either* case, the next buffer of text is submitted to `fit_line`, to see whether *this* will all fit within the remaining page width.

If an application subclasses `lpr_read`, it may choose, however, to do one or more of the following:

- skip the word-wrap calculation altogether
- calculate “clipping”, whereby text that is too wide to fit within an allocated region of the page does not get word-wrapped, but rather is printed in a truncated form (with trailing or leading characters being omitted, as appropriate to the application)
- make the word-wrap calculation (or a clipping calculation) *more sophisticated*, by allowing *variable fonts* or *variable font styles*.

The default word-wrapping algorithm

The routine `fit_line` is hardly sophisticated:

(as well as being used within `LPRINTER` code, `fit_line` is also utilised within the `ws_wrap_para` method of `WSERV`)

```
GLDEF_C INT fit_line(VOID *wrap,TEXT *buf,INT blen,INT wid,INT (*f)(VOID
*,TEXT *,INT))
/*
Returns the number of characters, out of the buffer passed,
that can form a line of up to the preset width.
*/
    {
    INT nchars;
    INT thischar;
    INT safe;
    INT seenbreak;

    nchars=0;
    seenbreak=FALSE;
    while (blen--)
        {
        thischar=(*buf);
        wid-=(*f)(wrap,buf++,1);
        nchars++;
        if (thischar==' ')
            {
            safe=nchars;
            seenbreak=TRUE;
            }
        else if (wid<0)     /* right margin burst */
            return(seenbreak? safe: nchars-1);
        }
    return(nchars);
    }
```

As is evident, the only word delimiter recognised by `fit_line` is the space character.

Now a fundamental limitation of `fit_line` is that all the characters in the buffer passed to it are assumed to belong to the same font (and to have the same style). Thus the callback function `f` (which is `GetTextPrinterWidth` in the case when `fit_line` is called from `lprinter_lpr_read`) disregards the offset of characters within the buffer (for example, the width of the first 'I' in "ITALIC" would always be taken as the same as that of the second 'I' in that word).

Subclasses that `REPLACE lpr_read` may in fact consider any of the following three kinds of modifications to this method of reckoning widths:

- text in one column may have a different font and/or style than that in another column
- the text *within one column* (or, in the case when there is only one column across the whole page, the text *within one line*) may itself contain more than one font and/or style
- special characters, such as tabs, may need additional consideration.

The third of these possibilities is beyond the scope of this chapter. However, for each of the first two possibilities, it is clear that a generalisation of the `GetTextPrinterWidth` routine will be required.

Calculating widths of text with variable font

The `lpr_sense_buf_width` method of `LPRINTER` simply consists of a call to this same routine, `GetTextPrinterWidth`:

```
METHOD INT lprinter_lpr_sense_buf_width(PR_LPRINTER *self,TEXT *buf,INT len)
    {
    return(GetTextPrinterWidth(self->lprinter.wtab,buf,len));
    }
```

In turn, the code for `GetTextPrinterWidth` is as follows (note: this code is also the same as the `wdr_sense_width` method of the `WDR` class):

```
LOCAL_C INT GetTextPrinterWidth(VOID *pwid,TEXT *buf,INT len)
/*
Return the printed width (in printer units) of buf, len
*/
    {
    INT sum;

    if (*(UBYTE *)pwid==0)
        return(len*(*((UBYTE *)pwid+1)));
    for (sum=0;len--;)
        sum+=*((UBYTE *)pwid+*buf++);
    return(sum);
    }
```

From this code, the structure of WDR printer font width tables can be seen. These fall into two types:

- for monospaced fonts, the table is only two bytes long, with the first byte being `0` and the second being the common width, in printer units, of any of the characters in the font
- for proportional fonts, the table is 256 bytes long, with the width of character `'a'` (which has ASCII value 65), say, being the 65th byte in the table, and so on.

Where printer font width tables come from

The field `wtab` in `LPRINTER` property is the address of the font width table of the default font for the currently selected printer model. The value of `wtab` is filled in by the following line of code in the `ReadWdrData` routine called during the first visit to `lprinter_lpr_read`:

```
self->lprinter.wtab=(UBYTE *)p_send5(self->lprinter.wdr,
            O_WDR_GET_WIDTH_TABLE,self->lprinter.f.fid,
            self->lprinter.f.height,self->lprinter.f.style);
```

As can be seen, printer font width tables are accessed by means of the `wdr_get_width_table` method of the `WDR` class. Briefly, the `WDR` class encapsulates the logic of reading printer driver *.wdr* files. (See the *WDR Printing* chapter in the *Additional System Information* manual for more information about *.wdr* files.)

There are three parameters to the `wdr_get_width_table` method:

- the typeface number of the required font
- the height identifier of that font
- the particular font style required.

In the case when this method is called by `LPRINTER`, these three parameters are taken from details of the default font associated with the current printer, and therefore are bound to be valid. But even if values are passed that are not directly known to the current printer model - for example, there could be an enquiry concerning an unknown typeface - the method will follow the standard `WDR` customs of “font mapping” and “font substitution”: details will be supplied of the “nearest” font, font height, and style actually known to the printer model.

LPRINTER initialisation - phase one

As has been mentioned, the initialisation of an `LPRINTER` instance takes place in two stages:

- some takes place within the `LPR_INIT` method
- another portion of the initialisation cannot, however, proceed until other parts of the WDR print system have been put into a full state of readiness, and has to wait until the first print element is requested, by `PAGES`, before taking place.

This section looks at the first of these two stages.

The code in the `LPR_INIT` method of `LPRINTER` is as follows:

```
METHOD VOID lprinter_lpr_init(PR_LPRINTER *self)
/*
Returns only when printing is complete
*/
    {
    INT preview;
    PRINTER_PARAMS *par;
    RBUF_PRINTING rbuf;

    preview=self->lprinter.subsqind;
    self->lprinter.subsqind=FALSE;
    p_send2(w_ws,O_WS_ENS_PRINT_CONTEXT);
    par=(PRINTER_PARAMS *)p_send2(w_ws->wserv.printer,O_PR_GET_PARAMS);
    self->lprinter.width=par->p.pg.body.width;  /* in twips at the moment */
    self->lprinter.f=par->d.f;
    if (preview)
        return;
    rbuf.calls.hread=self;
    rbuf.calls.mread=O_LPR_READ;
    rbuf.ppages=(&self->lprinter.pages); /* to take pages handle when known */
    runHwimDialog(-SYS_PRINTING_DIALOG,C_PRINTING,&rbuf);
    }
```

In fact, this is the Series 3a version of the code; the Series 3 version omits the lines

```
    preview=self->lprinter.subsqind;
    self->lprinter.subsqind=FALSE;
```

and

```
    if (preview)
        return;
```

These are connected with support for print preview (which is not available on the Series 3) and will be discussed in more detail later in this chapter.

Apart from these lines, this code

- sends `w_ws` a `ws_ens_print_context` message, for reasons discussed earlier in this chapter (essentially, to ensure that an instance of the `PRINTER` class has been created and initialised)
- senses the width of the printing portion of the page, from the `PRINTER` instance, and also the `SCRLAY_FONT` structure describing the default font for the current printer model
- begins to set up an appropriate initialisation struct for the `PAGES` active object
- launches an Hwim standard dialog (the `PRINTING` dialog), whose `dl_dyn_init` method will, in turn, set more code in motion that will create and initialise `PAGES`
- since the `PRINTING` dialog calls `am_start`, the `LPR_INIT` method does not return until this dialog has completed.

A brief description of the PAGES active object class

Clearly there has to be *some* active object involved with printing - in order that the lengthy process of printing can be interleaved with calls to the `ao_run` method of `w_ws` (for example, to service redraws, or handle keypresses); `PAGES` is this active object. `PAGES` in fact lies at the heart of the WDR printing subsystem; whereas it is possible for an application (for example, the Series 3 Word Processor and Database application) to perform WDR printing without making any use of `LPRINTER`, it is not possible to avoid using `PAGES`.

Whilst a full description of `PAGES` is beyond the scope of this manual, various points should be noted.

For its initialisation, `PAGES` requires to be initialised with, among other items, two object handles and two method numbers that collectively make up the `PAGES_CALLS` struct:

```
typedef struct
    {
    VOID *hread;        Callback handle for reading print elements
    WORD mread;         Callback method for reading print elements
    VOID *hdone;        Callback handle for %done & completion
    WORD mdone;         Callback method for %done & completion
    } PAGES_CALLS;
```

Whenever PAGES requires another print element, for the body area of a page, it sends hread an mread message. In the case when printing is started by LPRINTER, hread is set to the handle of the LPRINTER object itself, whereas mread is set equal to O_LPR_READ - as can be seen in the above code from lprinter_lpr_init.

Whenever PAGES has an event to report *to the user*, it sends hdone an mdone message. These events include: the end of a page, the end of a printing session, and an error condition. In fact, when printing is started by LPRINTER, hdone is set, in due course, to the handle of the PRINTING dialog, and mdone is set equal to O_PRINTING_PRINTING. For interest, part of the code of the HWIM PRINTING class follows:

```
LOCAL_C UINT GetFirstPageToPrint(VOID)
    {
    return(((PR_PRINTER *)(w_ws->wserv.printer))->printer.p.p.pgbeg);
    }

LOCAL_C VOID SetPageNumDisplay(INT num,INT resid)
    {
    TEXT buf[40];

    hAtos(&buf[0],resid,num);
    hDlgSetText(1,&buf[0]);
    }

LOCAL_C VOID SetPageNumDisplayCheck(PR_PRINTING *self,INT num)
    {
    SetPageNumDisplay(num,(self->win.flags&PR_WIN_WILL_SKIP &&
              GetFirstPageToPrint()>num)? -SYS_SKIPPING_PAGE: -SYS_PAGE_IS);
    }

#ifdef JPIC
#pragma METHOD_CALL
#endif

...

METHOD VOID printing_printing_done(PR_PRINTING *self,PAGES_DONE *d)
    {
    switch (d->event)
        {
    case PAGES_DONE_PAGE:
        SetPageNumDisplayCheck(self,d->page);
        break;
    case PAGES_DONE_ERROR:
    case PAGES_DONE_END:
        self->printing.pages=NULL;
        p_send2(self,O_DESTROY);
        break;
        }
    }
```

Note that, after sending hdone a PAGES_DONE_END or PAGES_DONE_ERROR event, PAGES destroys itself. This is the reason why the above PRINTING code nulls its copy of the handle of PAGES (otherwise, the destroy method of PRINTING would attempt to destroy PAGES a second time, in the standard procedure of automatic destruction of component objects). Finally note that a destroy message can also be sent to PRINTING as a result of the user pressing the ESC key - this happens automatically on account of standard DLGBOX level code; in this case, it *is* appropriate for PAGES to be destroyed as a consequence of PRINTING being destroyed; this is how printing terminates in response to the user's “Abandon” request.

More about the interface to and from PAGES

Two cases where application code might send messages *directly* to PAGES are as follows:

- if the application makes use of the flag WDR_PRINT_IDLE when it prints - in which case, as explained earlier in this chapter, it needs to send the PAGES object an ao_queue when it has determined what the next print element is
- if the application needs to destroy the PAGES object (but this will only arise if the application takes over the code that is, by default, handled by the destroy method of the PRINTING class).

In most cases, however, application code will never have any reason to send a message directly to the PAGES object. Instead, the parts of the interface to and from PAGES that are more likely to need to be understood are:

- how to create and initialise the PAGES object
- the nature of the mread and the mdone callbacks from PAGES.

A PAGES object is actually created by sending a message to the PRINTER object. There are in fact three very similar methods, all of which create and initialise PAGES in one way or another:

- the pr_print method creates and initialises PAGES in *printing* mode
- the pr_paginate method creates and initialises PAGES in *paginating* mode
- the pr_preview method (Series 3a only) creates and initialises PAGES in *print preview* mode.

In each case, there is one additional parameter - the address of a PAGES_CALLS struct, as described earlier. In each case, the method returns the handle of the PAGES object created.

Thus code called from inside the dl_dyn_init method of the HWIM PRINTING class (which is itself called from code inside the lpr_init method of the LPRINTER class) contains the following:

```
METHOD VOID *printing_printing_do_print(PR_PRINTING *self,PAGES_CALLS *pcalls)
    {
    return((VOID *)p_send3(w_ws->wserv.printer,O_PR_PRINT,pcalls));
    }

METHOD VOID printing_dl_dyn_init(PR_PRINTING *self)
    {
    RBUF_PRINTING *rbuf;

    ...
    rbuf=self->dlgbox.rbuf;
    rbuf->calls.hdone=self;
    rbuf->calls.mdone=O_PRINTING_DONE;
    self->printing.pages=(VOID *)p_send3(self,O_PRINTING_DO_PRINT,&rbuf-
>calls);
    if (rbuf->ppages)
        *rbuf->ppages=self->printing.pages;
    }
```

Note that the *paginating* mode of PAGES is provided specially for use by window objects based on the FORM SCRLAY and SCRIMG classes - such as the main windows of the Word Processor and Database applications. The fact that the pagination logic is so similar to printing logic needn't particularly concern most users of PAGES. The only potentially significant point concerns the parameters passed back to each mread callback. As can be seen from the listing given above for the lpr_read method of LPRINTER, a somewhat mysterious second parameter (called x in that listing) is passed. The actual significance of this is in whether or not PAGES is being run in paginating mode. Various optimisations can be made in this case within SCRLAY callback code. (To be completely accurate, the callback in this case is to the PRNLAY *subclass* of SCRLAY.) However, as is clear, this parameter is totally ignored when the callback is made, instead, to LPRINTER code.

A brief description of the WDR class

The WDR class shares with PAGES the feature of lying at the heart of the WDR printing subsystem. Whilst PAGES is the active object class that actually *drives* page formatting and printing, WDR supports various query functions concerning the current printer model. For example, the wdr_get_width_table method has already been mentioned.

Another useful service of the WDR class is that of converting a null-terminated sequence of UWORD values from twips into current printer units. The method involved here is wdr_twips_to_xy. An example of the use of this method is in the ReadWdrData function already mentioned:

```
LOCAL_C VOID ReadWdrData(PR_LPRINTER *self)
    {
    UWORD *lx[2];
    UWORD *ly[2];

    self->lprinter.lheight=self->lprinter.f.height;
    ly[0]=&self->lprinter.lheight;
    ly[1]=NULL;
    lx[0]=&self->lprinter.width;
    lx[1]=NULL;
    self->lprinter.wdr=((PR_PAGES *)self->lprinter.pages)->pages.in.wdr;
    p_send4(self->lprinter.wdr,O_WDR_TWIPS_TO_XY,&lx[0],&ly[0]);
    self->lprinter.wtab=(UBYTE *)p_send5(self->lprinter.wdr,
                O_WDR_GET_WIDTH_TABLE,self->lprinter.f.fid,
                self->lprinter.f.height,self->lprinter.f.style);
    }
```

In order to find out the number of typefaces supported by the current printer model, code such as the following can be used:

```
nt=((WDR_MODEL *)p_send2(wdr,O_WDR_SENSE_MODEL))->num_typefaces;
```

where the definition of the `WDR_MODEL` struct is (refer also to the the *WDR Printing* chapter in the *Additional System Information* manual)

```
typedef struct
    {
    UWORD minx;              minimum delta x (in twips, unless MINX_IS_DPI
flag set)
    UWORD miny;              minimum delta y in twips
    UWORD skipx;             amount printer auto indents
    UWORD skipy;             amount printer auto feeds
    UWORD flags;             orientation
    UWORD num_typefaces;     number of typefaces supported by model
    WDR_TYPEFACE *typeface[1]; list of typeface rids/pointers to typeface
    } WDR_MODEL;
```

For any given typeface, referred to by *index number* (0, 1, ...), the corresponding *name* and *typeface number*, among other things, can be found out by sending the `WDR` object a `wdr_typeface` message, which takes the index number as a parameter, and which returns a pointer to a `wdr_typeface` struct:

```
typedef struct
    {
    TEXT name[WDR_FONT_NAME_LEN];      Typeface name
    UWORD typeface;          RTF/Word compatible typeface
    UWORD type;              WDR_TYPF_XXX
    WORD trans_rid;          rid of translates record
    UWORD num_heights;       Number of different typeface heights
    WDR_FONT font[1];        List of different heights
    } WDR_TYPEFACE;
```

Another approach to finding a typeface with a given typeface number is to send the `WDR` object a `wdr_search_typeface` message, which has the following definition:

```
METHOD INT wdr_wdr_search_typeface(PR_WDR *self,INT typf,WORD
*pix,WDR_TYPEFACE **ppt)
/*
Write the index and address of the typeface struct with RTF/Word
typeface number typf to *pix and *ppt respectively and return TRUE
if a matching typeface was found.
Otherwise return FALSE and write a recommended typeface
or -1 if no such typeface number exists in the current model.
Either pix or pt may be NULL if that part of the return is not required.
*/
```

As will be appreciated, this method contains the “font mapping/ substitution” logic of the `WDR` class.

Finally, for a given typeface, the way to determine the range of *heights* available (and also the recommended way of *matching* a desired font height) can be seen from the following code - which is actually an extract from the standard Hwim “Font selector” dialog code:

```
LOCAL_C VOID ResetSizes(PR_FONTSEL *self,INT typfix)
    {
    UWORD n,i;
    WORD height;
    TEXT buf[12];

    p_send2(self->fontsel.sizes,O_VA_RESET);
    n=((WDR_TYPEFACE *)p_send3(self->fontsel.wdr,O_WDR_TYPEFACE,typfix))-
>num_heights;
    for (i=0;i<n;i++)
        {
        buf[p_itob(&buf[0],p_send4(self-
>fontsel.wdr,O_WDR_FONT_HEIGHT,typfix,i)/20)]=0;
        p_send3(self->fontsel.sizes,O_VA_APPEND,&buf[0]);
        }
    height=self->fontsel.pf->height;
    hDlgSetChlist(2,p_send4(self-
>fontsel.wdr,O_WDR_SEARCH_HEIGHT,typfix,&height));
    }
```

(Note that the values returned by the `wdr_font_height` method are in *twips*: hence the multiplication by 20, to convert into points prior to presenting the values in a dialog for inspection by users.)

Creating and destroying WDR objects

Evidently, `LPRINTER` code - and any other WDR printing code - relies on a suitable `WDR` object having been created and initialised. The `PAGES` class in particular contains a property field given the handle of a `WDR` object, and the `ao_init` method of `PAGES` requires to be passed this handle as part of its initialisation data.

There is also a slot for the handle of a `WDR` object within `PRINTER` property. For this reason, `PRINTER` code called inside the `pr_print` method (also called by the `pr_paginate` and `pr_preview` methods) contains the following lines:

```
    if (!self->printer.wdr)
        p_send2(self,O_PR_OPEN_WDR);
```

The `pr_open_wdr` method of `PRINTER` is as follows:

```
METHOD PR_WDR *printer_pr_open_wdr(PR_PRINTER *self)
    {
    INT model;
    TEXT name[P_FNAMESIZE];

    model=p_send3(self,O_PR_SENSE_MODEL,&name[0]);
    printer_pr_close_wdr(self);
    self->printer.wdr=f_newsend(CAT_FORM_FORM,C_WDR,O_WDR_INIT,&name[0],model)
    return(self->printer.wdr);
    }
```

In turn, the `pr_sense_model` method obtains the appropriate printer model

- by preference, from data set in `PRINTER` property by a prior call to `pr_set_model` (see the end of this chapter for an example of code sending a `pr_set_model` message)
- otherwise, from the `P$M` print model environment variable
- failing that, from the hard-wired default of `ROM::BJ.WDR`.

Note that this mechanism leaves open the possibility of the application creating and initialising a `WDR` object, for its own purposes, well before the user selects any print menu command. Evidently, the way to do this is to

- ensure that the `PRINTER` object (handle at `w_ws->wserv.printer`) has been created and initialised
- send this object a `pr_open_wdr` message.

Incidentally, it is perfectly possible for there to be more than one `WDR` object in existence, at the same time in the same application (although only one of them can have its handle written into `PRINTER` property). Thus when dialogs inside the HWIM “Print setup” dialog suite are operational, a “scratch” `WDR` is used at various

times, in order to enquire details of *.wdr* printer model files other than the one to which the application is currently "logged".

On the other hand, WDR objects should be destroyed as soon as they are no longer required. (For example, a considerable amount of memory may be tied up by all the font width tables that may have been loaded.)

To achieve this, simply send PRINTER a pr_close_wdr message. Thus the destroy method of LPRINTER is as follows:

```
METHOD VOID lprinter_destroy(PR_LPRINTER *self)
    {
    if (w_ws->wserv.printer)
        p_send2(w_ws->wserv.printer,O_PR_CLOSE_WDR);
    p_supersend2(self,O_DESTROY);
    }
```

(Note that the test of whether w_ws->wserv.printer is non-null is necessary because the lpr_init method of LPRINTER could fail prior to the completion of the call to ws_ens_print_context.)

In turn, the code in the pr_close_wdr method of PRINTER is, naturally enough,

```
GLDEF_C VOID DestroyRef(PR_ROOT **ref)
    {
    if (*ref)
        {
        p_send2(*ref,O_DESTROY);
        *ref=NULL;
        }
    }

METHOD VOID printer_pr_close_wdr(PR_PRINTER *self)
    {
    DestroyRef(&self->printer.wdr);
    }
```

Using XPRINTER for print preview

Just as there are various levels at which the subject of printing can be approached, so also are there various levels at which the subject of print preview can be approached. However,

- like printing, the requirements of most applications for print previewing can be met very simply, by means of creating and using a subclass of LPRINTER - except that this time a subclass of XPRINTER is required (XPRINTER itself being a subclass of LPRINTER)
- in these cases, what makes application coding particularly easy is the fact that *exactly the same* subclass will suffice both for printing purposes and for print preview purposes
- underlying this similarity is the fact that printing and print previewing are both driven by the PAGES active object, which requires in both cases to be fed by application code with a series of print elements (*the same set of print elements in both cases*).

The difference between XPRINTER and LPRINTER

First, note that XPRINTER is defined in the XADD library, which is not present in the ROM of the Series 3, so that print preview support does not exist on the Series 3 - only on the Series 3a. (Further to this, the versions of LPRINTER on the Series 3 and the Series 3a are also critically different, in a few small but key places - though the calling interface remains exactly the same.)

Next, witness the entirety of the code of XPRINTER:

```
#include <xprinter.g>
#include <prev.g>
#include <xadd.g>

GLREF_D PR_APPMAN *w_am;
GLREF_D VOID *w_ws;
```

```
#ifdef JPIC
#pragma METHOD_CALL
#endif

METHOD VOID xprinter_destroy(PR_XPRINTER *self)
    {
    if (self->xprinter.locked)
        p_send3(w_ws,O_WS_LOCK,FALSE);
    p_supersend2(self,O_DESTROY);
    }

METHOD VOID xprinter_lpr_init(PR_XPRINTER *self,INT commid)
    {
    if (!commid)
        {
        p_send3(w_ws,O_WS_LOCK,TRUE);
        self->xprinter.locked=TRUE;
        }
    self->lprinter.subsqind=commid;     /* communicate with subclass */
    p_supersend2(self,O_LPR_INIT);
    if (!commid)
        return;
    f_newsend(CAT_XADD_XADD,C_PRVVIEW,O_PVV_INIT,self,commid,&self-
>lprinter.pages);
    }
```

and the entirety of the corresponding class definition:

```
CLASS xprinter lprinter
    {
    REPLACE destroy
    REPLACE lpr_init
    PROPERTY
        {
        WORD locked;
        }
    }
```

Evidently, `XPRINTER` adds two pieces of functionality to `LPRINTER` (one rather trivial, and the other more fundamental):

- `XPRINTER` locks the application whilst it is printing, to lessen the chance of “accidents” half-way through printing due to the user inadvertently switching files from the System screen (setting the application locked means the System screen will block any attempted file switch with a “XXX is busy” infoprint)
- `XPRINTER` expects an extra parameter to its `LPR_INIT` method; if the value of this is zero, `LPRINTER` code is invoked in printing mode, whereas if it is non-zero, print preview takes place instead.

The actual meaning of this additional parameter - `commid` - is the method number of the command manager that system code will invoke if the user chooses the ‘Print’ menu command *from within the menubar available inside print preview*.

Extended example of print and print preview using XPRINTER

To illustrate use of `XPRINTER` for print *and* print preview purposes, consider the following variation upon the example applications presented earlier:

- once again, the user is given the choice of specifying a range of dates to be printed
- the dates will be printed in two columns (as in the second of the two earlier examples)
- the *first* column will be printed in bold (by way of illustrating some features described in the middle portion of this chapter)
- rather than a series of dialogs following each other irrevocably, the various possible choices in the application are available, in more standard style, as choices from the menu bar
- there are four menu commands available: Exit, Print setup, Print preview, and Print

- in order to illustrate another (quite separate) point, the application also attempts to load and save its current print setup to file, on startup and on exit (though discussion of this particular feature of the application is deferred until the end of the chapter).

On installation of the OOP component of the SDK, the source code for this application is copied into a *\sibosdk\wdrprint* directory.

The category file

```
IMAGE demo

EXTERNAL olib
EXTERNAL hwim
EXTERNAL xadd

INCLUDE hwimman.g
INCLUDE dlgbox.g
INCLUDE xprinter.g
INCLUDE time.g
INCLUDE epoc.h

CLASS dewserv wserv
    {
    REPLACE ws_dyn_init
    }

CLASS decomman comman
    {
    REPLACE com_init
    REPLACE com_exit
    ADD dec_psetup
    ADD dec_preview
    ADD dec_print=decomman_dec_preview
    ADD dec_print_direct
    TYPES
        {
        typedef struct
            {
            UWORD dayno;
            UWORD ndays;
            UWORD gap;
            } DE_PRINT_DETAILS;
        }
    PROPERTY
        {
        DE_PRINT_DETAILS dets;
        }
    }

CLASS dedlg dlgbox
    {
    REPLACE dl_dyn_init
    REPLACE dl_key
    }
```

```
CLASS delp xprinter
    {
    REPLACE lpr_sense_text
    CONSTANTS
        {
        DELP_STATE_COL1     0
        DELP_STATE_COL2     1
        DELP_STATE_COL3     2
        }
    PROPERTY 1
        {
        VOID *time;
        VOID *wtab;
        UWORD colwid;
        UWORD right;
        UWORD state;
        UWORD ndays;
        TEXT dname[E_MAX_DAY_NAME];
        TEXT buf[LN_TIME_DATE_STR];
        }
    }
```

Command manager

The four commands in the menu bar - Exit, Print setup, Print preview, and Print - are handled by the command manager methods `com_exit`, `dec_psetup`, `dec_preview`, and `dec_print`. Note that the definition

```
ADD dec_print=decomman_dec_preview
```

means that the single routine `decomman_dec_preview` handles *both* the menu commands Print and Print preview. This is a common feature of applications supporting print preview as well as print. As in all cases of this sort, the code can distinguish which of the two menu commands has actually been chosen by testing the value of the additional `commid` parameter that is always passed, by system code, to command manager methods. Thus the code for `decomman_dec_preview` is

```
METHOD VOID decomman_dec_preview(PR_DECOMMAN *self,INT commid)
    {
    if (LaunchDialog(C_DEDLG,DELP_DLG,&commid))
        PrintOrPreview(commid);
    }
```

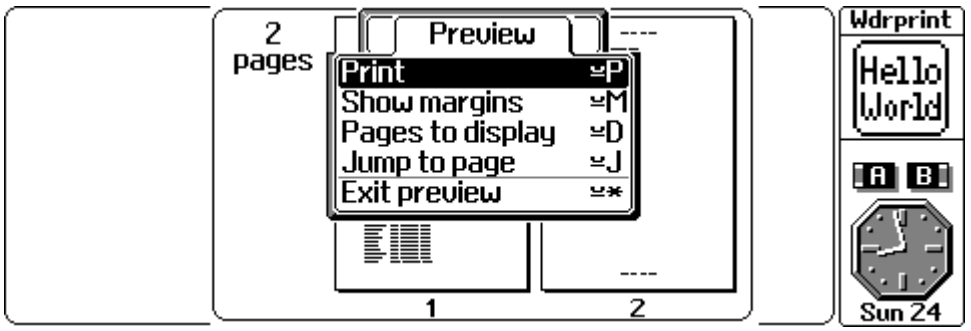
where `LaunchDialog` is the same as in previous examples (it is an entirely standard dialog-launching utility), and `PrintOrPreview` is as follows:

```
LOCAL_C VOID PrintOrPreview(INT commid)
    {
    commid=(commid==O_DEC_PREVIEW? O_DEC_PRINT_DIRECT: 0);
    hDestroy(f_newsend(CAT_DEMO_DEMO,C_DELP,O_LPR_INIT,commid));
    }
```

As can be seen, the parameter passed to the `lpr_init` method of `DELP` is either

- `0`, in the case when `DELP` is to *print*, or
- `O_DEC_PRINT_DIRECT`, in the case when `DELP` is to *print preview*.

Note that the `dec_print_direct` method of the command manager is *not* directly associated with the Print menu command from the base state of the application. On the contrary, as already stated, this menu command has corresponding command manager method `dec_print`, which is handled by the same code as the `dec_preview` method. The role of the `dec_print_direct` method is to service the Print menu command *from the special Print Preview submenu* that is available inside the print preview subsystem:



The code for the `dec_print_direct` method, in this example, is just

```
METHOD VOID decomman_dec_print_direct(VOID)
    {
    PrintOrPreview(O_DEC_PRINT);
    }
```

Note: the reason for using the term “direct” is that printing is to proceed directly, *without any additional ‘Print details’ dialog being presented first*. The print details are the same as in the dialog that invoked the print preview.

Apart from the methods of the command manager already covered above, `DECOMMAN` also has the following methods:

- the `dec_psetup` method just has one line, sending a `ws_edit_print_context` message to `w_ws`
- the `com_init` method writes the handle of the command manager into `DatApp2` (for convenience elsewhere in the code) and also calls the routine `TryLoadPrintContext`
- the `com_exit` method is as follows:

```
METHOD VOID decomman_com_exit(PR_DECOMMAN *self)
    {
    if (p_enter1(SavePrintContext))
        p_delete(SAVED_FILE_NAME);
    p_exit(0);
    }
```

More details of the `SavePrintContext` and `TryLoadPrintContext` methods are given nearer the end of this chapter.

Print details dialog

The code here contains two enhancements compared to the earlier example:

- if the dialog is visited more than once, it is seeded, in the `dl_dyn_init` method, with the values last set by the user (as sensed in the preceding `dl_key` method)
- the dialog *title* has to vary, to reflect whether Print preview is to follow, or Print.

The entire code in the dialog module for the application is

```
#include <demo.g>
#include <demo.rsg>
#include <hwim.h>

GLREF_D PR_DECOMMAN *DatApp2;
```

```
#pragma METHOD_CALL

METHOD VOID dedlg_dl_dyn_init(PR_DLGBOX *self)
    {
    INT commid;

    commid=(*(INT *)self->dlgbox.rbuf);
    if (commid==O_DEC_PREVIEW)
        hDlgSetTitleByRid(DELP_PREVIEW_TITLE);
    if (!DatApp2->decomman.dets.dayno)
        return;
    hDlgSetDtedit(1,DatApp2->decomman.dets.dayno);
    hDlgSetNcedit(2,DatApp2->decomman.dets.ndays);
    hDlgSetChlist(3,DatApp2->decomman.dets.gap);
    }

METHOD INT dedlg_dl_key(PR_DLGBOX *self)
    {
    DatApp2->decomman.dets.dayno=hDlgSenseDtedit(1);
    DatApp2->decomman.dets.ndays=hDlgSenseNcedit(2);
    DatApp2->decomman.dets.gap=hDlgSenseChlist(3);
    return(WN_KEY_CHANGED);
    }
```

Note that the actual `DE_PRINT_DETAILS` data structure edited in this dialog no longer exists purely on the stack (as it did in versions of this example discussed earlier in this chapter). Instead, to give this data greater persistence, it now exists within the property of the command manager - whose handle has been written to `DatApp2` for convenience. (Otherwise, this handle could be obtained from `w_ws->wserv.com`.)

The way the `dl_dyn_init` routine determines whether the dialog has been invoked before - ie determines whether there is data in the `DE_PRINT_DETAILS` struct that ought to overwrite the defaults provided by the resource controls for the dialog - is by testing the value of `DatApp2->decomman.dets.dayno`, to see whether it is non-zero. But note that, of course, the test for whether the dialog *title* should be changed is quite independent of this.

Application initialisation

Apart from the code in the `com_init` method, the other initialisation code for the application is in `main` itself, and in the `ws_dyn_init` method of the `wserv` subclass (as can be seen, there is nothing unusual in any of it):

```
GLREF_D WSERV_SPEC *wserv_channel;

LOCAL_C INT StatusWindowWidth(VOID)
    {
    P_EXTENT ext;

    wInquireStatusWindow(-1,&ext);
    return(ext.width);
    }

LOCAL_C VOID InitClientWindow(PR_DEWSERV *self)
    {
    W_WINDATA wd;
    PR_WIN *cliwin;

    wd.extent.tl.x=wd.extent.tl.y=0;
    wd.extent.height=wserv_channel->conn.info.pixels.y;
    wd.extent.width=wserv_channel->conn.info.pixels.x-StatusWindowWidth();
    cliwin=f_newsend(CAT_DEMO_HWIM,C_BWIN,O_WN_CONNECT,NULL,W_WIN_EXTENT,&wd);
    self->wserv.cli=cliwin;
    cliwin->win.flags=PR_BWIN_CORNER_4|PR_BWIN_SHADOW_1;
    p_send3(cliwin,O_WN_EMPHASISE,TRUE);
    hInitVis(cliwin);
    }
```

```
#pragma METHOD_CALL

METHOD VOID dewserv_ws_dyn_init(PR_DEWSERV *self)
    {
    wsSetList(W_STATUS_WINDOW_ICON,NULL,0);
    wStatusWindow(W_STATUS_WINDOW_BIG);
    InitClientWindow(self);
    }

GLDEF_C VOID main(VOID)
    {
    IN_HWIMMAN app;
    IN_WSERV wserv;

    p_linklib(0);

app.flags=FLG_APPMAN_RSCFILE|FLG_APPMAN_SRSCFILE|FLG_APPMAN_CLEAN|FLG_APPMAN_F
ULLSCREEN;
    wserv.com_cat=app.wserv_cat=p_getlibh(CAT_DEMO_DEMO);
    app.wserv_class=C_DEWSERV;
    wserv.com_class=C_DECOMMAN;
    p_send4(p_new(CAT_DEMO_HWIM,C_HWIMMAN),O_AM_INIT,&app,&wserv);
    }
```

XPRINTER subclass initialisation

All the initialisation code for the `XPRINTER` subclass of the applicaiton is called inside the first `lpr_sense_text` callback. This establishes:

- a `TIME` object, suitably prepared (as in previous examples) to render textual versions of given dates, and initialised with the start date from the `DE_PRINT_DETAILS` data structure
- a printer font width table (handle written to `wtab`) for the *bold* version of the default print font
- the value of the `colwid` property field, that gives the overall width for the first column:

```
#include <demo.g>
#include <demo.rsg>
#include <hwim.h>

GLREF_D PR_DECOMMAN *DatApp2;

#define TIME_FMT_FLAGS (PR_TIME_MONTH_NAME|PR_TIME_SUFFIX_NAME)

LOCAL_C VOID InitTimeObject(PR_DELP *self)
    {
    P_DAYSEC ds;
    SE_TIME_FORMAT fmt;

    ds.day=DatApp2->decomman.dets.dayno;
    ds.sec=0;
    self->delp.time=f_new(CAT_DEMO_OLIB,C_TIME);
    p_send4(self->delp.time,O_TO_SET,SET_TIME_DAYSEC,&ds);
    fmt.flags=TIME_FMT_FLAGS;
    p_send4(self->delp.time,O_TO_SET_FORMAT,&fmt,TIME_FMT_FLAGS);
    }

LOCAL_C UINT SenseBufWidth(PR_DELP *self,TEXT *pb)
    {
    return(p_send5(self->lprinter.wdr,O_WDR_SENSE_WIDTH,self-
>delp.wtab,pb,p_slen(pb)));
    }
```

```
LOCAL_C VOID FindWidthFirstColumn(PR_DELP *self)
    {
    INT i;
    TEXT buf[E_MAX_DAY_NAME];
    UWORD wid;

    for (i=0; i<7; i++)
        {
        p_nmday(&buf[0],i);
        wid=SenseBufWidth(self,&buf[0]);
        if (wid>self->delp.colwid)
            self->delp.colwid=wid;
        }
    self->delp.colwid+=SenseBufWidth(self,"  ");    /* two spaces */
    if (self->delp.colwid>(3*self->lprinter.width/4))
        {
        hInfoPrint(DELP_PAPER_NARROW);
        p_leave(RUN_ACTIVE_CLEANUP_NONOTIFY);
        }
    self->lprinter.subsqind=self->delp.colwid;
    }

LOCAL_C VOID InitWidthTable(PR_DELP *self)
    {
    self->delp.wtab=(VOID *)p_send5(self->lprinter.wdr,O_WDR_GET_WIDTH_TABLE,
                self->lprinter.f.fid,self->lprinter.f.height,
                self->lprinter.f.style|WDR_STYLE_BOLD);
    }
```

For more information relevant to parts of this code, see the appropriate sections earlier in this chapter.

The XPRINTER LPR_SENSE_TEXT callback

What is *particularly* noteworthy about the `DELP` code in this application (referring in fact to *the totality* of the code in this class) is that although there are, admittedly, differences from the `DELP` code given for earlier examples in this chapter, these differences *have nothing to do* with the extra support that this class is now providing for Print Preview as well as Print. These differences are purely to do with comparatively incidental points, such as the fact that, for example, the first column is now being printed in bold.

In other words, the extra support for Print Preview is achieved *without any change* at the `LPRINTER` subclass level - except that the definition of the class now specifies `XPRINTER` as the superclass, instead of just `LPRINTER`. The code *itself* could survive unchanged:

```
METHOD INT delp_lpr_sense_text(PR_DELP *self,WDR_PRINT *pr)
    {
    P_DATE dt;
    P_DAYSEC ds;

    if (!self->delp.time)
        {
        self->delp.ndays=DatApp2->decomman.dets.ndays;
        InitTimeObject(self);
        InitWidthTable(self);
        FindWidthFirstColumn(self);
        }
    else if (!self->delp.ndays)
        return(FALSE);
    switch (self->delp.state++)
        {
    case DELP_STATE_COL1:
        if (DatApp2->decomman.dets.gap)
            {
            p_send4(self->delp.time,O_TO_SENSE,SENSE_TIME_DATE,&dt);
            if (!dt.day)
                {
                pr->down=pr->height;
                if (DatApp2->decomman.dets.gap==1)
                    pr->down>>=1;
                }
            }
        p_send4(self->delp.time,O_TO_SENSE,SENSE_TIME_DAYSEC,&ds);
        p_nmday(&self->delp.dname[0],P_WEEK(ds.day));
        self->delp.right=self->delp.colwid-SenseBufWidth(self,&self-
>delp.dname[0]);
        pr->buf=(&self->delp.dname[0]);
        pr->style|=WDR_STYLE_BOLD;
        break;
    case DELP_STATE_COL2:
        pr->flags&=(~(WDR_PRINT_LINE|WDR_PRINT_TEXT));
        pr->flags|=WDR_PRINT_RIGHT;
        pr->right=self->delp.right;
        return(TRUE);
    case DELP_STATE_COL3:
        p_send4(self->delp.time,O_TO_SENSE,SENSE_TIME_DATESTR,&self-
>delp.buf[0]);
        pr->buf=(&self->delp.buf[0]);
        pr->flags&=(~WDR_PRINT_LINE);
        pr->indent=self->delp.colwid;
        self->delp.ndays--;
        p_send4(self->delp.time,O_TO_ADD_DAYS,1,0);
        self->delp.state=DELP_STATE_COL1;
        }
    pr->blen=p_slen(pr->buf);
    return(TRUE);
    }
```

Comments on the differences between XPRINTER and LPRINTER

One *minor* change that *would* have to be made, however, between `LPRINTER` subclass code and `XPRINTER` subclass code, would be in any replacement `lpr_init` method. This simply due to the fact that the `lpr_init` method of `XPRINTER` requires the additional `commid` parameter - to differentiate between the case of Print Preview and the case of Print.

Note incidentally that, for compatibility reasons, this extra parameter could not be added in at the `LPRINTER` level. This is on account of existing applications (ie pre-Series 3a applications) that interact with `LPRINTER` code without passing *any* parameter explicitly to the `lpr_init` method.

The way around this constraint, in the development of the Series 3a ROM code, can be seen from the code given earlier for the `lpr_init` methods of both `LPRINTER` and `XPRINTER`:

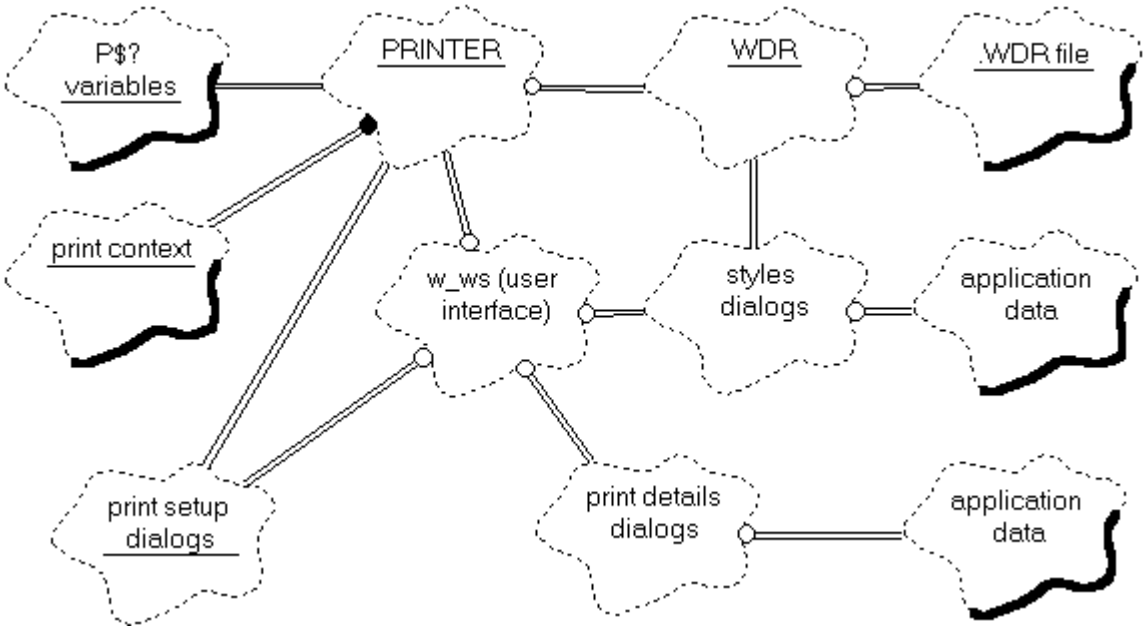
- the `subsqind` property field is re-used as a temporary “postbox” between the two classes
- after its use, in this way, at the initialisation stage, its value is set back to zero

- the rationale behind this is that the `subsqind` field cannot be written to, by application subclass code, until inside the first callback to `lpr_sense_text` (or `lpr_read`).

WDR printing miscellany

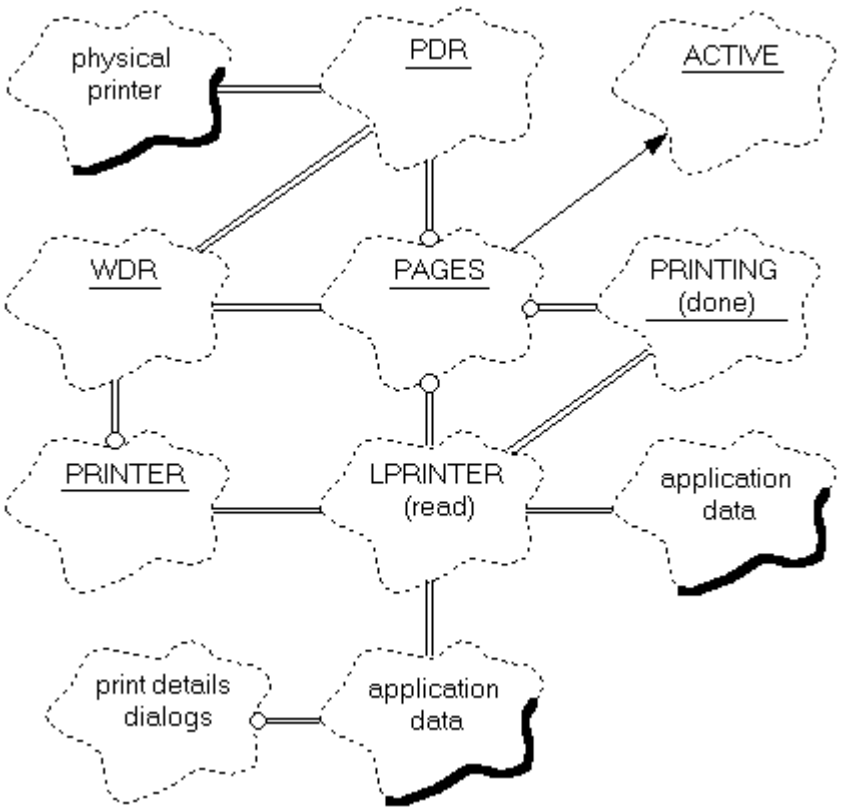
WDR printing classes pictorial overview

The following class diagram shows a possible application state *before printing actually starts*:



(A typical application need not, however, contain all the application-specific components shown here. For example, applications that print will not necessarily contain "styles dialogs".)

The next diagram shows a possible situation once printing is underway (for clarity, many aspects of the *previous* diagram are omitted from this one):



(Nothing too significant should be read into the type or the direction of the connections shown between different classes in these diagrams. As the foregoing chapter has made clear, the connections between the actual classes are, at times, rather more complex than could be done justice in any one diagram.)

The PDR class

The role of the PDR object is possibly worth mentioning. This receives all "printing output" from PAGES, and directs into towards the relevant IO channel - be it the parallel port, the serial port, or just an opened file (in the case of printing to file). The PDR class accesses data held by the WDR class, to let it know exactly which escape sequences (or whatever) are required to effect various results - such as changing from one font to another, indenting by a given amount, and so on.

For some kinds of printers, the PDR object has to be an instance of an appropriate *subclass* of the PDR class in FORM - rather than just being an instance of PDR itself. This happens when an appropriate flag is set in the *.wdr* file for that printer. In this case, system code looks for a suitably named DYL in the same directory as the *.wdr* file. See the *WDR Printing* chapter in the *Additional System Information* manual for some more details, or contact Psion directly for more information about writing PDR subclasses.

The role of the PDR object in the above diagrams is of interest for one additional reason: when the WDR print system is driven in preview mode, as opposed to printing mode, this is one of only two parts of the diagram to change. Rather than create any instance of PDR (or a printer-specific subclass thereof), system code in this case creates an instance of the FORM class PRVPDR. Rather than direct printing output towards any IO channel, this directs it towards a suitable window - the print preview window.

The *other* part of the diagram that changes is the PRINTING part: the PAGES mdone callback is directed to a different object, namely an instance of PRVVIEW. This is discussed briefly in the following section.

Print preview without XPRINTER

The code given earlier in this chapter for XPRINTER makes it clear what would be required should an application, for whatever reason, wish to access the print preview subsystem without using XPRINTER. Essentially, something equivalent to the following line of code is required:

```
    f_newsend(CAT_XADD_XADD,C_PRVVIEW,O_PVV_INIT,self,commid,&self-
>lprinter.pages);
```

This creates and initialises an instance of the XADD class PRVVIEW.

The code in the pvv_init method of PRVVIEW is as follows:

```
METHOD VOID prvview_pvv_init(PR_PRVVIEW *self,VOID *xp,INT commid,VOID
**ppages)
    {
    IN_PRVVIEW init;

    init.Calls.hread=xp;            /* xprinter */
    init.Calls.mread=O_LPR_READ;
    init.Calls.hdone=self;
    init.Calls.mdone=O_PVV_FALSE;
    init.PrintMethod=commid;
    init.Spare1 = init.Spare2 = 0;
    p_send4(self,O_WN_INIT,&init,FALSE);
    *ppages=self->prvview.pPages;
    self->prvview.Started=TRUE;
    p_send2(w_am,O_AM_START);
    }
```

Applications may wish to avoid calling *this* method, but they cannot practically avoid sending the PRVVIEW object the wn_init message. Note in this context the definition of the IN_PRVVIEW struct:

```
typedef struct
    {
    PAGES_CALLS Calls;
    WORD        PrintMethod;
    WORD        Spare1;
    WORD        Spare2;
    } IN_PRVVIEW;
```

where Spare1 and Spare2 should be set to zero.

The default mdone callback method, pvv_false, always just returns FALSE, and will be suitable for almost every client of PRVVIEW. (Without going into too many details, there are in fact *two* layers of done callbacks when print preview applies: a first level callback from PAGES to PRVVIEW, always using the method pvv_pages_done, and a possible second level callback from PRVVIEW to any specified recipient object.)

Finally, the significance of the final TRUE/FALSE parameter to the wn_init method of PRVVIEW can be seen in the following code at the very end of this wn_init method:

```
    if (DoAmStart)
        {
        self->prvview.Started = TRUE;
        p_send2(w_am,O_AM_START);
        }
```

The only reason that the code in pvv_init cannot pass the DoAmStart parameter as TRUE is in order to write back the handle of PAGES (effectively in LPRINTER property), prior to the call to am_start being made. This allows code in, for example, lpr_read callbacks (which take place, of course, before the am_start call returns) to access PAGES as required.

Saving and restoring print context from file

As mentioned earlier, the PRINTER class has methods allowing the print context (the subject matter of the 'Print setup' dialog suite) to be set and sensed. These methods can be utilised to allow the print context to be saved to file, if desired, and then restored the next time the file is opened.

Any application that wishes to save the print context to file has to answer a number of design decisions:

- what actual format should the data be stored in?
- what kind of integrity check might be performed on file data, before setting it into PRINTER property on application startup?
- what kind of error recovery procedure should be adopted, if there is any run-time error, either on saving the data, or on loading it?

This is not the place to discuss these matters at any length. Accordingly, many aspects of the example code, in the WDRPRINT subdirectory, will just be taken for granted in this discussion (though this is not to imply that there is anything special about the design decisions embodied therein).

What *can* be briefly covered here, however, are various methods of PRINTER, which fall into two categories: those *sensing* the print context, and those *setting* it.

The following code senses the print context and writes it out to file:

```
LOCAL_C INT SavePrintContext(VOID)
    {
    VOID *fcb;
    VOID *printer;
    UBYTE *p;
    struct
        {
        UBYTE model;
        UBYTE name[P_FNAMESIZE+1];
        } m;

    printer=w_ws->wserv.printer;
    if (!printer)
        return(0);
    f_open(&fcb,SAVED_FILE_NAME,P_FREPLACE|P_FSTREAM|P_FUPDATE);
    p=(UBYTE *)p_send2(printer,O_PR_GET_PARAMS);
    f_write(fcb,p,sizeof(PRINTER_PARAMS));
    m.model=p_send3(printer,O_PR_SENSE_MODEL,&m.name[0]);
    f_write(fcb,&m.model,1+p_slen(&m.name[0])+1);
    p=(UBYTE *)p_send3(printer,O_PR_GET_HD,PRINTER_HDR_TOP);
    f_write(fcb,p,p_slen(p)+1);
    p=(UBYTE *)p_send3(printer,O_PR_GET_HD,PRINTER_HDR_BOT);
    f_write(fcb,p,p_slen(p)+1);
    p_close(fcb);
    return(0);
    }
```

This code uses the following PRINTER methods:

- pr_get_params: returns the address of the PRINTER_PARAMS data structure inside PRINTER property
- pr_sense_model: writes a ZTS specification of the current *.wdr* file, and returns the index number of the current printer model *within* this file
- pr_get_hd: returns the address of a ZTS giving either the header text or the footer text, depending on the final parameter passed.

Note that aspects of for example the *alignment* and the *printer font* of the header and footer are stored as parts of the fixed-length PRINTER_PARAMS struct: it is only the (variable length) *text* of the header and footer that requires a separate method to sense it.

The code to read the print context from file, and to set it into PRINTER property, is rather longer - but that is only because of the integrity tests that it makes:

```
LOCAL_C VOID TryLoadPrintContext(VOID)
    {
    P_INFO junk;
    VOID *fcb;
    VOID *printer;
    UBYTE buf[512];
    INT blen;
    INT ind;
    UBYTE *p1,*p2,*p3,*p4;

    if (p_finfo(SAVED_FILE_NAME,&junk))
        return;     /* eg file does not exist */
    f_open(&fcb,SAVED_FILE_NAME,P_FOPEN|P_FSTREAM|P_FSHARE);
    blen=f_read(fcb,&buf[0],512);
    p_close(fcb);
    if (blen<=sizeof(PRINTER_PARAMS)+2)
        goto file_corrupt;
    p1=(&buf[0]);
    blen-=sizeof(PRINTER_PARAMS);
    p2=p1+sizeof(PRINTER_PARAMS);
    ind=p_bloc(p2+1,blen-1,0);
    if (ind<0)
        goto file_corrupt;
    p3=p2+1+ind+1;
    blen-=1+ind+1;
    if (blen<=0)
        goto file_corrupt;
    ind=p_bloc(p3,blen,0);
    if (ind<0)
        goto file_corrupt;
    p4=p3+ind+1;
    blen-=ind+1;
    if (blen<=0)
        goto file_corrupt;
    ind=p_bloc(p4,blen,0);
    if (ind!=blen-1)
        {
    file_corrupt:
        hInfoPrint(DELETING_CORRUPT_FILE);
        p_delete(SAVED_FILE_NAME);
        return;
        }
    p_send2(w_ws,O_WS_ENS_PRINT_CONTEXT);
    printer=w_ws->wserv.printer;
    p_bcpy((VOID
*)p_send2(printer,O_PR_GET_PARAMS),p1,sizeof(PRINTER_PARAMS));
    p_send5(printer,O_PR_SET_MODEL,FALSE,p2+1,*p2);
    p_send4(printer,O_PR_SET_HD,PRINTER_HDR_TOP,p3);
    p_send4(printer,O_PR_SET_HD,PRINTER_HDR_BOT,p4);
    }
```

The two new `PRINTER` methods used here are:

- `pr_set_model`: the first parameter is a pointer to a ZTS giving the *.wdr* filename, and the second is the printer model index number
- `pr_set_hd`: the first parameter specifies whether the header text or the footer text is being set, and the second gives a ZTS containing this text.

CHAPTER 14

LINK PASTE

This chapter contains a practical introduction to programming “Link Paste” (also called “Bring”):

- how to service link paste requests from other applications (the *server* side of link paste)
- how to obtain link paste data from other applications (the *client* side of link paste)
- the role of the OLIB classes `LINKSV`, `LINKCL`, and `SYSTEM`
- the definitions of various link data “types”
- specific HWIM assistance for link paste involving edit windows.

For the sake of concreteness, the discussions in this chapter are mainly based around various modifications and extensions of the *Ehello* example application that features in the opening sections of the *Edit Windows* chapter, and which is optionally installed into the *\sibosdk\ehello* directory. It should be stressed, however, that it is possible to grasp the concepts involved in programming link paste independently of any appreciation of programming edit windows.

The modifications required to the original *Ehello* code, to add link paste functionality, are all given below.

The server side of link paste

When the user selects the ‘Bring’ menu command in application X, say, and sees new data added to application X, this data has come from another application - Y, say - that was in background at the time the menu command was issued. In this example, application X is the “link client” and application Y is the “link server”.

It is important to realise that the data is fetched directly from application Y *at the time the ‘Bring’ request is issued*. That is, the data is *not* fetched into any independent “clipboard application” at any earlier stage (eg when application Y was in foreground). There is no “clipboard application” in the Epoc architecture (neither at the OS level nor at the HWIM level - nor at any intermediate level).

Thus applications which wish to function as link servers have to be prepared to receive requests for data *whilst they are in background*. These requests are in fact interprocess communication (IPC) messages of a particular type - but this is largely hidden from HWIM applications, with the details of the IPC being handled, on the server side, by the OLIB `LINKSV` class.

Creating a LINKSV subclass instance

The `LINKSV` class contains two deferred methods, `ls_set_format` and `ls_get_data`, that *have* to be supplied by any link-serving application. For this reason, applications never create an instance of `LINKSV` itself, but rather an instance of an application-specific subclass of `linksv`.

For example, the following additional class definition could be added into the file *ehello.cat* (see later for the significance of the property fields)

```
INCLUDE ipc.g

...

CLASS ehlinksv linksv
    {
    REPLACE ls_set_format
    REPLACE ls_get_data
    PROPERTY
        {
        WORD sellen;
        TEXT *pbuf;
        }
    }
```

and the following line of code should be added to application start-up code, eg in the `ws_dyn_init` method of the `WSERV` subclass (in other applications, the code could be placed instead in the `com_init` method of the `COMMAN` subclass - depending on what was most convenient):

```
f_newsend(CAT_EHELLO_EHELLO,C_EHLINKSV,O_SV_INIT);
```

Note that there is rarely any need to record the handle of the created `LINKSV` subclass instance anywhere in application code:

- the object will continue in existence throughout the lifetime of the application, and so there is no need to hold onto its handle just in order to send it a `destroy` message at some later stage
- the handle of the object *is* in fact recorded in the linked-list of so-called “server handles” held by the system `IPCS` object, and whenever a suitable IPC message is received by the application, the `IPCS` object automatically redirects it to the `LINKSV` object.

Note also that any `sv_init` call will *fail* - with panic 55 - unless the flag `FLG_APPMAN_IPCS` is set in `main` (in fact, this flag is effectively *always* set for HWIM applications on the Series 3a - but it is good practice to set it explicitly, in `main`, whenever an application creates a `SERVER` object of its own). Therefore the line in *ehmain.c* that defines the `APPMAN` flags for the application becomes

```
app.flags=FLG_APPMAN_RSCFILE|FLG_APPMAN_SRSCFILE|FLG_APPMAN_CLEAN
                                |FLG_APPMAN_IPCS|FLG_APPMAN_LINKING;
```

so that `w_am` is initialised with an `IPCS` component (see later for the significance of the `FLG_APPMAN_LINKING` flag).

Note finally that it is impossible to delay creating the `LINKSV` object until an actual link paste data request is received - which might at first seem a good idea (in order to cut down on the memory overhead of an object that might never actually be needed). The point is that the `LINKSV` object is needed in order to receive the request in the first place (on pain of a panic 158). However, it is common practice to delay the allocation of *additional* data buffers (such as the `pbuf` field of `EHLINKSV` will point to) until actually required.

Declaring link paste server status

In order for an application to receive a link paste data request, *two* pre-conditions have to be satisfied:

- the application has created and initialised a `LINKSV` object - as described above
- the application has sent the `SYSTEM` component of `w_am` an `sy_link_server` message, declaring the presence (and type) of linkable data.

Declaring current link paste server status is done every time the application passes into background. Therefore the application has to subclass the `ws_background` method of `WSERV`. For example, the declaration of `EHWSERV` in *ehello.cat* becomes

```
CLASS ehwserv wserv
    {
    REPLACE ws_dyn_init
    REPLACE ws_background
    }
```

with the actual contents of the `ws_background` method being:

```
GLREF_D PR_APPMAN *w_am;
GLREF_D PR_EDWIN *DatApp3;

...

METHOD VOID ehwserv_ws_background(PR_EHWSERV *self)
    {
    INT which;

    if (!(self->wserv.flags&PR_WSERV_RECEIVED_KEY))
        return;
    which=0;
    if (DatApp3->edwin.select)
        which=1<<DF_LINK_TEXT;
    p_send4(w_am->appman.system,O_SY_LINK_SERVER,which,0);
    }
```

The significance of the test on `PR_WSERV_RECEIVED_FLAG` in this code is to prevent the application "stealing" the link paste server status just because the user happens to task through it. Suppose that the user has, long ago, selected some text in application A, but now wants to link paste some data from application B to application C. The user therefore highlights the data in B, and then tasks to C. But suppose that A is tasked to foreground first - as may well happen if, in particular, A and C are instances of the *same* application (eg two different Spreadsheet files). So long as the user pressed no keys while transiently tasking through A (apart from the task keys themselves), the above test ensures that the eventual 'Bring' menu command in C fetches data from B and not from A.

The bulk of the link-paste specific code in an application's `ws_background` method usually consists of determining the set of link-paste "types" that the current state of the application can support. See later for a discussion of various different standard link paste types. In the example above, the decision is a straightforward choice between two possibilities: either there is some selected text in the editor - in which case linkable data of type `DF_LINK_TEXT` is available - or else there is not - in which case *no* linkable data is available.

In general, an application will often be able to provide more than one type of data at any given time. For example, an application will often be able to offer both "plain text" and "native format data" - depending on who the recipient of the data is. Thus if the recipient of the data is another instance of the same application (eg one spreadsheet link-pasting data from another), a "native format" data transfer will generally transfer more information that the kind of "plain text" data transfer that would happen, instead, if the recipient application is *not* the same.

For this reason, the `sy_link_server` method takes two parameters, which between them form a `ULONG` "mask" made up of 32 bits. The more different bits that are set, the more different types of formats which the link server is prepared to "render" at that moment. To signal that the link server is prepared to render format type `DF_LINK_TEXT`, the bit `(1<<DF_LINK_TEXT)` should be set in the mask, and so forth.

Note that if the application currently has *no* data available for link paste, the mask value of zero should be reported. This will clear any record that may be left over from earlier, when the application *did* have data available for link paste.

In the example above, `DatApp3` has been set to point, for convenience, to the `EDWIN` object in the main window of the application - via the following line added to `ehbwin_wn_init`:

```
GLREF_D PR_EDWIN *DatApp3;

...

    DatApp3=self->ehbwin.edwin;
```

The test on `DatApp3->edwin.select` therefore just detects whether there is any select region in the editor: if there is, the application is prepared to render plain text format link paste data; otherwise, it has no data to render.

Initialising the SYSTEM component of w_am

Before the application can send a `sy_link_server` message to `w_am->appman.system`, it is necessary to arrange for the creation and initialisation of the `SYSTEM` component of `w_am`. This is arranged very simply: by setting the `FLG_APPMAN_LINKING` flag in `main`.

Note carefully that setting the FLG_APPMAN_SYSTEM flag will *not* have the desired effect. That would succeed in creating and initialising an instance of the SYSTEM object, but it would be the *wrong* type of SYSTEM object - being oriented towards the process *sys$shll.img* instead of towards the process *sys$wsrv.img* (see the *OLIB Reference* manual for more details).

The anatomy of a link paste transaction (server-side viewpoint)

In general, a link-paste transaction is seen, at the server end, as

- one call to ls_set_format
- followed by a *number* of calls (one or more) to ls_get_data.

The transaction takes place in a number of stages, in general, since there is a limit to the amount of data that can reasonably be transferred in any one stage, and in order that the foreground application can remain responsive to redraw requests in the meanwhile. The link server will usually possess some “state variables” - generally in the property of its LINKSV subclass - in order to keep track of the progress of the current transaction.

The purpose of the ls_set_format call is

- to specify which of the profferred data formats is actually being requested
- to allow the link server to reset its state variables, reflecting the fact that such-and-such a type of link paste data transfer is about to start.

The purpose of each subsequent ls_get_data call is

- to assemble data into a suitable buffer (if necessary) and to set a suitable variable (namely the linksv.buf property field) to point to this buffer
- to specify the length of the data to be transferred in this stage of the transaction (this should be written to the linksv.len property field)
- to indicate, by means of the return value (TRUE or FALSE) whether the transaction has completed.

Notice that a link paste transaction can end either because:

- the link server has no more data to transmit
- the link client does not wish to receive any more data.

In the latter case, system code informs the LINKSV subclass by means of specifying a *negative* len parameter - see the example code below. (Ordinarily, this parameter gives the size of the buffer in the data space of the recipient where the data is to be copied to.)

Note that the buffer whose address is written to linksv.buf must *not* be on the stack (for obvious reasons).

Example LINKSV code

The code for the ls_set_format and ls_get_data methods of the EHLINKSV methods is as follows:

```
#include <ehello.g>
#include <p_gen.h>

GLREF_D PR_EDWIN *DatApp3;

LOCAL_C VOID LinkServiceOver(PR_EHLINKSV *self)
    {
    p_free(self->ehlinksv.pbuf);
    self->ehlinksv.pbuf=NULL;
    self->ehlinksv.sellen=0;
    }

#pragma METHOD_CALL
```

```
METHOD VOID ehlinksv_ls_set_format(PR_EHLINKSV *self,INT type)
    {
    UWORD top;
    SENSE_EDWIN sense;
    SE_EDWIN se;

    LinkServiceOver(self);
    if (type!=DF_LINK_TEXT)
        p_leave(E_GEN_NSUP);
    p_send3(DatApp3,O_EW_SENSE,&sense);
    if (sense.cursor<sense.anchor)
        {
        top=sense.cursor;
        self->ehlinksv.sellen=sense.anchor-top;
        }
    else if (sense.cursor>sense.anchor)
        {
        top=sense.anchor;
        self->ehlinksv.sellen=sense.cursor-top;
        }
    else
        return;
    self->ehlinksv.pbuf=f_alloc(self->ehlinksv.sellen);
    p_send3(DatApp3,O_WN_SENSE,&se);
    p_bcpy(self->ehlinksv.pbuf,se.buf+top,self->ehlinksv.sellen);
    }

METHOD INT ehlinksv_ls_get_data(PR_EHLINKSV *self,INT len)
    {
    if (len<0 || !self->ehlinksv.sellen)
        {
        LinkServiceOver(self);
        return(FALSE);
        }
    if (len>self->ehlinksv.sellen)
        len=self->ehlinksv.sellen;
    self->linksv.len=len;
    self->linksv.buf=self->ehlinksv.pbuf;
    self->ehlinksv.sellen=0;
    return(TRUE);
    }
```

In the `ls_set_format` method, the code somewhat kindly just calls `p_leave(E_GEN_NSUP)` in any case that the client requests data not available for rendering. Arguably, it might be more appropriate to panic the client in this case:

```
p_ppanic(self->server.cid,xxx);
```

with some well-chosen panic number `xxx` (since the application has at no time ever declared that it could supply any other format of data - so there must be a bug in the client program for requesting such data).

General remarks about link servers

Note that a request for data to be rendered for link paste can be received even if

- the application has one or more dialogs showing
- the application has a menu showing
- the application has a help screen showing.

The HWIM architecture handles this completely smoothly: the application receives `ws_background`, `ls_set_format`, and `ls_get_data` messages *completely independently of whether there are dialogs or menus (etc) current*. This is in contrast with the case of, for example, most Hwif or OPL/w applications, when any link paste request IPC messages would go completely unacknowledged in such a case. (And since there is an assumption in `LINKCL` code that link paste request IPC messages do *not* remain unacknowledged - on pain of the link client application hanging - this is a strong reason why non-HWIM applications should, in general, avoid declaring themselves as link paste servers.)

Some standard link paste data formats

In most cases, applications need only consider three standard link paste data formats:

- `DF_LINK_TEXT`, in which data is transmitted as a series of buffers of purely printable characters
- `DF_LINK_TABTEXT`, in which the buffers of data can contain, in addition, tab characters
- `DF_LINK_PARAS`, in which the buffers of data can also contain embedded paragraph delimiters.

(In addition to these *standard* formats, applications may also consider any number of application-specific so-called *native* formats. Native formats are discussed later in this chapter.)

Now the code discussed so far may have given the impression that a `DF_LINK_TEXT` data transfer only consists of *one* buffer of text. But that would be a mistaken impression. To see this, consider the following simple changes in the *Ehello* code:

- declare an extra `WORD` property field, `ntimes`, for `EHLINKSV`
- change the `ehlinksv_ls_get_data` method so that the `sellen` property field only gets reset to zero after the available data has been link pasted out three times in all.

Thus the `ls_get_data` method becomes

```
METHOD INT ehlinksv_ls_get_data(PR_EHLINKSV *self,INT len)
    {
    if (len<0 || !self->ehlinksv.sellen)
        {
        LinkServiceOver(self);
        return(FALSE);
        }
    if (len>self->ehlinksv.sellen)
        len=self->ehlinksv.sellen;
    self->linksv.len=len;
    self->linksv.buf=self->ehlinksv.pbuf;
    if (!--(self->ehlinksv.ntimes))
        self->ehlinksv.sellen=0;
    return(TRUE);
    }
```

and a new line

```
    self->ehlinksv.ntimes=3;
```

appears at the end of the `ls_set_format` method. Highlighting some text in the editor and then choosing 'Bring' in an application such as the Word Processor or the Database now results in the highlighted text being transferred three times in all - going into three different paragraphs in the process.

Note however that some link clients - such as the Series 3a Agenda - will terminate the transaction after only absorbing the first buffer of data. This is perfectly within their right (see below for the details of how to achieve this result).

DF_LINK_TEXT and DF_LINK_PARAS contrasted

Next, consider another change in *Ehello* code, in which the line in the `ws_background` method now declares the availability of `DF_LINK_PARAS` as well as `DF_LINK_TEXT`:

```
    if (DatApp3->edwin.select)
        which=(1<<DF_LINK_TEXT)|(1<<DF_LINK_PARAS);
```

At the same time, the test on `type` in the `ls_set_format` method has tobeome less restrictive:

```
    if (type!=DF_LINK_TEXT && type!=DF_LINK_PARAS)
        p_leave(E_GEN_NSUP);
```

Suppose the text `Hello world` is highlighted in *Ehello* and the user tasks to the Word Processor and invokes 'Bring'. The text that appears in the Word Processor window this time is

```
Hello worldHello worldHello world
```

ie with all three copies being added into the current paragraph, whereas before, when DF_LINK_TEXT was specified, the text appearing in the Word Processor wndow would have been

```
Hello world
Hello world
Hello world
```

with the three copies going to *different* paragraphs.

This experiment confirms that different link paste formats can differ not only in the *contents* of the buffers transferred, but also in the *interpretation* of the data in these buffers.

When DF_LINK_PARAS applies, the link paste server agrees to include paragraph delimiters (ie character \0's) in place in the text being transmitted, and the link paste client agrees not to infer any additional paragraph breaks between separate buffers transmitted. When DF_LINK_TEXT (or DF_LINK_TABTEXT) applies, however, the server agrees not to pass any zeros in place, and the client must infer a paragraph break in between each pair of buffers of text.

One limitation of DF_LINK_TEXT should now be apparent, involving the size of the buffer used to transfer the text. If a paragraph of text is longer than the size of this buffer, it will have to be split into two, with the different parts being interpreted as belonging to two different paragraphs. (It is possible to observe this effect with the *Notes* example application - which only uses the DF_LINK_TEXT format.)

Word wrap and link paste

Note that soft line breaks on the screen of the link server are generally *ignored* by link paste protocols. Here, a "soft line break" is one that is caused purely by the application of word-wrap, and which might well fall in a different place were the screen window changed or the screen display font changed.

In general, word wrap *in the client* will produce a very different result to word wrap *in the server*. For this reason, none of the standard link paste formats pay any regard to soft line breaks (this matches the fact that, as discussed in the *Edit Windows* chapter, there is no representation of soft line breaks *at the document level* of an EDWIN object).

For example, the terminal emulation application *Comms* always requests DF_LINK_PARAS, if it is available, and word-wraps the text received according to the "Bring margin" which the user can specify independently. Note that specifying DF_LINK_TEXT would be *less* satisfactory, for the reason (noted earlier) that paragraphs in the link server would sometimes end up split in two - at an apparently random position.

DF_LINK_TABTEXT

In some ways, DF_LINK_TABTEXT does for embedded tabs what DF_LINK_PARAS does for embedded paragraph delimiters:

- if a client asks for DF_LINK_TABTEXT, it is prepared to hunt for tabs in the passed buffers, and interpret them as makes sense *within the context of the client application*
- if a link server is asked for DF_LINK_TEXT, it must ensure that tab characters are all stripped out of the text before it is transmitted.

For example, the Spreadsheet application regards embedded tabs as *column delimiters*. Edit windows, on the other hand, interpret embedded tabs according to the tabstops (and other relevant parameters) applicable to that edit window at the time.

If tab characters have to be removed before transmission, it is standard simply to convert them into *single* spaces (it makes little sense to attempt to convert them to a *variable* number of spaces, since in general the number of spaces required is going to depend on settings in the client context).

The hierarchy of text types

Note that the functionality of DF_LINK_PARAS is assumed to be a *superset* of that of DF_LINK_TEXT. Any application which can handle embedded paragraph delimiters is assumed to be able to handle embedded tab characters.

The client side of link paste

Just as there is an OLIB class, LINKSV, which encapsulates most of the functionality of the server side of link paste, so also is there an OLIB class, LINKCL, which encapsulates most of the functionality of the client side of link paste. Between them, these two classes protect application programmers from needing to worry about the details of the IPC messaging involved in the implementation of link paste.

Determining whether there is suitable data available

When an application receives a 'Bring' menu command, one of the first things it has to do is to find out

- if any other application has registered data as available for link pasting
- what formats that data can be rendered into.

To this end, the SYSTEM component of w_am has to be sent an sy_link_paste message, eg as follows:

```
WORD lkpid;
ULONG fmt;

lkpid=p_send3(w_am->appman.system,O_SY_LINK_PASTE,&fmt);
if (!lkpid || !(fmt & (1<<DF_LINK_TEXT)))
    {
    hInfoPrint(-SYS_NOTHING_TO_BRING);
    return;
    }
```

The ULONG mask of available formats, if any, is written into fmt, and the PID of the process which has registered the data is written to lkpid.

Before the application can send a sy_link_paste message to w_am->appman.system, it is necessary to arrange for the creation and initialisation of the SYSTEM component of w_am. This is arranged very simply: by setting the FLG_APPMAN_LINKING flag in main.

Note incidentally that it is perfectly possible for an application to act as a link client but not as a link server. Such an application would have to set FLG_APPMAN_LINKING, but would not need to set FLG_APPMAN_IPCS (unless, of course, it created *other* kinds of SERVER objects, ie apart from LINKSV).

In the above code fragment, a test is made on fmt as well as on lkpid. A test should *always* be done on fmt, though the *nature* of the test made will of course depend on which kinds of link paste data the application is prepared to accept.

The text of the system message SYS_NOTHING_TO_BRING is, in English, *Nothing to bring*. Applications are free to substitute more specific messages if they wish.

The anatomy of a link paste transaction (client-side viewpoint)

Whereas LINKSV is designed to be *subclassed* - so that applications never create a direct instance of LINKSV - LINKCL is useable as it stands. For this reasons, applications have no need to declare any subclass of LINKCL in their *.CAT* file.

Thus application link paste client code will usually contain a line such as

```
link=f_newsend(CAT_EHELLO_OLIB,C_LINKCL,O_LC_START,lkpid,DF_LINK_TEXT);
```

directly creating and initialising an instance of LINKCL. Here, the PID of the link server is specified as one parameter, and the required format type is specified in another. (The format specified in this lc_start message is passed through to the ls_set_format method processed by the link server.)

Note another contrast with the case of LINKSV: the instance of LINKCL is only created when explicitly needed - in response to a 'Bring' menu command. The handle of the instance needs to be stored

- so that subsequent LC_GET_DATA messages can be sent to it, for each buffer of data to be collected
- so that an LC_STOP message can be sent to it, if required
- so that the object can be destroyed at the end of the transaction.

In practice, once created, the LINKCL object usually has its handle added to the cleanup list, and the way the object is destroyed is by a subsequent call to cl_clean_item.

Simple example of use of LINKCL

Consider the following modification of *Ehello*: when the key combination PSION-ENTER is received, it is regarded as a 'Bring' menu instruction (recall that, for simplicity, *Ehello* has only the barest bones of a real menu bar). Code gets added to *ehbwin.c* as follows:

```
#include <olib.h>
#include <s_.h>

GLREF_D PR_APPMAN *w_am;
GLREF_D PR_WSERV *w_ws;
GLREF_D PR_EDWIN *DatApp3;

LOCAL_C VOID DoLinkPaste(VOID)
    {
    WORD lkpid;
    ULONG fmt;
    VOID *link;
    INT cl_link;
    TEXT buf[52];
    WORD len;

    lkpid=p_send3(w_am->appman.system,O_SY_LINK_PASTE,&fmt);
    if (!lkpid || !(fmt & (1<<DF_LINK_TEXT)))
        {
        hInfoPrint(-SYS_NOTHING_TO_BRING);
        return;
        }
    link=f_newsend(CAT_EHELLO_OLIB,C_LINKCL,O_LC_START,lkpid,DF_LINK_TEXT);
    cl_link=cl_add_object(link);
    len=p_send4(link,O_LC_GET_DATA,&buf[0],50);
    if (len>0)
        {
        buf[len]=0;
        p_send4(DatApp3,O_EW_REPLACE,&buf[0],0);
        }
    w_ws->wserv.flags&=(~PR_WSERV_RECEIVED_KEY);
    cl_clean_item(cl_link);
    }

#pragma METHOD_CALL

METHOD VOID ehbwin_wn_key(PR_EHBWIN *self,INT keycode,INT mods)
    {
    SE_EDWIN sense;

    if (keycode!=W_KEY_RETURN)
        p_send4(self->ehbwin.edwin,O_WN_KEY,keycode,mods);
    else if (mods&W_PSION_MODIFIER)
        DoLinkPaste();
    else
        ...
    }
```

In a more general setting, one call to lc_start will normally be followed by a *sequence* of calls to lc_get_data, continuing until the len return value from one of them is negative (it will actually be the value E_FILE_EOF, but it is not necessary to test for this explicitly).

The client also has the option of terminating the transaction by calling lc_stop at any stage. This has not been done in the above example, simply because the destroy method of LINKCL (which is triggered by the above call to cl_clean_item) automatically sends self an lc_stop message.

Note that the client has to specify the *amount* of data it is prepared to accept, at each stage of the transaction, by means of the final parameter to the lc_get_data message. This value is communicated to the link server as the len parameter in the ls_get_data message.

The significance of the line of code

```
w_ws->wserv.flags&=(~PR_WSERV_RECEIVED_KEY);
```

is to avoid the link client “stealing” the link paste server status, when it next passes into background.

Special help with link pasting to and from edit windows

The ew_bring_in method of EDWIN

In practice, any application that wishes to link paste text into an instance of `EDWIN` would actually use the `ew_bring_in` method of that class - which encapsulates the stages of

- creating the `LINKCL` object
- sending the relevant `lc_start` and `lc_get_data` messages
- sending various messages to itself.

The `ew_bring_in` method also encapsulates knowledge of the various standard types of textual link paste formats. For interest, the complete code of `edwin_ew_bring_in` follows (though it will be necessary to read the *Edit Windows* chapter carefully to appreciate some parts of it - eg some of the utility routines used):

```
METHOD INT edwin_ew_bring_in(PR_EDWIN *self,INT lkpid,INT format)
    {
    PR_ROOT *link;
    INT cl_link;
    INT SingleShot;
    UINT pos;
    UINT totlen;
    INT len;
    INT err;
    INT offset;
    TEXT buf[258];
```

```
        CheckNotReadOnly(self);
        SingleShot=format&EW_BRING_SINGLE_SHOT;
        if (format&(1<<DF_LINK_PARAS))
            format=DF_LINK_PARAS;
        else if (format&(1<<DF_LINK_TABTEXT))
            format=DF_LINK_TABTEXT;
        else
            format=DF_LINK_TEXT;
        link=f_newsend(CAT_HWIM_OLIB,C_LINKCL,O_LC_START,lkpid,format);
        cl_link=cl_add_object(link);
        pos=self->edwin.cpos;
        totlen=0;
        buf[0]=0;
        offset=1;
        while ((len=p_send4(link,O_LC_GET_DATA,&buf[1],256))>=0)
            {
            if (!offset)
                len++;
            if ((err=p_entersend5(self,O_EW_EP_INSERT,pos,&buf[offset],len))!=0)
                {
                p_send4(self->edwin.doc,O_EP_DELETE,self->edwin.cpos,self-
    >edwin.cpos+totlen);
                cl_clean_item(cl_link);
                p_send3(self,O_EW_LEAVE,err);
                }
            totlen+=len;
            pos+=len;
            if (SingleShot)
                {
                p_send2(link,O_LC_STOP);
                break;
                }
            if (format!=DF_LINK_PARAS)
                offset=0;
            }
        self->edwin.clen+=totlen;
        EdwinFwdChange(self);
        SetEdwinSelect(self,self->edwin.cpos,totlen);
        w_ws->wserv.flags&=(~PR_WSERV_RECEIVED_KEY);
        cl_clean_item(cl_link);
        return(0);      /* confirm no leave */
        }
```

Simple example of calling EW_BRING_IN

The code in the `ncoe_bring` method of the command manager of the example *Notes* application (optionally installed into *\sibosdk\notes*) shows how simple it can be to call `ew_bring_in`:

```
    METHOD VOID nocomman_ncoe_bring(PR_NOCOMMAN *self)
        {
        INT lkpid;
        ULONG lkfmt;

        CheckEditing(self);
        lkpid=p_send3(w_am->appman.system,O_SY_LINK_PASTE,&lkfmt);
        if (!lkpid || !(lkfmt & (1<<DF_LINK_TEXT)))
            {
            hInfoPrint(-SYS_NOTHING_TO_BRING);
            return;
            }
        p_send4(DatApp3,O_EW_BRING_IN,lkpid,1<<DF_LINK_TEXT);
        }
```

Note however that the value `EW_BRING_SINGLE_SHOT` can be *or*red into the specified format mask, to force the transaction to terminate after just one stage. If the code given earlier for the *Ehello* application were to be modified to call `ew_bring_in` instead, `EW_BRING_SINGLE_SHOT` would need to be specified in that case.

The EWLINKSV class

Just as the ew_bring_in method of EDWIN provides system support for link pasting *into* edit windows, so also is there system support from link pasting *out* of edit windows. This is the EWLINKSV class, from HWIM.

Despite its name, EWLINKSV is *not* a subclass of LINKSV; rather, it is a subclass of ROOT, but its name indicates its intended use as a *component* of a LINKSV (subclass) object.

For example, consider the declaration of the NOLINKSV class in the *Notes* application category file:

```
CLASS nolinksv linksv
    {
    REPLACE ls_set_format
    REPLACE ls_get_data
    PROPERTY
        {
        PR_EWLINKSV *ewls;
        }
    }
```

The intended use of EWLINKSV can be seen from the following example code (providing the ls_set_format and ls_get_data methods of NOLINKSV):

```
#include <notes.g>
#include <hwim.h>

GLREF_D PR_EDWIN *DatApp3;

LOCAL_C VOID DestroyLinker(PR_NOLINKSV *self)
    {
    hDestroy(self->nolinksv.ewls);
    self->nolinksv.ewls=NULL;
    }

#pragma METHOD_CALL

METHOD VOID nolinksv_ls_set_format(PR_NOLINKSV *self,INT type)
    {
    DestroyLinker(self);
    self-
>nolinksv.ewls=f_newsend(CAT_NOTES_HWIM,C_EWLINKSV,O_EWLS_INIT,DatApp3,type);
    }

METHOD INT nolinksv_ls_get_data(PR_NOLINKSV *self,INT len)
    {
    if (len>0)
        {
        if (((INT)(self->linksv.len=p_send4(self->nolinksv.ewls,
                                O_EWLS_EXTRACT,&self->linksv.buf,len)))>=0)
            return(TRUE);
        }
    DestroyLinker(self);
    return(FALSE);
    }
```

Evidently, the ewls_init method needs to be passed the handle of the associated EDWIN instance (in this case, this is stored at DatApp3) and the specified format type. Thereafter the ewls_extract method returns the value that should be written into linksv.len (or a negative number, if the transaction has terminated), and also writes back, to one of the passed parameters, the value for linksv.buf.

The three text formats revisited

The code for EWLINKSV, reproduced below, contains (in conjunction with the code for the ew_bring_in method of EDWIN) what is in effect the *implicit* definition of the three standard link paste text formats:

```
CLASS ewlinksv root
Component of linksv class, extracts data from edwin
    {
    ADD ewls_init
    ADD ewls_extract
    PROPERTY
        {
        VOID *doc;
        WORD state;
        UWORD pos;
        UWORD parend;
        UWORD posend;
        TEXT buf[256];
        }
    }

METHOD INT ewlinksv_ewls_extract(PR_EWLINKSV *self,TEXT **ppb,UINT blen)
    {
    UINT len;
    INT striptabs;
    TEXT *pb;
    TEXT *pbend;

    len=self->ewlinksv.posend-self->ewlinksv.pos;
    if (!len)
        return(-1);     /* finished */
    if (len>=blen)
        len=blen;
    p_send5(self->ewlinksv.doc,O_EP_EXTRACT,self->ewlinksv.pos,&self-
>ewlinksv.buf[0],len);
    if (self->ewlinksv.state!=DF_LINK_PARAS)
        {
        striptabs=(self->ewlinksv.state==DF_LINK_TEXT);
        pb=(&self->ewlinksv.buf[0]);
        for (pbend=pb+len; pb<pbend; pb++)
            {
            if (!*pb)
                {
                self->ewlinksv.pos++;   /* skip the zero too */
                break;
                }
            if (striptabs && *pb=='\t')
                *pb=' ';
            }
        len=pb-(&self->ewlinksv.buf[0]);    /* excludes any trailing zero */
        }
    self->ewlinksv.pos+=len;
    *ppb=(&self->ewlinksv.buf[0]);
    return(len);
    }

METHOD VOID ewlinksv_ewls_init(PR_EWLINKSV *self,PR_EDWIN *edwin,INT state)
    {
    UINT len;

    self->ewlinksv.doc=edwin->edwin.doc;
    len=p_send3(edwin->edwin.scrimg,O_SI_GET_SELECT,&self->ewlinksv.pos);
    self->ewlinksv.posend=self->ewlinksv.pos+len;
    self->ewlinksv.state=state;
    }
```

Native formats

In many cases, applications will wish to support “native” formats of link paste data. For example, the Word Processor link pastes styles and emphasis data along with the text selected. Another example is that the Series 3a Agenda link pastes the details of an appointment - including the alarm setting and memo setting, if any.

In cases like this, the mask specified in the `sy_link_server` message should include the bit for `DF_LINK_NATIVE`.

For example, the ws_background method of the Word Processor is as follows:

```
METHOD VOID wpwserv_ws_background(PR_WPWSERV *self)
    {
    INT type;

    if (self->wserv.flags&PR_WSERV_RECEIVED_KEY)
        {
        type=(1<<DF_LINK_TEXT)+(1<<DF_LINK_TABTEXT)+(1<<DF_LINK_PARAS);
        if (!IsAlias())    /* see whether plain text alias */

type=(1<<DF_LINK_TEXT)+(1<<DF_LINK_TABTEXT)+(1<<DF_LINK_PARAS)+(1<<DF_LINK_NAT
IVE);
        if (!((PR_EDWIN *)HandTWin)->edwin.select)
            type=0;
        p_send4(w_am->appman.system,O_SY_LINK_SERVER,type,0);
        }
    }
```

Note that the Window Server process - which is the central store, on the Series 3 and the Series 3a, for link paste data information - never reports the DF_LINK_NATIVE bit to an application that differs from that which registered the data. (This test is based on the result of calling p_pname.) For this reason, when the DF_LINK_NATIVE bit is set in the mask of available formats, the application can rest assured that the link paste server is indeed the same application as itself (though, possibly, running under a different alias).

By convention, the top eight bits in the 32-bit wide mask of formats are reserved for applications passing more information to each other about which *types* of their own native format are presently available.

Final comments

Note that the link paste server must run *unattended*. There is no point in presenting the user with a query dialog, asking how the link paste service is to proceed. This is because the link paste server is *in background*: the user will not see the dialog, and will just notice that the link paste operation in the foreground window has completely stalled.

If really desired, the link paste server can obtain additional information from its client, by having the client present the query dialog on its behalf. The information required to effect this would have to be defined and included as part of the native format (bear in mind that each partner in the process has access to the PID of the other so that the link paste IPC can be supplemented, if desired, by other forms of IPC at that moment).

One other point that may be worth mentioning is the fact that the link server can happily call p_leave (directly or indirectly) at any stage of its operation. System code (in LINKSV and LINKCL) ensures that the error notification takes place in the client process, and not in the server process.

CHAPTER 15

HWIM RESOURCE FILES

The basic information about resource files that applies to all SIBO applications is covered in the *Resource Files* chapter of the *Additional System Information* manual. That chapter includes:

- the resource file format
- the different possible locations for resource files
- advice about multi-lingual applications
- use of the resource compiler tool *rcomp.exe*
- the allowed content of a *.rss* source file, including `STRUCT` and constand declarations

This chapter contains additional information that is specific to HWIM applications.

The application resource file

As was mentioned in the *Introduction* chapter, all HWIM applications must have an application resource file that contains at least the resources required to construct the application's menu bar and pull-down menus. Simple examples of such resources appear in the *Hello World* and *Commands and Command Menus* chapters.

A further, more realistic, example can be found in the source files for the Record application. In addition to the resources for the application's command menus, the file *record.rss* contains a number of resources of various types. In particular, it `#includes` the file *record.hlp* that contains Help resources.

Resource file location

By default, the application resource file is expected to be built into the application's image file, with the aid of an add-file list (*.afl*) file, in the second of the four possible add-file slots. A resource file in this location will automatically be opened by system code during the initialisation of the application.

An application that wishes to load its resource file from a different location may subclass the `HWIMMAN` application manager, replacing the `am_rscname` method. This method is passed a pointer to a buffer and must write the full file specification of the resource file to this buffer.

For example, an application that has an application resource file with the same name as the application's image file, and resident in the same directory as the image file, could use the following replacement `am_rscname` method:

```
METHOD VOID myappman_am_rscname(PR_MYAPPMAN *self,TEXT *pname)
    {
    p_fparse(".RSC",DatCommandPtr,pname,NULL);
    }
```

A multi-lingual application with a resource file for each of a number of languages could use code similar to that suggested in the *Resource Files* chapter of the *Additional System Information* manual:

```
METHOD VOID myappman_am_rscname(PR_MYAPPMAN *self,TEXT *pname)
    {
    P_INFO f;

    p_atos(pname,"\\app\\archive\\archiv%02d.rsc",p_getlanguage());
    p_fparse(pname,DatCommandPtr,pname,NULL);
    if (p_finfo(pname,&f)<0)
        p_supersend3(self,O_AM_RSCNAME,pname);
    }
```

As explained in that chapter, the resource files are assumed to reside in an application-specific subdirectory of the *\app* directory; in this example the directory is *\app\archive*. The resource file for the default language should be built into the image file so that it is available for use on a machine that is set to a language not supported by the application. If the call to `p_finfo` indicates that a resource file for a particular language is not available, superseding the `AM_RSCNAME` message ensures that the built-in resource file will be used.

See also the description of this method in the chapter *Using the System Components*.

Loading an application resource

Once the resource file is open, any of its resources may be read by sending the application manager either an `AM_LOAD_RESOURCE` or an `AM_LOAD_RES_BUF` message (or by use of the equivalent `hLoadResource` or `hLoadResBuf` utility functions) passing the resource id of the required resource. Note that the resource ids are published in the *.rg* file that is generated during compilation of the source file by *rcomp.exe*.

Resource Structures

The basic resource structures used for strings, dialogs and the standard dialog components are defined in the include file *hwim.rh*. This file should always be `#included` in the application-specific resource file. Application-specific resource structures, if required, will need additional structure definitions. These may appear in-line in the *.rss* source file or may be written in a separate header file, to be `#included` in the source file, together with *hwim.rh*. It is customary to give such a file a *.rh* extension.

The system resource file

The system resource file resides in the ROM, and the (English) source is copied into a *\sibosdk\resource* directory when the OOP option of the SDK is installed, This source consists of the files *s_.rss*, *s_.hlp* and *sx_.ra*. The first of these three is the main source file and includes the other two. The file *s_.hlp* contains the source of the system help resources and *sx_.ra* contains those resources that are specific to the Series 3a. All resources other than the ones in *sx_.ra* are common to both the Series 3 and Series 3a, although a small number of them contain additional items that are not available on the Series 3. All such items are commented.

The system resource file contains text strings, dialogs and other resources that are used by system code. In addition to supplying ready-made resources that can also be loaded by application code, the source files provide a wide range of example templates for constructing application-specific resources.

The source files contain comments that are primarily supplied to aid translators to produce non-English versions. These comments may, however, prove of use to developers by indicating the purpose of, and/or the constraints associated with, a particular resource.

Loading a system resource

A resource is identified as a system resource by specifying a negative resource id. Otherwise, the process of loading a system resource (using, for example, `hLoadResource` or `hLoadResBuf`) is identical to loading an application resource. For example, the following code can be used to allocate memory and load the choice list system resource with resource id `SYS_NO_YES`:

```
TEXT *p;

hLoadResource(-SYS_NO_YES,&p);
```

Note the explicit minus sign.

The resource ids of system resources are published in the include file *s_.h*.

Using system resources

Many system resources are used implicitly by application code. For example, calling `hBusyPrint` implicitly loads the `SYS_BUSY` resource string, and an application that runs one of the system dialogs will cause the associated dialog resource to be loaded from the system resource file.

A number of system resources are suitable for explicit use within an application. Perhaps the most commonly used group is the various string resources that are used as information messages. After a successful copy operation, for example, an application could present confirmation to the user with the code:

```
hInfoPrint(-SYS_COPIED_PROMPT);
```

Other useful system resources include dialog box components such as the various choice lists, particularly `SYS_OFF_ON` and `SYS_NO_YES`, and the general-purpose action lists, for example, `SYS_AC_NO_YES` and `SYS_AC_CONTINUE`. The following extract shows the use of a No/Yes choice list and is suitable for inclusion as an element of an application `DIALOG` resource:

```
...
CONTROL
    {
    class=C_CHLIST;
    prompt="Ignore parity";
    info=CHLIST{rid=-SYS_NO_YES;};
    }
...
```

Caution

In an application it is possible that a system resource could be used in a context that is different from the one for which it was designed. This could cause problems in a multi-lingual application, since translations of system resources may expose differences in context that are not apparent in a single language. System resources should therefore be used cautiously in applications that are intended to run in more than one language. If an application writer has any doubt about the intended use of a system resource, he or she should use an application-specific resource.

Help resources

An application can supply application-specific Help information by means of one or more top-level `HELP_ARRAY` resources. An example of such a resource is as follows:

```
RESOURCE HELP_ARRAY myapp_help
{
topic="Myapp";
topic_id=myapp_help_index;
}
```

It contains a `topic` item, used to construct the title for a Help screen, and a `topic_id` which contains the resource id of a `TOPIC_ARRAY` resource. Such a resource is illustrated below; it contains an `id_lst` array of the resource ids of one or more secondary `HELP_ARRAY` resources:

```
RESOURCE TOPIC_ARRAY myapp_help_index
{
id_lst=
{
basics,
-SYS_HELP_EDIT,
-SYS_HELP_PRINT,
-SYS_HELP_FILES,
-SYS_HELP_NO_SYS_MEM
};
}
```

Note that this array may contain references to any combination of system and application-specific resources, in any order.

A secondary `HELP_ARRAY` resource contains a `topic` item, again used in a title, followed by a `strlist` array of strings, each of which will be displayed on a single line of a Help screen.

```
RESOURCE HELP_ARRAY basics
{
topic="Basics";
strlst=
{
STRING {str="Enter to confirm selection";},
STRING {str="";},
STRING {str="To move around:";},
STRING {str=ARROWS" to move cursor";},
STRING {str="Psion-"<WS_SYMBOL_LEFT_KEY><WS_SYMBOL_RIGHT_KEY>" go to start/end
of line";},
STRING {str="Psion-"<WS_SYMBOL_UP_KEY><WS_SYMBOL_DOWN_KEY>" to PageUp/Down";},
STRING {str="Control-Psion-"<WS_SYMBOL_UP_KEY><WS_SYMBOL_DOWN_KEY>" go to
top/bottom";}
};
}
```

Each line of text, after compilation, must not include more than 39 characters.

Using Help resources

The Help information that is provided when a user presses the Help key may be set within an application by writing the resource id of a top-level `HELP_ARRAY` resource to the window server object's `help_index_id` property. For example, to use the Help data shown above:

```
w_ws->wserv.help_index_id=MYAPP_HELP;
```

Application code may set the Help resource id at any time. If the value of the window server object's `help_index_id` is zero (the default value) system Help will be supplied. Most applications that supply their own Help will normally set the Help resource id during initialisation, in the `ws_dyn_init` method.

An application may provide context-sensitive Help by changing the Help resource id as the application context changes. This may be done either by writing a new value to `w_ws->wserv.help_index_id`, or by subclassing the client window to replace its `wn_sense_help` method.

The Help supplied for a dialog box may be set independently by creating additional sets of Help resources. The Help for a dialog box may be set either by writing to its `helprid` property, or by supplying a replacement `wn_sense_help` method in a `DLGBOX` subclass. Again, default system Help is supplied for dialog boxes.

CHAPTER 16

USING THE SYSTEM COMPONENTS

The application manager and the window server object contain a number of method functions that an application may use to perform common operations.

Some methods can *never* fail and will therefore *never* call `p_leave`. The title line of a number of the more significant methods of this type are marked with a leading ✪ symbol.

The application manager

`HWIMMAN` is a subclass of the OLIB `APPMAN` class and provides the functionality of the HWIM application manager. All HWIM applications create and initialise an instance of (a subclass of) `HWIMMAN` during their initialisation. The handle of this instance is automatically written to the magic static `w_am`.

Subclassers may replace existing methods but, for future compatibility, should avoid adding new methods or property.

A built-in application, or any application which supports only one user language, may not need to subclass `HWIMMAN`. Third party multi-lingual applications will need at least to replace the `am_rscname` method to load the application resource file.

Property

Application manager property is accessible via the `w_am` magic static. Although this means that the property may be accessed from any point in the application code, the property should, except where write access is specifically allowed, be considered as read-only. An application may find it useful to access the following items of property:

`hwimman.command`	the command byte, if any, read from the process command line. The value `H_COMMAND_DEFAULT_FILE` will never appear, since it is converted to one of `H_COMMAND_CREATE_FILE` or `H_COMMAND_OPEN_FILE`, depending on whether the specified file already exists
`hwimman.defext`	a pointer to the application's default file extension, if any, as supplied in the process command line
`hwimman.aliasinfo`	a pointer to the application's alias information, if any, as supplied in the process command line

A command line example

If the heap cell (pointed to by `DatCommandPtr`) containing the command line for a file-based application has, for example, the following content:

```
ROM::WORD.APP<0><0x29>OProgram<0>.OPL OROPO<0>LOC::M:\WRD\MYPROG.OPL<0>
```

then, at the conclusion of system initialisation:

`hwimman.command` is `'O'` (`H_COMMAND_OPEN_FILE`)

`hwimman.defext` points to the string `".OPL"` (the original following space is overwritten by zero)

`hwimman.aliasinfo`	points to the string `"OROPO"`
`DatProcessNamePtr`	points to the string `"Program"`
`DatUsedPathnamePtr`	points to the string `"LOC::M:\WRD\MYPROG.OPL"`
`DatStatusNamePtr`	points to the string `"MYPROG.OPL"`

For an application that is not file-based, and therefore has nothing following the application file name in its command line, all the items in the above list will be set to `NULL`.

See *The Series 3 command line* in the *Communicating with the System Screen* chapter of the *Series 3 Programming Guide* for further information on the contents of the command line.

Usable methods

The methods described in this section are intended to be called explicitly by application code. An application will not normally replace any of these methods.

AM_NEW_FILENAME — Record a new filename

```
VOID am_new_filename(TEXT *newname);
```

Record the new file name pointed to by `newname`. The buffer pointed to by `newname` must remain in existence until the name is changed again.

If the current file name is specified to be in the process command line (`hwimman.contig` is `TRUE`) then the allocated heap cell containing the command line is truncated to remove the name, and `hwimman.contig` is set to `FALSE`.

The magic statics `DatUsedPathNamePtr` and `DatStatusNamePtr` are set up to point to the appropriate positions in the text string.

AM_LOAD_RESOURCE — Load a resource

```
INT am_load_resource(INT resid, UBYTE **ppdata);
```

Allocate a buffer and load into it the resource with id `resid` from the appropriate resource file.

A negative `resid` indicates that the resource is to be found in the system resource file, with an id equal to the absolute value of `resid`. Otherwise it is assumed that the resource is to be found in the application resource file.

HWIM applications must indicate their intention to use both the application and system resource files by setting the `FLG_APPMAN_SRSCFILE` and `FLG_APPMAN_RSCFILE` flags, in `main()`, before creating and initialising the application manager.

The attempt to load the resource may fail due to an out of memory condition, in which case `p_leave(E_GEN_NOMEMORY)` is called.

Any other error when attempting to read an application resource is assumed to be due to the removal of the application resource file. (It is assumed that only the application resource file can be removed, since the system resource file is in the ROM.) An attempt is then made to locate the resource file by sending `AM_FINDIMG` and `AM_RSCNAME` messages. If this is unsuccessful the method calls `p_leave(E_FILE_NXIST)`, otherwise the resource file is reopened - which could itself fail, calling `p_leave(E_GEN_NOMEMORY)`. Following this the resource is loaded which, again, may fail and call `p_leave(E_GEN_NOMEMORY)`.

The method returns the size of the loaded resource.

For more information on resource files, see the *Resource Files* chapter in this manual.

AM_LOAD_RES_BUF — Load a resource to a buffer

```
INT am_load_res_buf(INT resid,UBYTE *pbuf);
```

Load into the buffer pointed to by `pbuf` the resource with id `resid` from the appropriate resource file. There is no memory allocation involved, and thus no likelihood of failure due to an out of memory condition.

A negative `resid` indicates that the resource is to be found in the system resource file, with an id equal to the absolute value of `resid`. Otherwise it is assumed that the resource is to be found in the application resource file.

HWIM applications must indicate their intention to use both the application and system resource files by setting the `FLG_APPMAN_SRSCFILE` and `FLG_APPMAN_RSCFILE` flags, in `main()`, before creating and initialising the application manager.

Any error when attempting to read an application resource is assumed to be due to the removal of the application resource file. (It is assumed that only the application resource file can be removed, since the system resource file is in the ROM.) An attempt is then made to locate the resource file by sending `AM_FINDIMG` and `AM_RSCNAME` messages. If this is unsuccessful the method calls `p_leave(E_FILE_NXIST)`, otherwise the resource file is reopened. Following this, the resource is loaded.

Note that the implementation of the `am_findimg` method means that, in practice, the `am_load_res_buf` method can never fail in an HWIM application.

The method returns the size of the loaded resource.

✪ AM_FINDIMG — Find application image file

```
INT am_findimg(VOID);
```

Refuse to continue until the image file has been located.

If the image file can not be found, it is assumed that this is because the SSD containing it has been removed. The application is suspended by displaying an alert requesting that the SSD be replaced.

It is worth noting that, in consequence, the `am_load_res_buf` method can never fail in an HWIM application.

✪ AM_ADD_TASK — Add a task

```
VOID am_add_task(PR_ACTIVE *hand);
```

Add an initialised active object to the task queue, in priority order, as determined by the active object's `active.priority` field.

The item is added to the list immediately following all existing items with the same (or higher) priority.

AM_YIELD — Wait for all activity to cease

```
VOID am_yield(VOID);
```

Suspend the current action until all active objects with priority greater than `PRIORITY_ACTIVE_COMPUTE` have had the opportunity to service their outstanding events.

See the *OLIB Reference* manual for a description of active objects.

Replaceable methods

The methods described in this section are called by system code and are not intended to be called explicitly by application code. They may be replaced in an application-specific subclass of `HWIMMAN`.

AM_RSCNAME Generate resource file name

```
VOID am_rscname(TEXT *pname);
```

Write, to the buffer pointed to by `pname`, (which must be at least `P_FNAMESIZE` bytes long) the default full file specification (see the *Files* chapter of the *PLIB Reference* manual) of the application resource file.

The name is generated from the application's start-up full file specification, pointed to by the magic static `DatCommandPtr`. The name also depends on the current language, as determined by the value returned by a call to `p_getlanguage`.

If the language is English (`p_getlanguage` returns a value of 1) the resource file is assumed to be built into the image file, so that the full file specification of the resource file is identical to that of the image file.

For all other languages the resource file is assumed to be located in the same directory and have the same name as the application image file, but with a language-dependent file name extension. The extension is assumed to be *.~nn*, where the characters *nn* represent the language number, as two decimal digits. Thus, a German language resource file would have a *.~03* extension.

Note

This method is designed for use by built-in applications, whose resource files are in the `ROM::` device (which does not support subdirectories) and whose default language is English. A multi-lingual application that is run from an SSD should subclass `HWIMMAN` to replace this method. A possible replacement method is described in the *Resource Files* chapter.

✪ AM_NOTIFY Display notifier

```
VOID am_notify(UINT mess1, UINT mess2, UWORD *pbut);
```

Call the `p_notify` service, with text loaded from resource files.

Up to two text messages are specified by the resource ids `mess1` and `mess2`. If `pbut` is `NULL`, the single default button 'CONTINUE' (or the non-English equivalent) will be displayed. Otherwise, `pbut` is assumed to point to an array of three resource ids for the three notifier buttons.

All resource ids follow the resource id rules as specified in the description of the `am_load_resource` method. If any id is `NULL` then no text is loaded for that id.

Once the resource strings are loaded the `p_notify` service is invoked. The allocated space for the resource strings is freed after use.

This method will not fail due to lack of memory. If there is not enough memory available to load any of the specified resources, the corresponding part of the notification text is not displayed.

✪ AM_NOTIFYERR Report an error

```
VOID am_notifyerr(INT err,INT resid);
```

Notify an error for error number `err`, with text as specified by the resource id `resid` providing additional information about the context of the error. If no additional text is required, `resid` may be zero. This method is guaranteed to succeed in displaying an error message.

Cancels any busy indicator by calling `wCancelBusyMsg` and may write zero to the `WSERV` active object's `wserv.filter` property (see the later section on *Keyboard filters*). An application that needs to restore either or both of these to their original state on conclusion of the error report may need to subclass this method.

If `err` has the value `RUN_ACTIVE_CLEANUP_NONOTIFY` the method returns at this point, without displaying any error notification.

Unlike in the superclass `am_notifyerr` method, the `p_notify` service is not used. The method calls the `hErrorDialog` utility function, to attempt to display the error in an error dialog. If this fails due to lack of memory, the error is displayed as an alert, by sending `w_ws` a `WS_ALERT` message. This alert cannot fail, but may not display any context information specified by `resid` if the attempt to load this resource fails.

Using one of these two forms of error display means that the error notification is *application modal*; only that application is suspended and the user may task switch to another application. The superclass method is *system modal*, preventing interaction with *any* application until the user has responded to the error report.

The window server active object

An instance of the WSERV active object class is the application's event source for events (keypresses, redraws, and so on) generated by the window server process. Every HWIM application creates an instance of (a subclass of) WSERV at an early stage in its initialisation and this instance must remain in existence until the application terminates. The handle of this instance is automatically written to the magic static w_ws.

There is no need to send a DESTROY message to the application's instance of WSERV since its resources will be released, along with all other application resources, on termination of the application.

The handle of an application's WSERV object is stored in the magic static w_ws and is therefore available to the whole of the application.

In addition to its main role as the source of window server events, WSERV supplies a number of general services and utilities, including, for example, methods to:

- start up an application-specific dialog
- run a variety of system-supplied dialogs
- evaluate a numeric expression
- word wrap a paragraph of text

An application is expected to subclass WSERV, at least to supply a ws_dyn_init method which should perform all application-specific initialisation.

Subclassers may, where appropriate, replace existing methods but, for future compatibility, should avoid adding new methods or property.

Property

WSERV property is accessible via the w_ws magic static. Although this means that the property may be accessed from any point in the application code, the property should, except where write access is specifically allowed, be considered as read-only. An application may find it useful to access the following items of property:

wserv.com	the handle of the application's command manager, assumed to be an instance of (a subclass of) COMMAN, set by system initialisation code
wserv.bar	NULL if the application is not displaying a menu bar, otherwise the handle of the application's menu bar, set by system code
wserv.cli	the handle of the application's client window, set by application-specific initialisation code (in ws_dyn_init) and modified either directly by application code or by the ws_change_cliwin method
wserv.filter	if not NULL, may be either -1 to discard all incoming keypresses, or the handle of a window to which all keypresses are diverted (see the following section on *Keyboard filters*). An application may write to this property
wserv.filmethod	if not NULL, the method number of the message to be sent to the object specified by wserv.filter (provided this is also not NULL) on receipt of a keypress (see the following section on *Keyboard filters*). Otherwise a WN_KEY message is sent. An application may write to this property
wserv.flags	a collection of state flags, for system use
wserv.help_index_id	either zero or the resource ID of the application's Help index. An application may write to this property

`wserv.lock`	zero if the application is not locked. Otherwise a count of the current number of times the application is locked. The count is incremented and decremented by the `ws_lock` method
`wserv.printer`	the handle of the application's print manager, assumed to be an instance of (a subclass of) the FORM `PRINTER` class, set by the `ws_ens_print_context` method

Keyboard filters

The normal processing of keys can be subverted by setting `wserv.filter` and, optionally, `wserv.filmethod` to non-zero values. The effect differs slightly between the Series 3 and the Series 3a.

Filtering on the Series 3

If `wserv.filter` is set to -1 (0xffff) all keypresses are discarded. Any other non-zero value is assumed to be the handle of an object to which all keypress messages should be redirected, passing both the keycode and the modifiers flags as two parameters. By default the keypress is sent as a `WN_KEY` message but if, in addition, `wserv.filmethod` is non-zero, its value is taken to be the method number of the message to be sent. The behaviour is as indicated by the following code.

```
VOID ProcessKey(INT keycode,INT modifiers)
    {
    INT consumed;

    if (w_ws->wserv.filter)
        {
        if (w_ws->wserv.filter==(PR_WIN *)0xffff)
            return;
        if (w_ws->wserv.filmethod)
            consumed=p_send4(w_ws->wserv.filter,w_ws-
>wserv.filmethod,keycode,modifiers);
        else
            consumed=p_send4(w_ws->wserv.filter,O_WN_KEY,keycode,modifiers);
        if (consumed==WN_KEY_CHANGED)
            return;
        }
    ...
    /* Help, Menu and client window key processing */
    ...
    }
```

The keypress method that is called when a filter is set must return `WN_KEY_CHANGED` to indicate that it has processed the key. Otherwise it should return `WN_KEY_NO_CHANGE`, in which case the keypress will be offered for further processing, as if the filter had not been set. Note that consumption of the key by the filter means that Help and Menu bar interactions are disabled.

If an error occurs while a filter is set, the application manager `am_notifyerr` method clears `wserv.filter` (but not `wserv.filmethod`) otherwise the error notification dialog could not be cancelled by pressing the Esc key. A filter is normally set temporarily, for the duration of a specific process (say, while loading a file) and an error will usually terminate the process. Clearing the filter on an error condition is thus normally a correct action.

In exceptional circumstances an application may require the filter to be maintained across an error condition. This can be assured by replacing the `am_notifyerr` method, as follows:

```
METHOD VOID myappman_am_notifyerr(PR_MYAPPMAN *self,INT err,INT resid)
    {
    PR_WIN *filter;

    filter=w_ws->wserv.filter;
    p_supersend4(self,O_AM_NOTIFYERR,err,resid);
    w_ws->wserv.filter=filter;
    }
```

Filtering on the Series 3a

Filtering on the Series 3a follows the same general pattern as described above for the series 3, except that the `am_notifyerr` method clears both `wserv.filter` and `wserv.filmethod`.

In addition, `wserv.filmethod` may be set to a negative value. In this case the Help key is not filtered and the `am_notifyerr` method will restore the values of `wserv.filter` and `wserv.filmethod` after the error notification is complete. A negative `wserv.filmethod` is still interpreted as a (positive) method number for the purposes of sending a keypress message to the object whose handle is in `wserv.filter`.

A negative value of `wserv.filmethod` is intended to be used in a situation where a filter is set for the duration of an operation that is less transient than, say, the loading of a file. An example of such a use is in the Record application while recording or playing a file (with repeats and trailing silence, playing could last several hours) or while paused before recording (a truly indefinite period).

Usable methods

The methods described in this section are intended to be called explicitly by application code. An application will not normally replace any of these methods.

✪ WS_CHANGE_CLIWIN — Log a new client window

```
PR_WIN *ws_change_cliwin(PR_WIN *whand);
```

Provided that the window with handle `whand` is not already the current client window, record this window as the new client window.

The previous client window, which must still exist, is de-emphasised (it is sent a `WN_EMPHASISE,FALSE` message). The new client window is emphasised (sent a `WN_EMPHASISE,TRUE` message) and is made the foreground window. Its handle is stored in `wserv.cli`.

The method returns the handle of the previous client window.

If the new window handle is the same as the previous one, the method does nothing except to return the client window handle.

WS_DO_DIAL — Run a dialog

```
INT ws_do_dial(HANDLE cat, INT class, DL_DATA *pd);
```

Load, initialise and run the dialog specified by category handle `cat` and class `class`. All HWIM dialogs must be started via this method.

The `DL_DATA` struct is defined in *hwimman.g* as:

```
typedef struct
    {
    UWORD id;           dialog resource id
    VOID *rbuf;         NULL or pointer to dialog result buffer
    PR_DLGBOX **pdlg;   NULL or location to receive dialog handle
    } DL_DATA;
```

Users of this method should note the following:

- the dialog receives, in order, `DL_INIT`, `DL_DYN_INIT`, and `DL_SET_SIZE` messages before being made visible
- the creation and initialisation is protected from out of memory failure, by the use of OLIB `CLEANUP` mechanisms, until the dialog has been made visible. The application does not need to provide any explicit protection against failure unless application initialisation code allocates additional resources
- the dialog handle is not lodged in `*pd->pdlg` until the dialog has been made visible

The return value is zero if the dialog is cancelled by the cancel mechanism provided by system code (that is, without the intervention of application-specific code - see the *Dialogs* chapter for further details). This value is only of significance for dialogs for which the `DLGBOX_NO_WAIT` flag is *not* set.

The method calls `p_leave` (which will trigger the automatic cleanup mechanism) on failure.

WS_WRAP_PARA — Paragraph word wrap

```
INT ws_wrap_para(TEXT *buf, INT len, WRAP_DATA *pd);
```

Word wrap the first `len` characters in the buffer pointed to by `buf`, writing the number of characters in each line to successive bytes of the table pointed to by `pd->ptable` (assumed to be `pd->nlines` bytes long).

The `WRAP_DATA` struct is defined in *hwimman.g* as:

```
typedef struct
    {
    UWORD margin;   width (pixels) to fill with text
    UWORD fmargin;  width (pixels) of first line
    WORD font;      font used
    UWORD style;    style used
    WORD nlines;    max no lines to add to table
    UBYTE *ptable;  table of bytes to receive line lengths
    } WRAP_DATA;
```

Returns the number of lines into which the text has been wrapped. If there are not enough entries in the line-length table, (i.e. if the potential number of lines is greater than `pd->nlines`), then it returns zero.

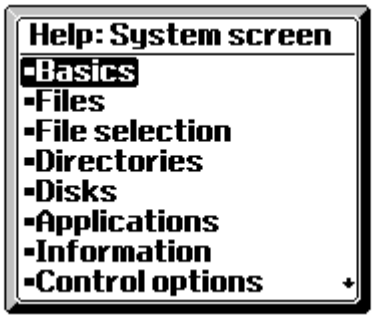
The `margin` field gives the width of each line in pixels. This effectively prescribes the space available to be filled with text.

The `fmargin` field is similar to `margin` but applies only to the first line.

WS_DO_HELP — Run help system

```
VOID ws_do_help(INT start_id);
```

Create, initialise and make visible a help dialog displaying the help information contained in the resource with id `start_id`, assumed to be a `HELP_ARRAY` resource.



This method is normally called by system code, in response to the user pressing the Help key. See the *Resource Files* chapter for more information on Help resources.

WS_LOAD_CHLIST_RES — Get choice list resource text

```
TEXT *ws_load_chlist_res(INT rid, INT nsel, TEXT *buf);
```

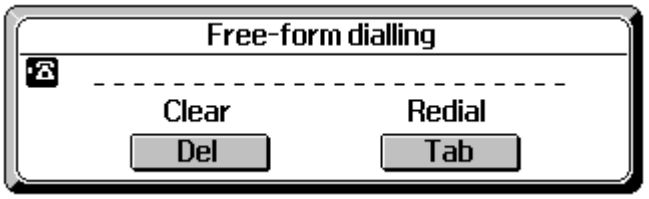
Copy, into the buffer pointed to by `buf`, the zero terminated string that forms choice list item number `nsel` (0 selects the first item) in the choice list resource with id `rid`. This method is somewhat analogous to the more general application manager `am_load_res_buf` method.

It is the user's responsibility to ensure that `rid` refers to a choice list resource and that the buffer is sufficiently long to contain the specified string.

WS_FREE_DIAL — Run the free-form dialling dialog

```
VOID ws_free_dial(VOID);
```

Create, initialise and make visible the free-form dialling dialog:



If the free-form dialog is already present *or* a Help dialog is present, then the method does nothing and returns immediataley.

WS_LOCK — Alter the lock count

```
INT ws_lock(INT lock);
```

Add or remove a level of locking, depending on whether `lock` is `TRUE` or `FALSE`. A locked application does not receive *Shutdown* or *Switchfiles* messages from the system screen.

If `lock` is `TRUE` and the application is currently unlocked, an attempt is made to prevent the receipt of any outstanding window server event (since it may be a *Shutdown* or *Switchfiles* event). If this attempt is too late the incoming event is discarded. The magic static `DatLocked` is set to `TRUE` and `wserv.lock` is incremented.

If the application is already locked, the only action is to increment `wserv.locked`.

If `lock` is `FALSE`, `wserv.lock` is decremented and, if it is decremented to zero, `DatLocked` is set to `FALSE`.

WS_SET_MENUBAR — Set alternative menu bar

```
WSERV_INFO *ws_set_menubar(INT rid);
```

Use `hLoadResource` to allocate a cell and load into it the resource with id `rid` (assumed to be a menu bar resource) writing the address of the cell to `wserv.info`.

Returns a `WSERV_INFO`, pointer to the allocated memory holding the original menu bar resource data. It is the programmer's responsibility to store this value for future restoration of the original menu bar data (normally by use of `ws_reset_menubar`). If, exceptionally, this resource is not to be restored, it may be released with a call to `p_free`.

Note that this method does not make the new menu bar visible, but the new menu will appear when the user next presses the Menu key. There is no straightforward automatic means of forcing the new menu bar to be displayed.

WS_RESET_MENUBAR — Reset the menu bar

```
VOID ws_reset_menubar(WSERV_INFO *info);
```

Free the heap cell containing the current menu bar data and restore the one pointed to by `info`.

It is expected that the pointer is a value that was returned by an earlier use of the `ws_set_menubar` method.

WS_QUERY_DIALOG — Run a query dialog

```
INT ws_query_dialog(INT secondrid, INT rid, INT *pargs);
```

Run the system query dialog, with up to two lines of text.

The first line of text is generated, using `hAtob`, from the format string loaded from the resource with id `rid` and the list of arguments pointed to by `pargs`. The generated text may not exceed 50 characters, including the terminating zero. The value of pargs may be `NULL` if there are no arguments, and `rid` may be zero, in which case there will be no first line text. The second line of text is loaded from the resource with id `secondrid`, which is assumed to be a simple string resource. The value of `secondrid` may be zero, in which case there will be no second line text.

Returns `TRUE` if the user confirms the dialog, otherwise returns `FALSE`.

See also the hConfirm and h2LineConfirm utility functions.

WS_ERROR_DIALOG Run an error dialog

```
INT ws_error_dialog(INT err, INT rid, INT *pargs);
```

Run the system error dialog, with up to two lines of text.

The first line of text is generated, using p_errs, from the error number err. The second line of text is generated, using hAtob, from the format string loaded from the resource with id rid and the list of arguments pointed to by pargs. The generated text may not exceed 50 characters, including the terminating zero. The value of pargs may be NULL if there are no arguments, and rid may be zero, in which case there will be no second line text.

Returns zero, to confirm that the method did not call p_leave. This method is suitable for calling under the protection of p_enter.

See also the hErrorDialog utility function.

WS_EVALUATE Evaluate an expression

```
INT ws_evaluate(TEXT *pResult, TEXT *pExpr, VOID *oplmod);
```

Evaluate the expression contained in the zero terminated string pointed to by pExpr, writing the result to the buffer pointed to by pResult.

The result is evaluated according to the format preferences derived from the evaluator environment variable, *M$V*, accessed via the ws_eval_env method, described below.

If the initial value of the byte *pResult is zero, the evaluation is performed in so-called 'calculator' mode. After evaluation in this mode, the result buffer contains the numerical value, as a DOUBLE, in its first eight bytes. This is followed by the text representation of the value, as a zero terminated string, using the *calc* preferences from *M$V*.

Any initial non-zero value in *pResult causes the evaluation to be performed in 'evaluator' mode. After evaluation in this mode, the result buffer contains only the text representation of the value, as a zero terminated string, using the *eval* preferences from *M$V*.

The expression to be evaluated may be any expression that is acceptable to the OPL programming language. The value of oplmod may be either NULL or a pointer to the name of an OPL module containing additional functions (for example, a set of hyperbolic trigonometrical functions) to be used in evaluating the expression.

Returns -1 if the evaluation is successful. Otherwise reports an error, using hInfoPrintErr, and then returns the byte offset, within the buffer at pExpr, to the point at which the error was detected.

WS_EVAL_ENV Set or get evaluator environment variable

```
VOID ws_eval_env(EXTENDED_MEM_VALUES *pev, INT getit);
```

The *M$V* environment variable stores evaluator format preferences. The method writes from *pev to the *M$V* environment variable or reads from the *M$V* environment variable to *pev.

The EXTENDED_MEM_VALUES struct is defined in *h_eval.h* as:

```
typedef struct
    {
    UBYTE evalFormat;  /* Double to Buffer format code for all except calc */
    UBYTE evalDPlaces; /* Decimal places for all except calc */
    UBYTE calcFormat;  /* Double to Buffer format code */
    UBYTE calcDPlaces; /* Decimal places, if relevant */
    DOUBLE values[MAX_MEMORIES];
    } MEM_VALUES;
```

```
typedef struct
    {
    UBYTE evalDegrees;
    UBYTE calcDegrees;
    MEM_VALUES memVal;
    } EXTENDED_MEM_VALUES;
```

Note that the Series3 and Series3a support two global sets of evaluator preferences, one specifically for the Calculator application (set by the Calculator's *Format* command) and one for all other evaluations (set by the System screen's *"Evaluate" format* command).

If `getit` is `TRUE`, read the contents of the *M$V* environment variable into `*pev`. If the environment variable does not exist it is created with default values as follows:

```
pev->evalDegrees=DEGREES_MODE;
pev->calcDegrees=DEGREES_MODE;
pev->memVal.evalFormat=P_DTOB_FIXED;
pev->memVal.evalDPlaces=2;
pev->memVal.calcFormat=P_DTOB_GENERAL;
pev->memVal.calcDPlaces=12;
p_bfil(&pev->memVal.values[0],MAX_MEMORIES*sizeof(DOUBLE),0);
```

If `getit` is `FALSE`, write `*pev` to the *M$V* environment variable. Note that this process modifies the contents of `*pev`.

The symbol `DEGREES_MODE` is defined in the include file *hwimman.g*

For more information on the conversion of floating point numbers to text and the definition of the sysmbols `P_DTOB_FIXED` and `P_DTOB_GENERAL`, see the `p_dtob` function in the *Floating Point* chapter in the *Plib Reference* manual.

WS_DIAL_ENV Set or get dial environment variable

```
VOID ws_dial_env(DIAL_ENVAR *pev, INT getit);
```

The *D$X* environment variable stores telephone dialling preferences.

The method writes from `*pev` to the *D$X* environment variable or reads from the *D$X* environment variable to `*pev`.

The `DIAL_ENVAR` struct is defined in *dialdlgs.g* as:

```
typedef struct
    {
    UWORD toneLengthTicks;  /*length of each tone in ticks               */
    UWORD delayLengthTicks; /*interval between successive tones in ticks  */
    UWORD pauseLengthTicks; /*length of pause after dial-out code in ticks*/
    UBYTE dialOutCode[6];   /*dial-out-code to access an external line    */
    } DIAL_ENVAR;
```

If `getit` is `TRUE`, read the contents of the *D$X* environment variable into `*pev`. If the environment variable does not exist it is created with default values as follows:

```
pev->toneLengthTicks=8;
pev->delayLengthTicks=8;
pev->pauseLengthTicks=48;
p_scpy(&pev->dialOutCode[0],"9,"); /* from the SYS_DIAL_OUT system resource */
```

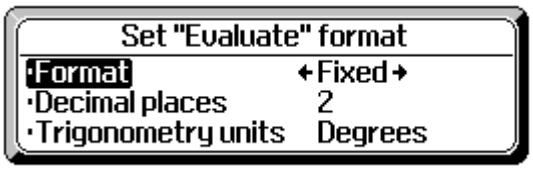
If `getit` is `FALSE`, write `*pev` to the *D$X* environment variable.

Note that a system tick is 1/32 second.

WS_FORMAT_DIALOG Run the set format dialog

```
VOID ws_format_dialog(INT flags);
```

Run the system dialog to set the evaluation format preferences.



If flags is TRUE, run the dialog to set the general evaluate preferences, as shown in the above diagram, otherwise run it to set the preferences for the calculator.

On exiting the dialog, except by pressing Esc, the preferences are written to the *M$V* environment variable. See the description of the EVALDLG system dialog class for further details.

✪ WS_ALERT — Interface to wsAlert

```
VOID ws_alert(TEXT *t1, TEXT *t2);
```

Ensures the magic static DatLocked is TRUE and then call:

```
wsAlertW(WS_ALERT_CLIENT,t1,t2,0,0,0);
```

On return from this call, DatLocked is restored to its original value.

This method is called from the HWIMMAN am_notifyerr method as the final fail-safe stage of the system's error reporting mechanism.

WS_ENS_PRINT_CONTEXT — Ensure print context data exists

```
VOID ws_ens_print_context(VOID);
```

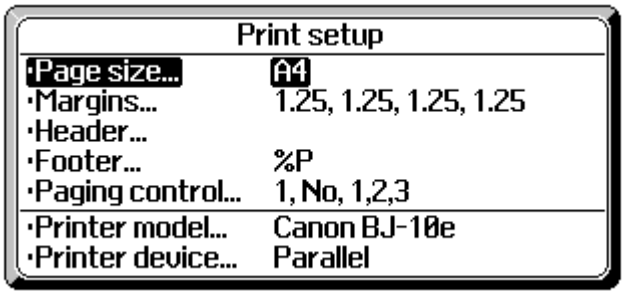
If it does not already exist, create and initialise an instance of the FORM PRINTER class, writing its handle to wserv.printer.

The method does nothing if wserv.printer is not NULL.

WS_EDIT_PRINT_CONTEXT — Run print setup dialog

```
VOID ws_edit_print_context(VOID);
```

Send a WS_ENS_PRINT_CONTEXT message and then run the print setup dialog as, for example, is used by the *Print setup* command in Word's *Special* menu:

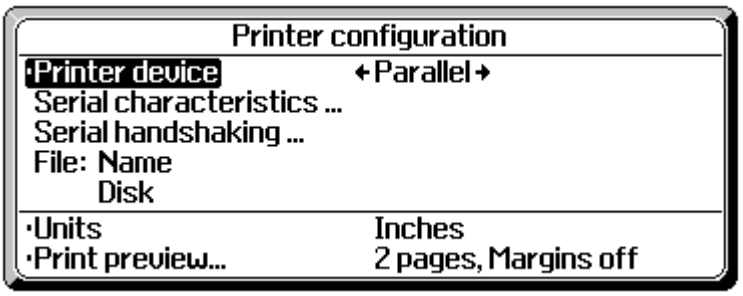


This dialog is used to review and/or modify the print context data stored in the application's instance of the PRINTER class, whose handle is in wserv.printer. See the description of the PRNCTRL system dialog class for further details.

WS_EDIT_PDEV_SETUP — Run printer configuration dialog

```
VOID ws_edit_pdev_setup(VOID);
```

Run the dialog to set up the system-wide printer device configuration, as for the *Printer setup* command in the System Screen's *Special* menu:



See the description of the PDEVDLG system dialog class for further details.

WS_SENSE_PDEV_TEXT Sense text for current printer device

```
VOID ws_sense_pdev_text(TEXT *buf);
```

Write as a zero terminated string, to the buffer pointed to by buf, the text that describes the current printer device.

This text will be one of "Parallel", "Serial" or, if printing to file, the name and extension of the print file.

Replaceable methods

The methods described in this section are called by system code and are not intended to be called explicitly by application code. They may be replaced in an application-specific subclass of WSERV.

WS_DYN_INIT Application-specific initialisation

```
VOID ws_dyn_init(VOID);
```

All HWIM applications must replace this method to provide application-specific initialisation. The supplied method does nothing.

The initialisation must minimally create and initialise a client window, and will generally create and initialise further objects, particularly 'engine' objects that store and provide access to the application's data.

A file-based application should open or create the application's current file, according to the data stored in DatUsedPathnamePtr and HWIMMAN property, as described in the *File-based Applications* chapter.

WS_FOREGROUND Foreground message

```
VOID ws_foreground(UINT flag);
```

The application receives this message, with flag set to TRUE, when it becomes the foreground process. The supplied method does nothing.

The flag parameter is passed so that a subclasser may map the ws_foreground and ws_background methods to a single method function, the two cases being distinguished by the value of flag.

WS_BACKGROUND Background message

```
VOID ws_background(UINT flag);
```

The application receives this message, with flag set to FALSE, when it becomes a background process. The supplied method does nothing.

The flag parameter is passed so that a subclasser may map the ws_foreground and ws_background methods to a single method function, the two cases being distinguished by the value of flag.

CHAPTER 17

HWIM UTILITY FUNCTIONS

HWIM utility and convenience functions have been developed in response to a number of observations and pressures:

- The need to minimise memory used by application code.
- The observation that a large range of applications have many code fragments that are in common use.

By formalising these code fragments into commonly available functions, application code can be made easier to read and can achieve savings in memory.

At first sight, the use of "free standing" functions seems to violate the most basic principles of object oriented programming. This is not the case. They can legitimately be used where similar actions are done repeatedly in various parts of the application code, even if that action involves the sending of messages.

For example, utility functions are very often used to send a standard message to system generated objects such as the application manager which exist throughout the lifetime of an application.

There is nothing to stop a developer from writing his/her own utility functions and is, in fact, encouraged to do so. It is appropriate in a situation where similar pieces of code are used over and over again. It is particularly suitable where code involving the sending of a message to a particular object may be repeated. There is no problem involved in encapsulating the sending of messages within utility functions.

The use of utility functions can save memory by:

- allowing a reduction in the number of parameters passed from application code. While this might save one or two bytes per call, the total saving in memory across a whole range of applications can be substantial
- avoiding the duplication of code fragments

The next sections describe the utility functions available for use by any HWIM application. Each section describes a set of functions grouped under the following headings:

- General utilities
- Text management
- User notification
- Running a dialog
- Dialog box utilities

In many cases, the implementation of utility functions is given in full or in skeleton form. This will assist developers to create their own utility/convenience functions.

Note that the HWIM utility functions are prototyped in *hwim.h*.

General utilities

As indicated by the title of this section, this is a group of miscellaneous functions with no connecting theme.

p_true — Return TRUE

```
INT p_true(VOID);
```

This is a simple function that returns the value `TRUE`.

It is particularly useful as a default method for a class where the method must return a simple `TRUE` or `FALSE` value.

For example, in the definition of the HWIM `comman` class, `p_true` is assigned to the method `com_accl_check` which means that when the method `com_accl_check` is called, `p_true` is executed, thus returning a value of `TRUE`

p_false — Return FALSE

```
INT p_false(VOID);
```

This is a simple function that returns the value `FALSE`.

It is particularly useful as a default method for a class where the method must return a simple `TRUE` or `FALSE` value.

See the earlier description of `p_true` for an example of how this could be implemented.

hDestroy — Destroy an object

```
VOID hDestroy(VOID *hand);
```

This is a function which will destroy an object by sending it the `DESTROY` message:

```
psend2(hand,O_DESTROY);
```

The handle of the object to be destroyed must be passed as a parameter to this function. If the handle is `NULL`, the function does nothing.

hInitVis — Make a window visible

```
VOID hInitVis(VOID *win);
```

This can be used in one of two ways depending on the current state of the window object with handle `win`:

- if the window has not yet been initialised, then it will be both initialised and made visible
- if the window has already been initialised, then it will be made visible.

The function is implemented by sending a `WN_VISIBLE` message to the window with handle `win` as shown below. The handle passed as a parameter must point to an object instanced from the `win` class or a subclass of `win`.

```
psend3(win,O_WIN_VISIBLE,WV_INITVS);
```

For further discussion on windows, see the *Windows* chapter.

hWservComSend — Send command to command manager

```
VOID hWservComSend(INT comid);
```

This utility simply sends the message with message (method) number `comid` and the *same* message (method) number as a *parameter* to the command manager.This is implemented as shown below:

```
p_send3(w_ws->wserv.com,comid,comid);
```

The additional `comid` parameter that is passed with the message is for the convenience of the receiving command manager method, which may choose to ignore it.

This function is particularly useful where a number of methods are implemented using the same code. Passing the method number as a parameter allows the code to identify the context of the call and to take appropriate action if it so wishes.

For further discussion on the command manager, see the *Commands and Command Menus* chapter.

hEnsurePath — Ensure path exists

```
VOID hEnsurePath(TEXT *fname);
```

If the path indicated by the file specification at `fname` does not exist, the function attempts to create the required directory structure.

The file specification is parsed using the Plib function `p_fparse`. If this is successful, the directory component of the parsed file specification is created, provided it does not already exist, using the Plib function `p_mkdir`.

No errors are reported. If this function fails, this could be due to a bad file specification in `*fname` or a problem with the device/medium when attempting to create the directory itself.

For example, suppose `fname` contains the string:

```
"FILES\\DOCS\\FRED.DOC"
```

and the default path is:

```
LOC::M:\
```

then the function will attempt to create the directory structure:

```
\FILES\DOCS\
```

at node LOC:: on device M:

For further information on file specifications see the *files* chapter in the *PLIB Reference* manual.

Text management

This group of functions is concerned with building text from various sources, displaying text in a variety of contexts and with text manipulation.

hLoadResource — Allocate cell and load resource

```
INT hLoadResource(INT rid, VOID *ppdata);
```

This is a convenience routine which allocates a cell of suitable length from the heap, loads the resource referenced by the resource id `rid` into the cell and returns the length of the loaded resource. The address of the cell is placed in `*ppdata`.

The function is implemented by sending an `am_load_resource` message to the application manager as shown below:

```
p_send4(w_am,O_AM_LOAD_RESOURCE,rid,ppdata);
```

Remember that the address of the application manager object is found in the magic static variable `w_am`.

If an error occurs, the function calls `p_leave`.

hLoadResBuf — Load resource into buffer

```
INT hLoadResBuf(INT rid, VOID *buf);
```

This is a convenience routine which loads the resource referenced by the resource id `rid` into the buffer pointed to by `buf` and returns the length of the loaded resource.

The function is implemented by sending an `am_load_res_buf` message to the application manager as shown below:

```
p_send4(w_am,O_AM_LOAD_RES_BUF,rid,buf);
```

It is similar to the previously described function `hLoadresource` except that it is the caller's responsibility to provide the buffer and to ensure that it is large enough to contain the loaded resource.

If an error occurs, the function calls `p_leave`.

As an example, consider the following code fragment that could come from any typical window subclass drawing method which, as part of its functionality, loads a resource string and prints it at a given point within the window:

```
.
.
hLoadResBuf(self->resid,&buf[0]);
p_supersend2(self,O_WN_DRAW);
gPrintText(50,50,&buf[0],p_slen(&buf[0]);
.
.
```

The important point to note here is that `buf` must be large enough to hold the loaded resource string.

hLoadChlistResBuf — Load choice list item into buffer

```
TEXT *hLoadChlistResBuf(INT rid, INT choice, TEXT *buf);
```

This is a convenience routine which loads a *single* choice list item from a choice list resource into a buffer.

The choice list resource is identified by the parameter `rid` within a resource file while the parameter `choice` identifies the particular choice list item *within* the resource.

The function returns a pointer to the string's terminating zero.

The function is implemented by sending a `ws_load_chlist_res` message to the window server object as shown below:

```
return((TEXT *)p_send5(w_ws,O_WS_LOAD_CHLIST_RES,rid,choice,buf);
```

It is the caller's responsibility to ensure that the buffer provided is large enough to contain the loaded choice list item.

If an error occurs, the function calls `p_leave`.

hErrs — Generate error text

```
VOID hErrs(TEXT *buf, INT err);
```

This is a convenience routine that writes the error text specified by the error number `err` to the buffer at `buf`.

If `err` is negative it is interpreted as a system error number and the text is obtained by calling the Plib function `p_errs`; the buffer must be at least `E_MAX_ERROR_TEXT_SIZE` bytes.

If `err` is non-negative it is assumed to refer to a resource string which is loaded from a resource file with resource id `err-ERROR_RID_OFFSET` (see the `WIN` class definition). The buffer supplied must be large enough to contain the maximum size string expected.

Because the resource id is given as `err-ERROR_RID_OFFSET`, then this *must* evaluate to a number greater than 1 in order to access resources within the application resource file. Consequently, `err` must be greater than `ERROR_RID_OFFSET + 1`.

The function is implemented as follows:

```
    .
    .
{
if (err<0)
    p_errs(buf,err);
else
    hLoadResBuf(err-ERROR_RID_OFFSET,buf);
}
    .
    .
```

If an error occurs, the function calls `p_leave`.

For further detail on resource files, see the chapter on *Resource Files* in this manual.

hAtob — Generate formatted string

```
INT hAtob(TEXT *buf, INT rid, INT *pargs);
```

This is a useful function which generates a zero terminated string in the buffer pointed to by `buf` and returns its length. The string is generated from a format string and a number of arguments pointed to by `pargs`. The format string is loaded from the resource `rid` in a resource file.

For more detail on the structure of the format string, see the description of the PLIB function `p_atob` in the *PLIB Reference* manual

There are a number of important points to note:

- the length of the format string in the resource file, when loaded, must *not* be greater than 80 bytes.
- it is the caller's responsibility to ensure that the buffer pointed to by `buf` is large enough to contain the generated string.

If an error occurs, the function calls `p_leave`.

The following routine and code fragment together provide a simple example of the use of this function. For the sake of argument, assume that `file` is a static variable that contains the channel of an opened text file.

```
GLDEF_C CDECL SampleRoutine(INT rid,INT arg,...)
    {
    INT  len;
    TEXT buf[25];
    len = hAtob(&buf[0],rid,&arg);
    p_write(file,&buf[0],len);
    }


    .
    .
INT  rid;
INT  num;
    .
    .
num = 65;
rid = RID_OF_MY_FORMAT_STRING;
SampleRoutine(rid,num,num,num,num,num,num);
    .
    .
```

Given that the format string in the resource file referenced by the resource id `RID_OF_MY_FORMAT_STRING` contains the character sequence:

```
[%b %c %d %o %u %x]
```

then the the following string of characters would be written to the text file:

```
[1000001 A 65 101 65 41]
```

hAtos Generate formatted string, variable argument count

```
TEXT *hAtos(TEXT *buf, INT rid, ...);
```

This is a useful function which generates a zero terminated string in the buffer pointed to by `buf` and returns a pointer to its zero terminator.The string is generated from a format string and a number of arguments. The format string is loaded from the resource `rid` in a resource file while the arguments are interpreted as for the PLIB function `p_atos`.

For more detail on the structure of the format string, see the description of the PLIB functions `p_atob` and `p_atos` in the *PLIB Reference* manual.

As with the function `hAtob`, described earlier, the same important points must be noted. To recap:

- the length of the format string in the resource file, when loaded, must *not* be greater than 80 bytes.
- it is the caller's responsibility to ensure that the buffer pointed to by `buf` is large enough to contain the generated string.

If an error occurs, the function calls `p_leave`.

For example, given that the format string in the resource file referenced by the resource id `RID_OF_MY_FORMAT_STRING` contains the character sequence:

```
%b %c %d %o %u %x
```

the effect of the following code fragment:

```
INT  num;
TEXT buf[23];
INT  rid;
    .
    .
num = 65;
rid = RID_OF_MY_FORMAT_STRING;
hAtos(&buf[0],rid,num,num,num,num,num,num);
```

is to build the string:

```
1000001 A 65 101 65 41
```

in the buffer at `buf`.

hAppendEllipsis Append ellipsis to text

```
VOID hAppendEllipsis(TEXT *p);
```

This is a simple convenience function that writes the ellipsis character, `WS_SYMBOL_ELLIPSIS`, followed by a zero terminator into the two bytes pointed to by `p`.

The ellipsis character is commonly used as an appendage to a choice list prompt to indicate that a subsidiary dialog is available.

For example the code fragment:

```
TEXT  buf[5] = {'A','B','C'};
hAppendEllipsis(&buf[3]);
```

would result in `buf` containing the four characters ABC... followed by the zero terminator.

hSetGTmode Set text mode

```
VOID hSetGTmode(UINT tmode);
```

This simple convenience function uses the window server function `gSetGC` to set the text mode of the *current* graphics context to that specified by `tmode`. Possible values of `tmode` are defined in the *Graphics Output* chapter of the *Window Server Reference* manual.

For example, as part of the drawing method of a `win` subclass, the following code fragment would create a a permanent graphics context with default values and then change the textmode so that when writing text, the 1's and 0's of the font characters overwrite the destination area:

```
        .
        .
    wValidateWin(self->win.id);
    p_supersend2(self,O_WN_DRAW);
    self->mywin.gcid = gCreateGC0(self->win.id);
    hSetGTmode(G_TRMODE_REPL);
    gPrintText(50,50,&buf[0],p_slen(&buf[0]);
    wFree(self->mywin.gcid);
        .
        .
```

hSetGFont — Set font

```
VOID hSetGFont(UINT fid);
```

This simple convenience function uses the window server function `gSetGC` to set the text font of the *current* graphics context to that specified by `fid`. Possible values of `fid` are defined in the *Graphics Output* chapter of the *Window Server Reference* manual.

For example, as part of the drawing method of a `win` subclass, the following code fragment would create a a permanent graphics context with default values and then change the text font to the ROM based monospaced font `WS_FONT_BASE+3`:

```
        .
        .
    wValidateWin(self->win.id);
    p_supersend2(self,O_WN_DRAW);
    self->mywin.gcid = gCreateGC0(self->win.id);
    hSetGFont(WS_FONT_BASE+3);
    gPrintText(50,50,&buf[0],p_slen(&buf[0]);
    wFree(self->mywin.gcid);
        .
        .
```

hSetGStyle — Set text style

```
VOID hSetGStyle(UINT style);
```

This simple convenience function uses the window server function `gSetGC` to set the text style of the *current* graphics context to that specified by `style`. Possible values of `style` are defined in the *Graphics Output* chapter of the *Window Server Reference* manual.

For example, as part of the drawing method of a `win` subclass, the following code fragment would create a a permanent graphics context with default values and then change the text style so that text characters are displayed in bold:

```
        .
        .
    wValidateWin(self->win.id);
    p_supersend2(self,O_WN_DRAW);
    self->mywin.gcid = gCreateGC0(self->win.id);
    hSetGStyle(G_STY_BOLD);
    gPrintText(50,50,&buf[0],p_slen(&buf[0]);
    wFree(self->mywin.gcid);
        .
        .
```

hGetBWid — Get normal text width for buffer

```
INT hGetBWid(TEXT *buf, INT len);
```

This function uses the window server function `gTextWidth` to calculate and return the width, in pixels, of the first `len` characters in the buffer pointed to by `buf`. This function assumes that the text would be displayed using the current system font (i.e `SYSTEM_FONT_ID`) and a normal style (i.e `G_STY_NORMAL`).

This is useful for planning the size and positioning of text in relation to surrounding text and graphic objects.

hGetSWid — Get normal text width for string

```
INT hGetSWid(TEXT *pzts);
```

This is similar to the function `hGetBWid`.

It calculates and returns the width, in pixels, of the zero terminated string pointed to by `pzts`. Like `hGetBWid`, this function assumes that the text would be displayed using the current system font (i.e `SYSTEM_FONT_ID`) and a normal style (i.e `G_STY_NORMAL`).

This function is useful for planning the size and positioning of text in relation to surrounding text and graphic objects.

hGetBBWid — Get bold text width for buffer

```
INT hGetBBWid(TEXT *buf, INT len);
```

This is similar to the function `hGetBWid` in that it uses the window server function `gTextWidth` to calculate and return the width, in pixels, of the first `len` characters in the buffer pointed to by `buf`. This function assumes that the text would be displayed using the font with id `BOLD_FONT_ID` and a normal style (i.e `G_STY_NORMAL`).

This is useful for planning the size and positioning of text in relation to surrounding text and graphic objects.

User notification

This group of functions is concerned with informing, warning and alerting the user in a variety of ways.

hInfoPrint — Display an information message

```
VOID hInfoPrint(INT rid, ...);
```

This function prints an information message window in the bottom right hand corner of the screen for 2 to 2.5 seconds or until cancelled.

The zero terminated text of the message is generated from a format string and the arguments which follow the parameter `rid`. The format string is loaded from the resource `rid` in a resource file.

For more detail on the structure of the format string, see the description of the PLIB function `p_atob` in the *PLIB Reference* manual

There are a number of important points to note:

- the length of the format string in the resource file, when loaded, must *not* be greater than 80 bytes
- if the generated text is greater than `W_INFO_MSG_MAX_LEN` bytes then the message will appear truncated
- the generated text including the zero terminator must not, in any event, be greater than 80 bytes.

If an error occurs, `p_leave` is called.

The following code fragment provides a simple example of the use of this function.

The string in the resource file referenced by the resource id `RID_OF_MY_FORMAT_STRING` contains the 25 character sequence:

```
The numbers are %d and %d
```

the effect of the following code fragment:

```
INT  num1,num2;
INT  rid;
    .
    .
num1 = 65;
num2 = 66;
rid = RID_OF_MY_FORMAT_STRING;
hInfoPrint(rid,num1,num2);
```

is to display the message:

```
The numbers are 65 and 66
```

in the bottom right hand corner of the screen.

hInfoPrintErr Display an error information message

```
VOID hInfoPrintErr(INT err);
```

This function prints an error information message window in the bottom right hand corner of the screen for 2 to 2.5 seconds or until cancelled.

The text of the message is related to the error number in `err` and is obtained in exactly the same way as described in the function `hErrs`; see the description of `hErrs` for more information.

There are a number of extra points to note:

- if the generated text is greater than `W_INFO_MSG_MAX_LEN` bytes then the message will appear truncated
- the generated text including the zero terminator must *not*, in any event, be greater than 80 bytes.

If an error occurs, `p_leave` is called.

hBusyPrint Display a busy message

```
VOID hBusyPrint(INT delay, INT rid, ...);
```

This function prints a flashing message window in the bottom left hand corner of the screen. The display of the message can be delayed by the number of *half-seconds* specified in the `delay` parameter; this can range from 0 to 63.

The zero terminated text of the message is generated from a format string and the arguments which follow the parameter `rid`. The format string is loaded from the resource `rid` in a resource file.

The message may be removed by a call to the window server function `wCancelBusyMsg`.

For more detail on the structure of the format string, see the description of the PLIB function `p_atob` in the *PLIB Reference* manual

There are a number of points to note:

- the generated text including the zero terminator must *not* be greater than 30 bytes. (Note that this is less than the 80 bytes common for other functions)
- the length of the format string in the resource file, when loaded, must *not* be greater than 80 bytes

If an error occurs, `p_leave` is called.

A common use of a flashing message area in the bottom left hand corner of the screen is to display a "busy" type messge to inform the user that the work in progress could take some time to complete.

hBeep Make a beep

```
VOID hBeep(VOID);
```

This is a convenience function that makes a short beep sound suitable for accompanying an error notification. It can, of course, be used wherever an application sees fit.

The function is implemented by making the following call:

```
p_sound(-5,320);
```

The beep is sounded for a duration of 5 system ticks (that is 5/32 sec) at a frequency of 512/320 KHz (that is 1.6 KHz).

Run a dialog

This group of functions is concerned with starting some standard dialogs and launching dialogs in general.

hLaunchDial — Launch a dialog

```
INT hLaunchDial(P_CATID cat, INT class, DL_DATA *data);
```

This is a convenience routine which loads, initialises and runs a dialog. The parameter `cat` is the category *number* of the dialog class while the parameter `class` is its class number.

The `DL_DATA` struct is defined in *hwimman.g* as:

```
typedef struct
    {
    UWORD id;           /* resource id of a DIALOG resource*/
    VOID *rbuf;         /* address of result buffer, or NULL */
    PR_DLGBOX **pdlg;   /* address of where to write handle of dialog, or NULL
*/
    } DL_DATA;
```

The function is implemented as shown below by sending a `O_WS_DO_DIAL` message to the `WSERV` object.

```
p_send5(w_ws,O_WS_DO_DIAL,p_getlibh(cat),class,data);
```

`hLaunchDial` returns whatever value that the method `O_WS_DO_DIAL` returns. This will be zero for dialogs cancelled without the intervention of application code.

hConfirm — Run a confirm dialog

```
INT hConfirm(INT rid,...);
```

This is a convenience routine which presents a standard one line query dialog which returns a `TRUE` or `FALSE` value. This type of dialog is often used to ask a user to confirm an intended action.

The text of the dialog is generated from a format string and (optionally) a number of arguments. The format string is loaded from the resource `rid` in a resource file while the arguments (if any) are interpreted as for the PLIB function `p_atos`. More information on `p_atos` can be found in the *PLIB Reference* manual.

The function is implemented as shown below by sending a `O_WS_QUERY_DIALOG` message to the `WSERV` object.

```
p_send5(w_ws,O_WS_QUERY_DIALOG,0,rid,&rid+1);
```

`hConfirm` returns whatever value that the method `O_WS_QUERY_DIALOG` returns. This will be `TRUE` if the user confirms the action.

There are a number of points to note:

- the length of the format string in the resource file, when loaded, must *not* be greater than 80 bytes.
- the resulting text string must not be greater than 100 bytes.

If the resulting text string is sufficiently large to force the width of the resulting dialog box to exceed the width of the window, then a panic will result.

h2LineConfirm — Run a two-line confirm dialog

```
INT h2LineConfirm(INT secondrid, INT rid, ...);
```

This is a convenience routine which presents a two line query dialog which returns a `TRUE` or `FALSE` value. This type of dialog is often used to ask a user to confirm an intended action.

The function is very similar to the function `hConfirm` described earlier. The first line of text of the dialog is generated from a format string and (optionally) a number of arguments. The format string is loaded from the resource `rid` in a resource file while the arguments (if any) are interpreted as for the PLIB function `p_atos`. The second line of text is basic text and is loaded from the resource `secondrid` in a resource file.

The function is implemented as shown below by sending a `O_WS_QUERY_DIALOG` message to the `WSERV` object.

```
p_send5(w_ws,O_WS_QUERY_DIALOG,secondrid,rid,&rid+1);
```

`h2LineConfirm` returns whatever value that the method `O_WS_QUERY_DIALOG` returns. This will be `TRUE` if the user confirms the action.

There are a number of points to note:

- the length of the format string in the resource file, when loaded, must *not* be greater than 80 bytes.
- the resulting text string must not be greater than 100 bytes.
- the second line of text in the resource file must not be greater than 100 bytes.

If either of the resulting text strings is sufficiently large to force the width of the resulting dialog box to exceed the width of the window, then a panic will result.

hErrorDialog — Run an error dialog

```
INT hErrorDialog(INT err, INT rid, ...);
```

This is a convenience routine which presents an error dialog with text derived from the error number and a resource file. It returns a zero if the dialog was presented successfully. It returns a non-zero value if the dialog presentation failed due to an out of memory error, in which case the dialog remains to be cleaned up (use the OLIB convenience function `cl_clean_level`. For further information, see *The CLEANUP Class* chapter in the *OLIB Reference* manual.).

The text derived from the resource file is generated from a format string and (optionally) a number of arguments. The format string is loaded from the resource `rid` in a resource file while the arguments (if any) are interpreted as for the PLIB function `p_atos`.

The function is implemented as shown below by sending a `O_WS_ERROR_DIALOG` message to the `WSERV` object.

```
p_send5(w_ws,O_WS_ERROR_DIALOG,err,rid,&rid+1);
```

There are a number of points to note:

- the length of the format string in the resource file, when loaded, must *not* be greater than 80 bytes.
- the text string resulting from the resource file must not be greater than 80 bytes.

This type of dialog is often used for the reporting of errors of a moderately serious nature; minor errors should use `hInfoPrintError` described earlier and more serious errors should use the application manager's `am_notify` method.

Dialog box utilities

The following utility functions are available for performing standard operations on the component controls of a dialog. They may be used directly by an application, or used as models for the construction of application-specific utilities.

The code of each used utility function is included in the application. These functions have optimised calling conventions and therefore offer a modest saving over the equivalent function supplied by the application itself.

Note that these utilities assume that the dialog handle is stored in the magic static `DatDialogPtr`. Thus they may not be used with any dialog which contains the `DLGBOX_NO_DDP` flag in `dlgbox.flags`.

hDlgSet — Set an item

```
VOID hDlgSet(UBYTE index, VOID *pset);
```

This function is a generalised way of setting data (or property) into the control of one of the components of the current dialog and is used in the more specialised dialog box utility functions described later.

As stated in the introduction to this section, it is assumed that `DatDialogPtr` points to the current dialog object.

The particular component within the dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0). The parameter `pset` is assumed to point to the appropriate structure containing the information to be set . The class which defines the referenced dialog box component *control*, will have defined a `O_WN_SET` method and will "know" how to interpret the data pointed to by `pset`.

The function is implemented by sending a `O_WN_SET` message to the *current dialog object* as shown below:

```
p_send4(DatDialogPtr,O_WN_SET,index,pset);
```

hDlgSetText — Set a text item

```
VOID hDlgSetText(UBYTE index, TEXT *buf);
```

This function sets text into the control of one of the current dialog's components. The control is assumed to be an instance of `TEXTWIN` or a subclass.

The particular component within the dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0). The parameter `buf` is assumed to point to a zero terminated string containing the text to be set.

This function is implemented as shown below and note that it uses the more generalised function `hDlgSet` described earlier.

```
GLDEF_C VOID hDlgSetText(UBYTE index, TEXT *buf)
    {
    SE_TEXTWIN settxt;

    settxt.buf=buf;
    settxt.len=p_slen(settxt.buf);
    settxt.flags=SE_TEXTWIN_TEXT;
    hDlgSet(index,&settxt);
    }
```

hDlgSetEdwin — Set text in edit window

```
VOID hDlgSetEdwin(UBYTE index, Text *buf);
```

This function sets text into the control of one of the current dialog's components. The control is assumed to be an instance of `EDWIN` or a subclass.

The particular component within the dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0). The parameter `buf` is assumed to point to a zero terminated string containing the text to be set.

This function is implemented as shown below and note that it uses the more generalised function hDlgSet described earlier.

```
GLDEF_C VOID hDlgSetEdwin(UBYTE index, TEXT *buf)
    {
    SE_EDWIN set;

    set.buf=buf;
    set.len=p_slen(set.buf);
    hDlgSet(index,&set);
    }
```

hDlgSetPrompt Set text in prompt window

```
VOID hDlgSetPrompt(UBYTE index Text *buf);
```

This function sets text into the prompt of one of the current dialog's components.

The particular component within the dialog is identified by the index parameter (the first component is identified by an index value of 0). The parameter buf is assumed to point to a zero terminated string containing the prompt text to be set.

This function is implemented as shown below and it may be of interest to know that the O_DL_SET_PROMPT method uses the more generalised function hDlgSet described earlier.

```
GLDEF_C VOID hDlgSetPrompt(UBYTE index, TEXT *buf)
    {
    SE_TEXTWIN settxt;

    settxt.buf=buf;
    settxt.len=p_slen(settxt.buf);
    settxt.flags=SE_TEXTWIN_TEXT;
    p_send4(DatDialogPtr,O_DL_SET_PROMPT,index,&settxt);
    }
```

hDlgSetChlist Set choice in choice list

```
VOID hDlgSetChlist(UBYTE index, INT nsel);
```

This function sets the choice number in a choice list control within one of the current dialog's components, ultimately causing that item in the list to be displayed (or highlighted if the whole list is shown). The parameter nsel holds the number of the choice to be set (where a value of 0 refers to the first item in the choice list).

The particular component within the current dialog is identified by the index parameter (the first component is identified by an index value of 0) and is assumed to be an instance of (or a subclass of) CHLIST.

This function is implemented as shown below and note that it uses the more generalised function hDlgSet described earlier.

```
GLDEF_C VOID hDlgSetChlist(UBYTE index, INT nsel)
    {
    SE_CHLIST set;

    set.nsel=nsel;
    set.set_flags=SE_CHLIST_NSEL;
    hDlgSet(index,&set);
    }
```

hDlgSetChlistOn Set On/Off choice list to On

```
VOID hDlgSetChlistOn(UBYTE index);
```

This function is a specialised version of the function hDlgSetChlist described earlier.

It sets the choice number in a choice list control to 1.

The particular component within the current dialog is identified by the index parameter (the first component is identified by an index value of 0) and is assumed to be an instance of CHLIST or a subclass.

This function can be used with any choice list but is especially useful if used in conjunction with an Off/On choice list (using the system resource `SYS_OFFON_MENU`). If the current dialog has a component containing this choice list, then using this function (with the appropriate `index`) will set the choice list to On.

This function is implemented as shown below and note that it uses the more generalised function `hDlgSetChlist` described earlier.

```
GLDEF_C VOID hDlgSetChlistOn(UBYTE index)
    {
    hDlgSetChlist(index,1);
    }
```

hDlgSetNcedit — Set value of numeric editor

```
VOID hDlgSetNcedit(UBYTE index, UINT value);
```

This function sets a value into the control of one of the current dialog's components. The control is assumed to be an instance of the numeric editor (`NCEDIT`) or a subclass.

The particular component within the dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0).

The value passed to the function in the parameter `value` can be any valid `UINT` type.

This function is implemented as shown below and note that it uses the more generalised function `hDlgSet` described earlier.

```
GLDEF_C VOID hDlgSetNcedit(UBYTE index, UINT value)
    {
    SE_NCEDIT set;

    set.value=value;
    set.flags=SE_NCEDIT_VALUE;
    hDlgSet(index,&set);
    }
```

hDlgSetLledit — Set latitude/longitude editor

```
VOID hDlgSetLledit(UBYTE index, INT value);
```

This function sets a value into the control of one of the current dialog's components. The control is assumed to be an instance of the latitude/longitude editor (`LLEDIT`) or a subclass. Whether the dialog control represents a latitude or a longitude depends on the way the editor is initialised.

The particular component within the dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0).

The `value` passed to the function should represent the number of *minutes* of latitude or longitude. A positive value represents the number of minutes *North* or *West* while a negative value represents the number of minutes *South* or *East*, respectively .

For a latitude, the magnitude of the value should be no greater than 5399 minutes (that is, 89 degrees and 59 minutes).For a longitude, the magnitude of the value should be no greater than 10799 minutes (that is, 179 degrees and 59 minutes). Larger values may be passsed but the editor will assume the appropriate maximum value.

This function is implemented as shown below and note that it uses the more generalised function `hDlgSet` described earlier.

```
GLDEF_C VOID hDlgSetLledit(UBYTE index, UINT value)
    {
    SE_LLEDIT set;

    set.value=value;
    hDlgSet(index,&set);
    }
```

For example, if the third dialog component in the current dialog box contains a longitude editor control and the fourth component contains a latitude editor control, then the code fragment:

```
        .
        .
    hDlgSetLledit(2,5110);
    hDlgSetLledit(3,-2010);
        .
        .
```

will set the respective editor values to:

85^0 10' West and 33^0 30' South.

hDlgSetPtedit — Set punctuation editor

```
VOID hDlgSetPtedit(UBYTE index, UBYTE ch);
```

This function sets a value into the control of one of the current dialog's components. The control is assumed to be an instance of the punctuation editor (`PUNCTUED`) or a subclass.

The particular component within the dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0).

The function allows the single character, passed in the parameter `ch`, to be set. Although an instance of a `PUNCTUED` class restricts user-keyed input to a valid punctuation character, the value passed in `ch` can be any valid character acceptable to `EDWIN`, the immediate superclass of `PUNCTUED`.

This function is implemented as shown below and note that it uses the more generalised function `hDlgSetEdwin` described earlier.

```
GLDEF_C VOID hDlgSetPtedit(UBYTE index, UBYTE ch)
    {
    INT chx;

    chx=ch;
    hDlgSetEdwin(index,(TEXT *)(&chx));
    }
```

For example, if the *third* dialog component in the current dialog box contains a punctuation editor control, then the following call would set the punctuation character to a comma:

```
        .
        .
    hDlgSetPtedit(2,',');
        .
        .
```

hDlgSetFledit — Set floating point editor

```
VOID hDlgSetFledit(UBYTE index, DOUBLE *pvalue);
```

This function sets a value into the control of one of the current dialog's components. The control is assumed to be an instance of the floating point editor (`FLEDIT`) or a subclass.

The particular component within the dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0).

The function allows the floating point number, pointed to by the parameter `pvalue`, to be set. This must be a *valid* double value otherwise the function does nothing.

This function is implemented as shown below and note that it uses the more generalised function `hDlgSet` described earlier.

```
GLDEF_C VOID hDlgSetFledit(UBYTE index, DOUBLE *pvalue)
    {
    SE_FLTEDIT set;

    p_fld(&set.current,pvalue);
    set.set_flags=SE_FLTEDIT_CURRENT;
    hDlgSet(index,&set);
    }
```

hDlgSetDtedit — Set date editor

```
VOID hDlgSetDtedit(UBYTE index, ULONG value);
```

This function sets a value into the control of one of the current dialog's components. The control is assumed to be an instance of the date editor (`DTEDIT`) or a subclass.

The particular component within the dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0).

The function allows the value passed in the parameter `value`, to be set. This is assumed to be in system time format, that is, the number of seconds since 00:00:00, January 1st, 1970.

This function is implemented as shown below and note that it uses the more generalised function `hDlgSet` described earlier.

```
GLDEF_C VOID hDlgSetDtedit(UBYTE index, ULONG value)
    {
    SE_DTEDIT set;

    set.value=value;
    set.flags=SE_DTEDIT_VALUE;
    hDlgSet(index,&set);
    }
```

hDlgSetRgedit — Set range editor

```
VOID hDlgSetRgedit(UBYTE index, UINT value1, UINT value2);
```

This function sets values into the control of one of the current dialog's components. The control is assumed to be an instance of the range editor (`RGEDIT`) or a subclass.

The particular component within the dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0).

The function allows the upper and lower limits for the range passed in the parameters `value1` and `value2` respectively, to be set. The range editor is a subclass of the `MFNE` (multiple field numeric editor) class .

This function is implemented as shown below and note that it uses the more generalised function `hDlgSet` described earlier.

```
GLDEF_C VOID hDlgSetRgedit(UBYTE index, UINT value1, UINT value2)
    {
    SE_RGEDIT set;

    set.value[IX_RGEDIT_VALUE_1]=value1;
    set.value[IX_RGEDIT_VALUE_2]=value2;
    set.flags=SE_RGEDIT_VALUE_1|SE_RGEDIT_VALUE_2;
    hDlgSet(index,&set);
    }
```

The symbols `IX_RGEDIT_VALUE_1` and `IX_RGEDIT_VALUE_2` are defined in the include file *rgedit.g*.

hDlgSetTitleByRid — Set dialog title

```
VOID hDlgSetTitleByRid(INT rid);
```

The function sets the title of the current dialog box to the zero terminated text referenced by the resource id `rid` in a resource file.

This function is implemented as shown below. The title of a dialog box is always the first component of that dialog box and, therefore, is always referenced by an index value of zero as demonstrated in the implementation of `hDlgSetTitlebyRid`, shown below.

```
GLDEF_C VOID hDlgSetTitleByRid(INT rid)
    {
    TEXT buf[60];

    hLoadResBuf(rid,&buf[0]);
    hDlgSetText(0,&buf[0]);
    }
```

A few points to note:

- the appropriate text string in the resource file, when loaded, cannot be greater than 60 bytes.
- if the text string is sufficiently large to force the width of the resulting dialog box to exceed the width of the window, then a panic will result.

hDlgSense — Sense an item

```
VOID hDlgSense(UBYTE index, VOID *psense);
```

This function is a generalised way of sensing or retrieving data (or property) from the control of one of the components of the current dialog and is used in the more specialised dialog box utility functions described later.

As stated in the introduction to this section, it is assumed that `DatDialogPtr` points to the current dialog object.

The particular component within the dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0). The parameter `psense` is assumed to point to the appropriate structure in memory into which the information is to be placed. The class which defines the referenced dialog box component *control* will have defined a `O_WN_SENSE` method and will "know" how to interpret the data structure pointed to by `psense`.

The function is implemented by sending a `O_WN_SENSE` message to the *current dialog object* as shown below:

```
p_send4(DatDialogPtr,O_WN_SENSE,index,psense);
```

hDlgSenseEdwin — Sense edit window

```
TEXT *hDlgSenseEdwin(UBYTE index);
```

This function senses the text from the control of one of the current dialog's components. The control is assumed to be an instance of `EDWIN` or a subclass.

The particular component within the dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0).

The function returns a pointer to a zero terminated string representing the text in the edit window.

This function is implemented as shown below and note that it uses the more generalised function `hDlgSense` described earlier.

```
GLDEF_C TEXT *hDlgSenseEdwin(UBYTE index)
    {
    SE_EDWIN sense;

    hDlgSense(index,&sense);
    return(sense.buf);
    }
```

For example the following code fragment copies the text sensed from an edit window into another buffer; the example assumes that the edit window is contained in the second component in the current dialog box:

```
        .
        .
    TEXT buf[100];
        .
        .
    p_scpy(&buf[0],hDlgSenseEdwin(1));
        .
        .
```

hDlgSenseChlist — Sense a choice list

```
INT hDlgSenseChlist(UBYTE index);
```

This function senses the choice number of the currently highlighted/selected choice item in a choice list control in a component of the current dialog and returns that value. A value of 0 refers to the first item in the choice list.

The particular component within the current dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0). The control is assumed to be an instance of `CHLIST` or a subclass.

This function is implemented as shown below and note that it uses the more generalised function `hDlgSense` described earlier.

```
GLDEF_C INT hDlgSenseChlist(UBYTE index)
    {
    SE_CHLIST sense;

    hDlgSense(index,&sense);
    return(sense.nsel);
    }
```

For example, the following code fragment sets text in an edit window depending on the current item selected in the choice list; the choice list and the edit window being (for the sake of the example) in the current dialog:

```
        .
        .
    #define CHOICE_TEXT_YES 1
    #define CHOICE_TEXT_NO  2
        .
        .
    TEXT buf[2];
        .
        .
    buf[1] = '\0';
    if (hDlgSenseChlist(2) == CHOICE_TEXT_YES)
        buf[0] = 'Y';
    else
        buf[0] = 'N';
    hDlgSetText(3,&buf[0]);
```

hDlgSenseNcedit — Sense a numeric editor

```
UINT hDlgSenseNcedit(UBYTE index);
```

This function senses the value from the control of one of the current dialog's components. The control is assumed to be an instance of the numeric editor (`NCEDIT`) or a subclass.

The particular component within the dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0).

The function returns the value currently held by the editor.

This function is implemented as shown below and note that it uses the more generalised function `hDlgSense` described earlier.

```
GLDEF_C UINT hDlgSenseNcedit(UBYTE index)
    {
    SE_NCEDIT sense;

    hDlgSense(index,&sense);
    return(sense.value);
    }
```

hDlgSenseRgedit Sense a range editor

```
VOID hDlgSenseRgedit(UBYTE index, UWORD *pvalues);
```

This function senses values from the control of one of the current dialog's components. The control is assumed to be an instance of the range editor (`RGEDIT`) or a subclass.

The particular component within the dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0).

The function senses the upper and lower limits of the range and places the two values into the first two words of memory pointed to by `pvalues`. It is the caller's responsibility to supply the two words of memory. The range editor is a subclass of the `MFNE` (multiple field numeric editor) class .

This function is implemented as shown below and note that it uses the more generalised function `hDlgSense` described earlier.

```
GLDEF_C VOID hDlgSenseRgedit(UBYTE index, UWORD *pvalues)
    {
    SE_RGEDIT sense;

    hDlgSense(index,&sense);
    *pvalues++=sense.value[IX_RGEDIT_VALUE_1];
    *pvalues=sense.value[IX_RGEDIT_VALUE_2];
    }
```

The symbols `IX_RGEDIT_VALUE_1` and `IX_RGEDIT_VALUE_2` are defined in the include file *rgedit.g*.

hDlgSenseFledit Sense a floating point editor

```
VOID hDlgSenseFledit(UBYTE index, DOUBLE *pvalue);
```

This function senses the value from the control of one of the current dialog's components. The control is assumed to be an instance of the floating point editor (`FLEDIT`) or a subclass.

The particular component within the dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0).

The function senses the floating point value and places it in the memory pointed to by `pvalue`. It is the caller's responsibility to supply this memory location.

This function is implemented as shown below and note that it uses the more generalised function `hDlgSet` described earlier.

```
GLDEF_C VOID hDlgSenseFledit(UBYTE index, DOUBLE *pvalue)
    {
    hDlgSense(index,pvalue);
    }
```

hDlgSenseLledit Sense a latitude/longitude editor

```
INT hDlgSenseLledit(UBYTE index);
```

This function senses a value from the control of one of the current dialog's components. The control is assumed to be an instance of the latitude/longitude editor (`LLEDIT`) or a subclass. Whether the dialog component represents a latitude or a longitude depends on the way the editor is initialised.

The particular component within the dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0).

The function returns the number of *minutes* of latitude or longitude. A positive value represents the number of minutes *North* or *West* while a negative value represents the number of minutes *South* or *East* respectively.

For example, a returned latitude value of -2010 means 33^0 30' South while a returned longitude value of 5110 means 85^0 10' West.

This function is implemented as shown below and note that it uses the more generalised function `hDlgSet` described earlier.

```
GLDEF_C INT hDlgSenseLledit(UBYTE index)
    {
    SE_LLEDIT sense;

    hDlgSense(index,&sense);
    return(sense.value);
    }
```

hDlgSensePtedit — Sense a punctuation editor

```
INT hDlgSensePtedit(UBYTE index);
```

This function senses a value from the control of one of the current dialog's components. The control is assumed to be an instance of the punctuation editor (`PUNCTUED`) or a subclass.

The particular component within the dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0).

The function returns two bytes: the character currently contained within the punctuation editor and a terminating zero

This function is implemented as shown below and note that it uses the more generalised function `hDlgSenseEdwin` described earlier.

```
GLDEF_C UINT hDlgSensePtedit(UBYTE index)
    {
    return(*hDlgSenseEdwin(index));
    }
```

hDlgSenseDtedit — Sense a date editor

```
ULONG hDlgSenseDtedit(UBYTE index);
```

This function senses a value from the control of one of the current dialog's components. The control is assumed to be an instance of the date editor (`DTEDIT`) or a subclass.

The particular component within the dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0).

The function returns the current value of the date editor. The returned value is in system time format, that is, the number of seconds since 00:00:00, January 1st, 1970.

This function is implemented as shown below and note that it uses the more generalised function `hDlgSense` described earlier.

```
GLDEF_C ULONG hDlgSenseDtedit(UBYTE index)
    {
    SE_DTEDIT sense;

    hDlgSense(index,&sense);
    return(sense.value);
    }
```

hDlgItemDim — Dim/undim an item

```
VOID hDlgItemDim(UBYTE index, INT flag);
```

This function changes an aspect of one of the current dialog's components. No assumption is made about the type of component.

The particular component within the dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0).

The function sets the *dim* status of the dialog box component if a value of `TRUE` is passed in parameter `flag`, otherwise it sets the *undimmed* status.

The dialog box class `DLGBOX` implements dimming by removing the bullet which normally precedes the component's prompt and blanks out the component's control.

This function is implemented as shown below by sending the current dialog a `O_DL_ITEM_DIM` message.

```
p_send4(DatDialogPtr,O_DL_ITEM_DIM,index,flag);
```

hDlgItemLock — Lock/unlock an item

```
VOID hDlgItemLock(UBYTE index, INT flag);
```

This function changes an aspect of one of the current dialog's components. No assumption is made about the type of component.

The particular component within the dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0).

The function behaves in a similar fashion to `hDlgItemDim` in that it sets the *locked* status of the dialog box component if a value of `TRUE` is passed in parameter `flag`, and sets the *unlocked* status otherwise.

The dialog box class `DLGBOX` implements locking in a similar way to dimming by removing the bullet which normally precedes the component's prompt However, it does not blank out the component's control but does prevent it from being emphasised .

This function is implemented as shown below by sending the current dialog a `O_DL_ITEM_LOCK` message.

```
p_send4(DatDialogPtr,O_DL_ITEM_LOCK,index,flag);
```

hDlgTakeFocus — Change focus

```
VOID hDlgTakeFocus(UBYTE index);
```

This function changes an aspect of one of the current dialog's components. No assumption is made about the type of component.

The particular component within the dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0).

The function changes the focus to the dialog box component identified by the `index` parameter. In general, this causes most keyboard activity to be directed to that component (with the usual exception of key and modifier combinations set to be captured by other processes).

This function is implemented as shown below by sending the current dialog a `DL_TAKE_FOCUS` message.

```
p_send4(DatDialogPtr,O_DL_TAKE_FOCUS,index);
```

This function, like the `dl_take_focus` method itself, is not suitable for being called from the `dl_dyn_init` method. If an application wishes to set the focus on initialisation, it should do so from a replaced `dl_set_size` method. It may be called from any other method (such as `dl_key`) once the dialog has been made visible.

hDlgSetTwips Set floating point editor from twips value

```
VOID hDlgSetTwips(UBYTE index, UINT value);
```

This function sets a value into the control of one of the current dialog's components. The control is assumed to be an instance of the floating point editor (`FLEDIT`) or a subclass.

The particular component within the dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0).

The number in the parameter `value` is assumed to be in twip units (that is, 1/1440 inch or 1/567 cm and generally relevant to word processing type applications). The function converts this value into a floating point measurement in the current measurement units, either inches or centimetres. The resulting floating point number is set in the floating point editor.

hDlgSenseTwips Sense twips value from floating point editor

```
UINT hDlgSenseTwips(UBYTE index);
```

This function senses the value from the control of one of the current dialog's components. The control is assumed to be an instance of the floating point editor (`FLEDIT`) or a subclass.

The particular component within the dialog is identified by the `index` parameter (the first component is identified by an `index` value of 0).

The function senses the floating point value. This is assumed to contain a measurement in the current measurement units, either inches or centimetres. This floating point value is converted into twip units (that is, 1/1440 inch or 1/567 cm) and is returned by the function.

CHAPTER 18

APPLICATION DESIGN

It is clearly beyond the scope of this manual to discuss in any detail the principles and practice of object oriented design. There are a number of books available on this topic - two that have proved useful are:

Object Oriented Modeling and Design, by James Rumbaugh et al. (Prentice-Hall International, 1991)

Object Oriented Design with Applications, by Grady Booch (Benjamin/Cummings, 1991)

This chapter offers some specific guidance on object oriented application design for the Series 3 and Series 3a, using the Record application that is built into the Series 3a as an example. The full source code of this application is supplied and may optionally be installed into a *\sibosdk\record* directory. Once installed it may be built by making \sibosdk\record the current directory ant typing:

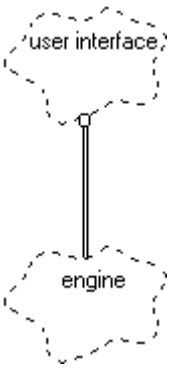
```
make record
```

The Record application illustrates a range of design techniques and solutions that may usefully be transferred to other applications. You should not, of course, take this to mean that Record is presented as a perfect example of application design. As with any design, the end result is a compromise between the ideal and the realistic. However, it is also true that, as a result of being designed, the application is much more robust and comprehensible than it would otherwise have been.

Basic design

The design of an application should, in general, separate into two layers:

- the *user interface* contains all aspects of the application that are dependent on a particular machine
- the *engine* (otherwise known as the *system model*) contains the data and the associated mechanisms that are particular to the application



For Series 3 and Series 3a applications, the user interface contains objects based on the HWIM library whereas engines do not. An engine may, however, optionally use any of the classes in the OLIB library.

As is illustrated in the accompanying class diagram, the user interface has access to the engine, but there is no unsolicited communication in the opposite direction.

The engine has no detailed knowledge of the classes making the calls. In practice this means that the user interface source files may include engine header files, but that the engine should not include user interface header files.

The division between user interface and engine gives rise to many advantages, some of the more significant ones being:

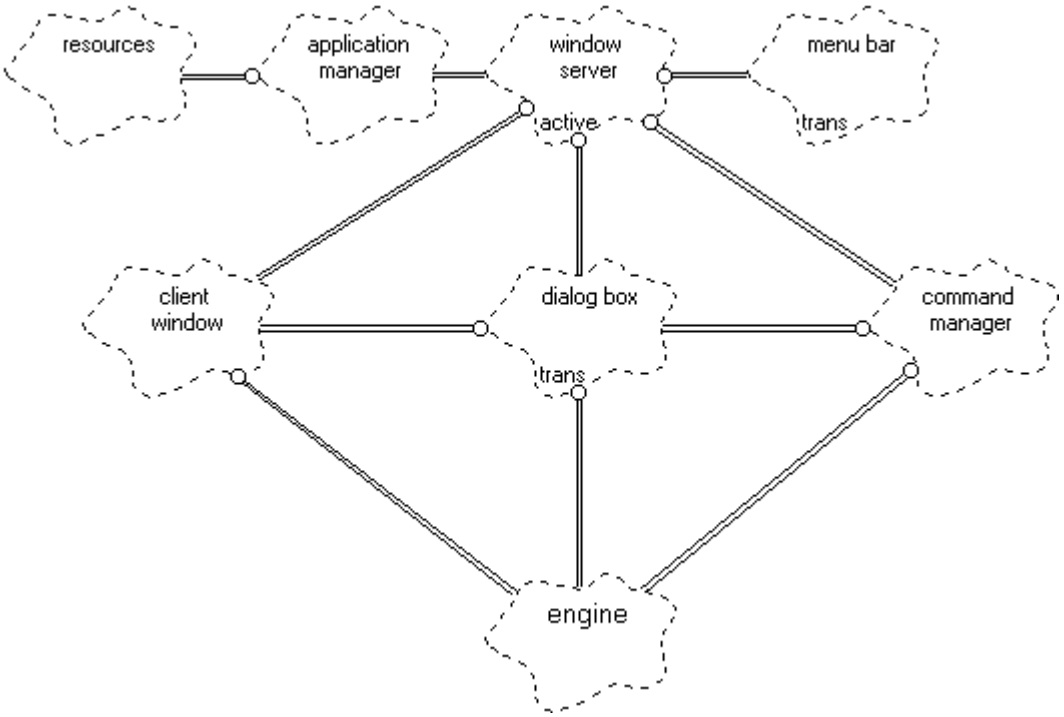
- separating the two aspects of the application reduces the complexity that has to be dealt with at any one time by both the designer and the implementer
- such separation helps the designer to reduce the number of interactions between the various component objects, resulting in a 'cleaner' and more maintainable application

- the engine does not need to be changed (or, at least, provides a good starting point) if the application is ported to another machine with a different user interface
- the engine may be tested independently - a test harness may be constructed to test all aspects of the engine, without the complications of testing via the user interface

A consequence is that it is in the programmer's interest to put as much of the application code as possible into the engine, to improve portability and to protect the investment of effort that the code represents.

A typical application

The following diagram shows the basic components of a typical application. Most of these components will be familiar from the discussion of the basic mechanisms of an object oriented application in the *Introduction* chapter.



The diagram concentrates on those aspects of the design that are common to all applications and thus omits some relationships that may be present in a particular application. It is, for example, likely that a number of classes may make use of utility functions by sending messages to the application manager and/or the window server object.

The handles of the application's instances of the application manager (`HWIMMAN`) and the window server object (`WSERV`), are globally accessible (via the magic statics `w_am` and `w_ws` respectively) and many application classes may use them - for example, to use the system services that they provide. For clarity, such relationships are not shown.

One of the more significant aspects of this diagram from a design viewpoint is the presence of the engine and its relationships with other application classes. As indicated, the engine is commonly accessed by the client window, the command manager and the application's dialogs, but not, for example, by `HWIMMAN` or `WSERV`.

The user interface

The HWIM library provides an effective design solution for many aspects of the user interface. As described in this manual, the library provides the necessary base classes and mechanisms to process the vast majority of messages that may be received from the window server process. For example, HWIM provides a ready-made solution for the processing of command options, selected either by an accelerator keypress or from pull-down menus. Thus, much user interface design reduces to considerations such as what commands options are required and what dialogs are needed to support them.

An exception to this is the client window itself, which is used to provide the top-level view of the data within the engine. Apart from the positioning and draw/redraw mechanisms, the supplied HWIM window classes provide very little in terms of general client window design. There is support, from the `EDWIN` class, for windows that display editable text and this is extended to cover printing and formatted text by the FORM library. Apart from the material in the *Edit Windows* and *Printing* chapters, these topics are beyond the scope of this manual.

The engine

An application's engine is specific to that application and, since it should be independent of the user interface, HWIM provides no significant support. In most cases, engine design will therefore be a major component of the design of the whole application.

Many engine classes will be specific to a particular application, and will directly subclass `ROOT`. It may, however, prove useful to examine the classes supplied by the OLIB library, to see if any of them could be used or subclassed. These classes are described in the *OLIB Reference* manual.

The Record application

Record is a reasonably simple, but non-trivial, application and, as such, is ideally suited to being used as an example of application design and implementation.

The application arose from a need to demonstrate the digital sound capability of the Series3a in an intuitive and easy to use manner. The application was required to provide for the recording of custom sounds to be attached to alarms and for the recording of voice "notes" for future recall, by playing back the sound. For all uses, recording has to use the minimum of key presses. Playing the sound back has also to be as easy as possible.

Constraints on human and machine resources meant that the application had to be capable of being produced quickly and be as small as possible. It may be of interest to note that the entire development, including requirements, specification, design, implementation and testing took around 90 man-hours and that the final application size is about 9 kbytes (against an initial target of 4 kbytes).

Specification

Record is a file-based application for the Series 3a. It manipulates one file at a time and keeps its current file open. The application operates on sound files that by default are stored in *\wve* directories and have *.wve* extensions. Such files are suitable for attaching to alarms. The descriptions of Series 3a sound files and the services that operate on them are given in the *General System Services* chapter of the *PLIB Reference* manual. One significant feature of the sound services is that they make direct read and write access to sound files. The Record application thus has no need to maintain an in-memory copy of its current file.

The Record application obeys *Switchfiles* and *Shutdown* messages from the system screen. It is never 'busy', even when recording or playing a file, so these messages can be received at any time.

The application has two persistent parameters, stored in environment variables:

- the sound volume at which to play back files
- the default file duration.

Top-level view

The large status window is on permanently.

The client window consists of a display of the attributes of the file being edited with buttons to perform actions on the file.

The permanently displayed attributes are:

- the name component only of the open file
- the duration of the sound in seconds

- the length of the file in bytes (so that the user is aware of the memory consumed by this file)

If appropriate, the following attributes are also shown:

- the number of repeats
- the duration of the trailing silence in seconds
- the total playing time

The buttons are:

- Record new (Tab) to record to a new file. A dialog is presented for the file name, the disk and the maximum duration. A generated file name, of the form *recordnn* where *nn* is *01*, *02*, *03* etc., is presented by default. The initially suggested disk is the default drive or, if that drive contains other than a RAM SSD, Internal.
- Record over (Space) to re-record to the file, overwriting its existing content. A dialog is presented allowing the maximum duration to be set (the suggested duration is that of the current file, rounded up to the next 2K bytes). The dialog contains a warning that the current data will be replaced.
- Play (Enter) to play back the file, including any repeats and trailing silences.

Playing

When playing the file, the three buttons are replaced by a single Stop (Esc) button. A bar graph is also presented, showing the elapsed playing time against the predicted total playing time.

Recording

After accepting the dialog presented by Record new or Record over, the three buttons are replaced by a single Start recording (Space) button. Pressing Space starts the recording and the button is replaced by a single Stop (Esc) button. A bar graph is also presented, showing the elapsed record time against the maximum duration. Pressing Esc terminates (but does not abort) the recording, resulting in a shorter sound file than would otherwise be produced. If Esc is not pressed, the recording terminates when the specified maximum duration has been reached.

Running for the first time

The first time the application is run, when no sound files exist under the icon, the application immediately presents (over a blank client window) the dialog that is presented when you subsequently press the Record new button. In this case the suggested file name is *record* (not *record01*) and pressing Esc in this dialog causes the application to exit.

The same behaviour results, but with the specified file name, if you start the application by use of the System Screen's New (Psion+N) option.

Menu

The menu commands are:

New file	Essentially has the same effect as pressing the Record new button.
Open file	Presents a dialog with standard controls to select a new current file.
Set repeat	Presents a dialog to set the repeat count (1 to 999) and the trailing silence (0 to 10.00) in the file header. This is intended to be of use for constructing custom alarms.
Adjust for alarm	Automatically sets the repeat and the trailing silence for a short recording (less than seven seconds or so) so that it is compatible with being clipped at 15 seconds by the alarm server.
Set preferences	Presents a dialog to set the volume of playback, the default duration and the default disk for new files (the last two being subsequently used by the Record to new file dialog).

Exit There is no need to save any changes - all changes are made to the file directly.

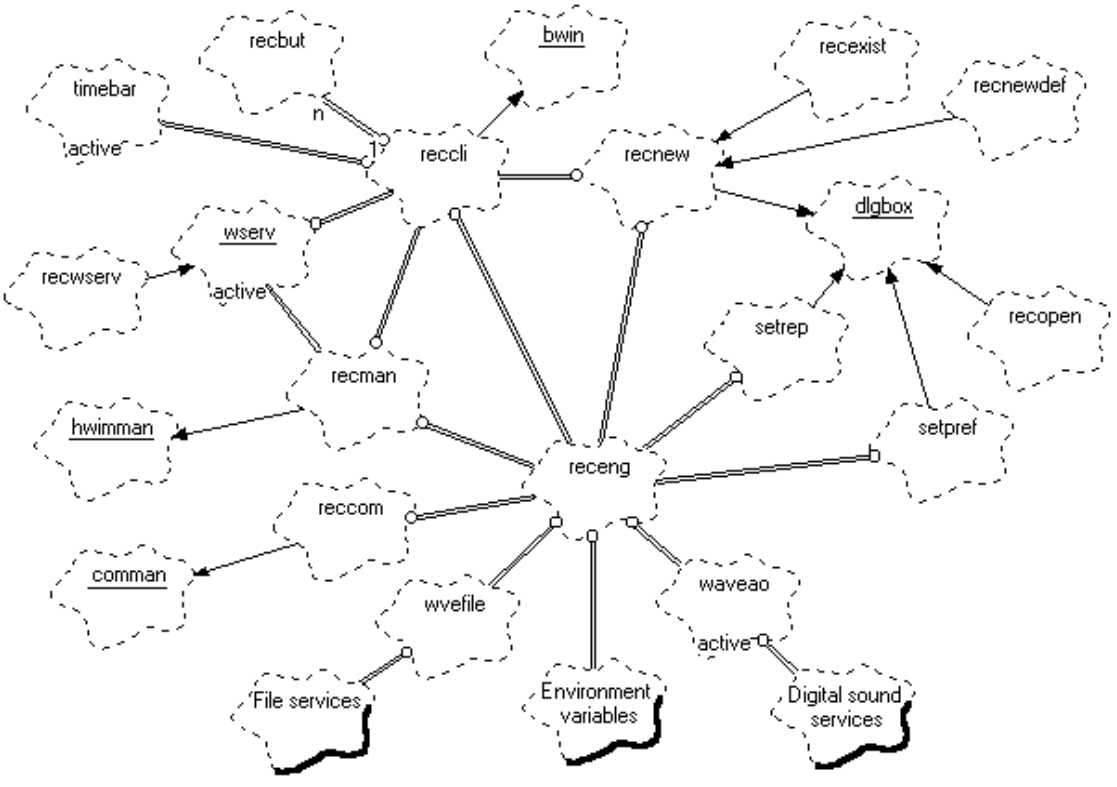
Design

The overall design of the Record application is shown in the following class diagram. Although the diagram is fairly complex, it is interesting to see that the essential design can be captured in a single diagram. The Record application represents about the highest level of complexity for which this is possible.

Despite its complexity, the diagram is still a simplification and shows less than the whole truth. It does not mention, for example, access to resource files and it omits a number of relationships of lesser significance and whose presence would hinder rather than help with an understanding of the design. The dialogs, for example, are generally run from within methods of the command manager, but including all such relationships would render the diagram totally unreadable.

The diagram does, however, show all the really significant aspects of the design. As an example, it accurately illustrates the relationships between user interface classes and the engine, although the relationship between the `RECNEW` dialog class and the engine is conceptual rather than actual (this is explained later in more detail). It is an important part of the design that the file services, digital sound services and the environment variables are accessed via the engine and are not referenced directly by any user interface component.

The relationships that are shown are drawn to represent the truth as closely as possible. The using relationship between the window server object and the client window is drawn to show that it is methods at the WSERV level that send messages to the client window. This reflects the fact the only relationships involved are those provided by system code. In contrast, as would be expected, the engine is used only by application-specific subclasses.

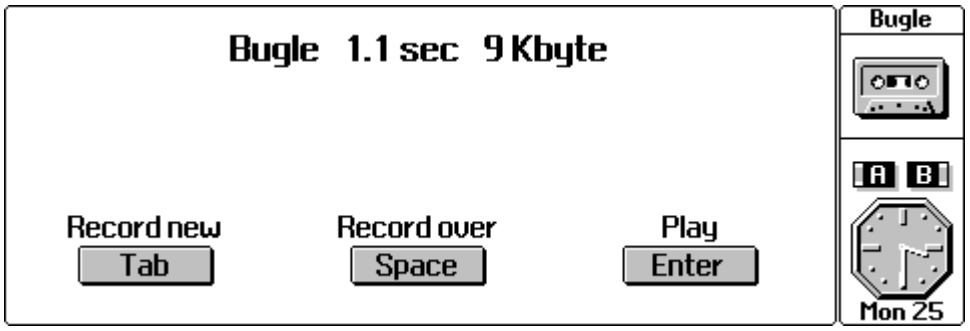


Booch class diagram for record.app

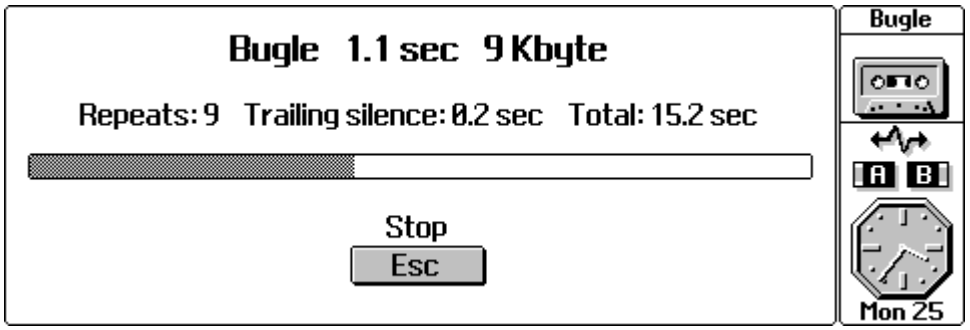
The following sections describe a number of aspects of the application's design

The client window

The client window forms the application's top-level view of the current file and initially presents a view similar to that shown below.



While recording or playing a file the buttons are replaced by a single button and a bar graph is presented to indicate the elapsed time, as illustrated in the next diagram. This diagram also shows the additional line of information that is shown for a file that is set to repeat and/or has trailing silence.

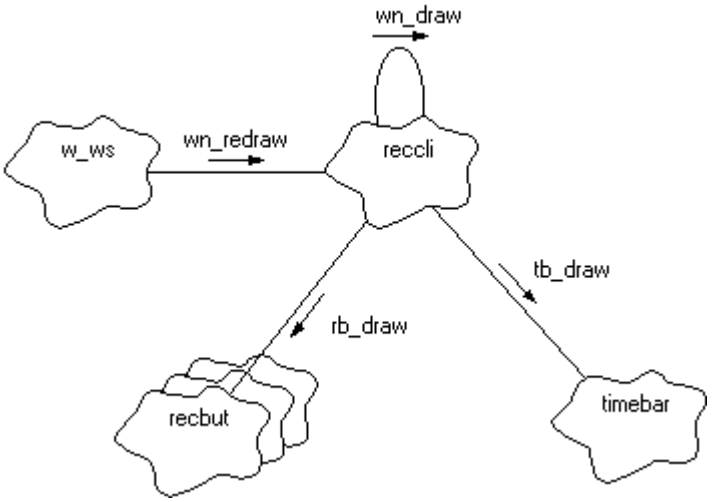


The following diagram shows the application when it is paused before making a recording, with another single button being visible.

The client window thus has, at various times, to display one or two lines of text, one or more of a set of five buttons, in various positions, and possibly a bar graph.

The client window maintains two text strings (the second of which may be a null string) for centred display at fixed vertical offsets whenever a `wn_draw` message is received.

In contrast, the bar graph and the five possible buttons are component objects of the client window and each of them contains the knowledge of its size and its position within the (fixed size) client window. Furthermore, each of these components contains a record of whether it is visible or not. Thus the client window does not need to maintain a record of which set of components is visible at any one time. An operation, such as pressing the Play button only needs to set the visibility of the appropriate components. On receipt of a `wn_draw` message, the client window always sends all six components an appropriate drawing message. Each component either acts upon or ignores this message, according to its own internal state. The following object diagram illustrates the drawing mechanism, in response to a `WN_REDRAW` message from the window server object. The same sequence is used for drawing that is initiated by the client window itself.



The client window thus has no direct knowledge of which buttons are visible. It must, however, respond to a set of keypresses that correspond to the buttons that are on display. Record accomplishes this by taking advantage of the keyboard filtering mechanism (see the *Keyboard filters* section of the *Using the System Components* chapter) to divert the processing of keypresses.

In its normal, three-button, state, keys are processed by the client window's `wn_key` method, in the normal way. When it switches to either of the other two states, not only is a different set of buttons made visible, but a filter is set, redirecting key processing to one of two alternate key processing methods - either `cl_sound_filter` or `cl_pause_filter`. The change of state actions are performed by the client window's `cl_update`, `cl_begin_sound` and `cl_pause` methods. Note that these two methods return `WN_KEY_CHANGED`, thus ensuring that interaction with the menu bar is disabled in the filtered states. Record uses negative values for `wserv.filmethod` so that Help is still available while keys are being filtered.

It is worth noting that the behaviour in each of the two filtered states is very similar to what could alternatively have been achieved by presenting a dialog. The decision to simulate dialog behaviour rather than to use dialogs for these states was largely based on cosmetic considerations.

The bar graph

While recording or playing a file, Record displays an animated bar graph to show the progress against the predicted total time for the operation. Since the sound services provide no information regarding their progress, the bar graph animation has to be performed independently of the sound recording or playback.

The `TIMEBAR` class therefore takes only a single item of external data - the total predicted time for the operation. It uses the free-running counter (`FRC:`) device in its repeating mode to ensure accurate synchronisation of the animation with the sound services.

Since the `FRC:` device is a scarce resource, it is essential that Record releases it as soon as possible. The device is therefore opened every time it is used, and closed on completion of every record or playback operation. It is particularly important to ensure that an error condition does not leave the `FRC:` device open, so its closing is made an essential part of the application's error-handling (see *The application manager*, later in this chapter).

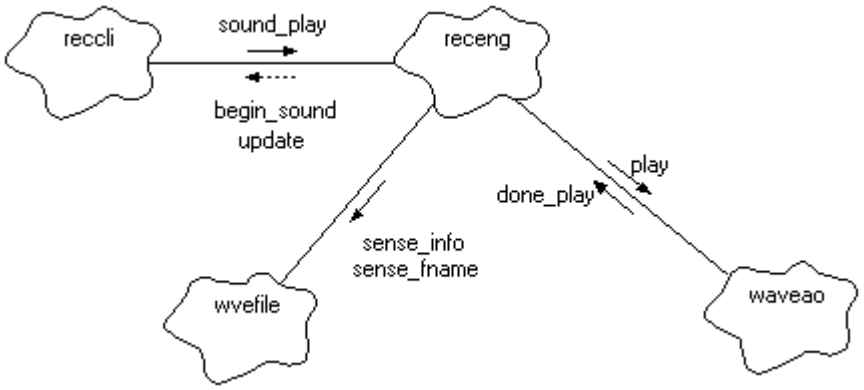
The engine

Record's engine is a static instance of the `RECENG` class, created on initialisation of the application and remaining in existence for the application's lifetime. This need not be the case in all applications - in many cases it may be appropriate to represent, say, a change of the application's file by destruction and recreation of the engine.

`RECENG` centralises access to and manipulation of the data associated with the application's current file. Its handle is globally available (via the global static variable `receng`) so that it may be referenced by any object in the user interface. The classes that actually use the engine are as indicated in the earlier application class diagram.

The engine owns instances of the WVEFILE and WAVEAO classes that respectively represent the current file and the record/playback process. All access to these classes is via RECENG methods. This reduces the engine's 'surface area', by avoiding the need for any other object to be aware of the existence of these two instances. Note that some engine methods add little value (see, for example, the eng_sense_file method that senses the file data held by WVEFILE) and exist simply in order to delegate the action to a component. Although this results in a small increase in the size of the application, this is outweighed by the design advantages that it brings.

The essentials of the sound-playing mechanism is illustrated in the following object diagram. Playing is initialised by pressing the Play button in the client window, which causes an ENG_SOUND_PLAY message to be sent to the engine.



The name and total sound duration of the current file are sensed by means of WVE_SENSE_FNAME and WVE_SENSE_INFO messages to WVEFILE and the name is passed to WAVEAO in a WV_PLAY message to initiate the sound. The engine sends a CL_BEGIN_SOUND message to the client window (see later) to cause it to change the button display and to initialise and make visible the bar graph.

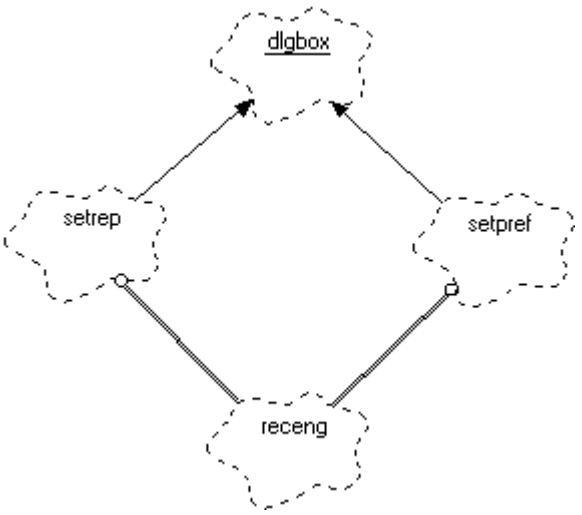
WAVEAO is an active object which breaks up the playing of the file into a series of sections, allowing the application to continue to respond to keypresses (to abort the playing of the sound) and to update the growing bar graph (another active object). When WAVEAO completes the playing of the file it sends an ENG_DONE_PLAY message to the engine which, in turn, sends a CL_UPDATE message to the client window, causing it to revert to its three-button display.

An engine should have no knowledge of user interface objects and certainly should not send them *unsolicited* messages. The design of the Record engine, however, requires the engine to send messages to the client window in response, for example, to the receipt of an ENG_SOUND_PLAY message. This apparent conflict is resolved by passing the client window's handle, and the message numbers of the required messages, to the engine as parameters. The parameters are normally sent to the engine (as in this case) with its initialisation message and are stored in the engine's property. The message can then be sent at a later time, with no knowledge of either the target object or the meaning of the message that is being sent. The engine thus preserves its ignorance of, and independence from, user interface classes. The special nature of this type of messages is indicated by the use of a dotted arrow in the above object diagram.

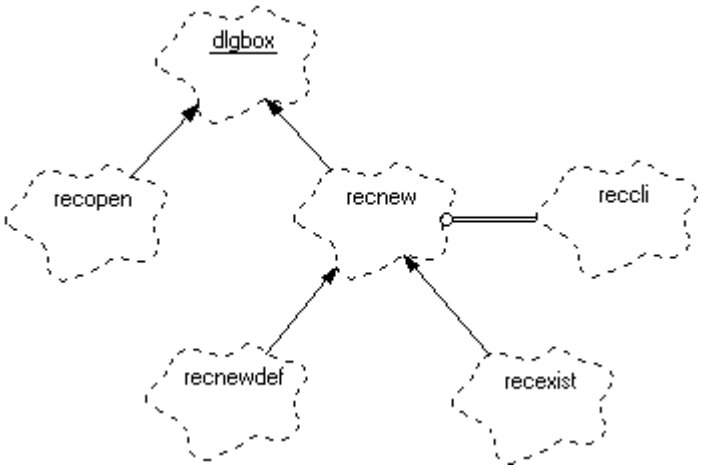
Recording to a new file or re-recording an existing file broadly follow the same pattern. The major apparent difference in the mechanism arises from the fact that these two operations are also available as command menu options. The initiation of these operations from the client window therefore shares code with the corresponding command manager methods, but the principles remain the same. Before recording to a file starts, the engine deletes any existing file of the same name.

Dialogs

As with the majority of applications, there is very little design associated with Record's dialogs since they are largely constrained by the system-supplied mechanisms.



The STREP and SETPREF classes, respectively used by the Set repeats and Set preferences menu options are typical subclasses of DLGBOX, replacing the dl_dyn_init and dl_key methods. As indicated in the above class diagram, both classes send messages to the engine to sense and set the relevant data.



The file-related dialog classes, RECOPEN, RECNEW, RECNEWDEF and RECEXIST, are related as shown in the above class diagram. The RECOPEN dialog class reports a selected file name by means of a result buffer, pointed to by an item of dialog box property, dlgbox.rbuf. It thus has no direct using relationship with any other class in the application.

The other three dialogs are associated with recording to a file. There is a conceptual relationship between these dialogs and the engine, since the current file name is found by means of a message sent (by the command manager) to the engine. This initial file name is passed to the dialogs by use of a result buffer, so the link to the engine is not represented in the above class diagram (although it is shown in the more conceptually orientated overall class diagram shown at the start of the *Design* section). All these dialogs initiate recording to the file by sending a CL_PAUSE message to the client window.

The application manager

On error, Record is designed to return to its base state, as on first entry to the application.

Record's application manager subclasses HWIMMAN to replace the am_clean_up method. This method is called by system code as part of the standard recovery from an error condition (that has caused a call to p_leave). Its purpose is to assist roll-back to a safe state by freeing any resources that have been allocated since the application was last in a secure state. For further details see *The CLEANUP Class* and the description of the am_clean_up method in *The APPMAN Application Manager Class*, both of which chapters appear in the *OLIB Reference* manual.

The replacement method adds value by also performing all central generalised error recovery needed for any error condition. Its actions include, for example, clearing the value of wserv->filter and sending

a `TB_STOP` message to the client window's bar graph (to ensure the release of the `FRC:` device). All these actions are guaranteed harmless if they are performed in a case when they are not needed.

This use of the `am_clean_up` method may not be so convenient in a more complex application where a variety of more specific error recovery procedures may be needed.

APPENDIX A

CATEGORY FILES

A category file contains the class definitions for all the subclasses provided by an image or dynamic library file. It is expected to have a *.cat* file name extension.

The category file defines the group of classes that form an object oriented category. Object oriented classes and categories are explained in the *Introduction* chapter of this manual and in the *Object Oriented Programming* chapter of the *PLIB Reference* manual.

A category file is the input to the category translator tool, *ctran.exe*, which generates a number of files, as described later in in this chapter.

Category file content

The content of a category file is best explained in conjunction with the following example:

```
A demonstration cat file
IMAGE demo

! External reference to OLIB library
EXTERNAL olib

INCLUDE p_std.h
INCLUDE p_object.h
INCLUDE varray.g           required for knowledge of VAFLAT

CLASS dummy root           the class name and its superclass
Dummy class definition,
as an illustration only
    {                      Methods follow...
    REPLACE destroy        free buffer and supersend
    ADD dm_init            create VAFLAT component and allocate buffer
    DEFER dm_sub           defined by a subclass...
    CONSTANTS              auxiliary symbolic constants
        {
        ! for the buffer
        DUMMY_BUF_SIZE 128 allocated buffer size
        ! for the VAFLAT component
        DUMMY_GRAN     16
        }
    TYPES                  contains auxiliary structs
        {
        typedef struct     /* comments here are exceptional */
            {
            TEXT *buf;     pointer to allocated buffer
            UWORD len;
            } DUMMY_BUF;
        }
    PROPERTY 1
        {
        PR_VAFLAT *array;  the component VAFLAT instance
        DUMMY_BUF buffer;
        }
    }
```

```
CLASS sub dummy
Subclass of dummy
    {
    REPLACE dm_sub       ..so that's what it does
    }
```

This example is copiously commented, to illustrate where comments are allowed. The general rules are:

- any number of lines of comment text may be placed at the start of the file, or in the lines immediately following a `CLASS` declaration
- any line starting with an exclamation mark, optionally preceded by whitespace, is ignored
- trailing comment text may be placed on any line (except for a `typedef struct` line in the `TYPES` section) provided it is separated by whitespace from significant content. The whole of a `typedef struct` line is output to the *.g* file. Any comment in this line must, therefore, be suitable for inclusion in a C source file.

Not counting comment lines, the structure of a category file is:

- an `IMAGE` or `LIBRARY` statement
- zero or more `EXTERNAL` statements
- one or more `INCLUDE` statements
- zero or more `CLASS` statements
- zero or more `REQUIRE` statements

The first non-comment line of the file declares the category type and name. The keyword must be one of:

`IMAGE`	the executable will be an image (*.img* or *.app*) file
`LIBRARY`	the executable will be a dynamic library (*.dyl*) file

The category name, in this case, `"demo"`, must be the same as the file name. This category file must, therefore have the name *demo.cat*.

An `EXTERNAL` statement declares an external reference to a dynamic library (DYL). The file may contain any number of such external references. There must be an external reference to a DYL before a category file class definition may refer to, or subclass - directly or indirectly - a class from that DYL.

The above example subclasses `ROOT`, which is in the OLIB library and so must declare an `EXTERNAL` reference to OLIB. The effect is to include an external reference (.ext) file - in this case *olib.ext* - from the designated include directory. External reference files are generated by the *ctran.exe* category translator, and are described in the *Category Translation* chapter.

Since all subclasses are ultimately derived from the `ROOT` class,[1] all category files (except that for OLIB itself) must contain an `EXTERNAL` reference to OLIB.

An `INCLUDE` statement includes a C language header file from the designated include directory. It is used in a similar way to `#include` in a C source file. All category files must `INCLUDE`, either directly or indirectly, *p_std.h* and *p_object.h*. The above example also `INCLUDE`s the header file *varray.g* (generated by the category translation of the OLIB dynamic library category file and copied to the *\sibosdk\include* directory during installation). This file contains C `#defines` and `typedefs` relating to the OLIB variable array classes, including the definition of the `PR_VAFLAT` struct.

Class definition

The `CLASS` keyword introduces a class definition. It is followed by the name of the class and then the name of the parent superclass. A class name may be up to 15 characters long. The above example defines the class `DUMMY` which is a direct subclass of the `ROOT` class. The layout of a class definition is significant; apart from leading whitespace, which is ignored, it must follow the pattern illustrated above - and in the class definitions given elsewhere.

[1] Exceptionally, a category may define its own root class and hence require no external reference to OLIB.

The class definition of each subclass lists its additional methods and any additional property. It may also, as in the above example, include the definitions of auxiliary structures and constants used by that class. There are many examples of class definitions in the manuals describing object oriented libraries (the *OLIB Reference* manual, for example).

The class definition may include any number of method declarations,[2] introduced by the `ADD`, `REPLACE` or `DEFER` keywords. Each of these is followed by a method name, which may be up to 21 characters long. The method declarations may be followed by one of each of the `CONSTANTS`, `TYPES` and `PROPERTY` keywords.

The method declaration keywords have the following meanings:

`ADD` — declare a method in addition to the methods provided by the superclass. The name must be unique in relation to all other methods in this category, or any externally referenced categories. Although not compulsory, the name conventionally starts with a short prefix related to the name of the class in which it is introduced.

`REPLACE` — declare a method whose functionality is to replace that of a method supplied by a superclass. The name must be that of an existing method in the superclass inheritance tree.

`DEFER` — declare an additional method as for `ADD`, except that the functionality of the method is not defined by the current class and is expected to be provided by a subclass (using `REPLACE`). A class containing `DEFER`red methods is known as an abstract class and, in general, no instances of such a class will ever be created.[3]

It is recommended that each method name be followed by a concise descriptive comment.

The `CONSTANTS` keyword introduces a list of symbolic constant definitions, each consisting of the symbol name (conventionally in upper case) followed by the numeric value. The value may be an expression involving symbolic constants defined earlier, either in the category file itself, or in any included file. The expression must not contain any whitespace.

The `TYPES` keyword introduces a list of C language `typedef` struct definitions, whose layout should follow that given in the example category file. (Many further examples may be found in the class definitions shown for each class in, say, the *OLIB Reference* manual.)

The `PROPERTY` keyword introduces a list of data element declarations to be included in the struct that defines the class property. The form of this struct is described in the *Category Translation* chapter.

This keyword may optionally be followed by a literal number (expressions may not be used) that specifies how many component items listed in the property are to be sent an automatic `DESTROY` message when an instance of the class is destroyed. This assumes that, for a value `ncomp`, the first `ncomp` items in the additional property for the class are either `NULL` or handles (pointers to instances) of component objects. The automatic destruction mechanism is described in the *Object Oriented Programming* chapter of the *PLIB Reference* manual. It is implemented in the `destroy` method of the `ROOT` class, described in the *OLIB Reference* manual. (See the *Building a Dynamic Library* chapter for a description of how a category may supply its own `ROOT` class.)

In the above example, `DUMMY`'s component `VAFLAT` instance will be automatically destroyed when `DUMMY` receives a `DESTROY` message.

Sub-category files

The contents of a category file may be be divided between a number of sub-category files, each of which must have a *.cl* file name extension.

[2] Subject to a maximum of 255 methods, including those inherited from the superclass tree.

[3] There is no formal requirement for all `DEFER`red methods to be `REPLACE`d and it is acceptable to create an instance of such a class provided that it is known that no `DEFER`red method will ever be called. See, for example, the HWIM `DLGBOX` class.

A sub-category file normally groups together a number of related classes.[4] The OLIB category, for example, is constructed from a number of sub-category files, including:

varray.cl	containing the class definitions of the segmented buffer and variable array classes described in the *SGBUF Segmented Buffer Class* and *Variable Arrays* chapters of the *OLIB Reference* manual
edit.cl	containing the class definitions of the classes described in the *Editable Documents* chapter of the *OLIB Reference* manual
time.cl	containing the single `TIME` class definition, described in the *TIME Class* chapter of the *OLIB Reference* manual

A sub-category file is included in the category file by means of the `REQUIRE` keyword. For example, *olib.cat* contains the line:

```
REQUIRE varray
```

to include the sub-category file *varray.cl*.

Not counting comment lines, the structure of a sub-category file is:

- a `NAME` statement
- one or more `INCLUDE` statements
- one or more `CLASS` statements

A sub-category file must start with a `NAME` statement, specifying the sub-category name. This name must be the same as the file name. Thus the *timer.cl* sub-category file must start with the declaration:

```
NAME timer
```

A sub-category file must not contain `EXTERNAL` statements and normally will not contain `REQUIRE` keywords.

During translation of the category file (described in the *Category Translation* chapter) a separate generated header (*.g*) file is created for each sub-category file.

Using sub-category files

A category file may contain in-line class definitions as well as those contained in `REQUIRE`d sub-category files. For example, the OLIB category file (*olib.cat*) is:

[4]The relationship may be by function, by inheritance, or by any other means appropriate to a particular application.

```
LIBRARY olib

INCLUDE p_std.h
INCLUDE p_object.h

CLASS   root
The ultimate superclass - all other classes have root as their
ancestor.
    {
    ADD destroy
    PROPERTY
        {
        P_OBJECT pc;    class link
        }
    }

REQUIRE varray
REQUIRE appman
REQUIRE ipc
REQUIRE factive
REQUIRE time
REQUIRE timer
REQUIRE tlvfile
REQUIRE edit
```

The OLIB category file is unusual in that it contains no references to external categories. It therefore contains no `EXTERNAL` statements and no `INCLUDE`s of any *.g* files.

The category translation of *olib.cat* generates *olib.g* and a further *.g* file (such as *varray.g*) for each of the sub-category files.

The information in a .g file may include:

- external category numbers
- includes of .h and .g files
- class numbers
- method numbers
- property structures
- auxiliary class constants
- auxiliary class structures

This information is mainly[5] required by:

- C source files that provide the method functions for the classes in the category
- C source files that provide the method functions for classes in another category, if they refer to these classes (either by using or by subclassing)

The definitions of category numbers only appear in the *.g* file generated from the main category file (such as *olib.g*). If this file contains no in-line class definitions, this is effectively all that the *.g* file contains.

The includes of *.h* and *.g* files result from `INCLUDE` statements in either the main category file or a sub-category file. The `INCLUDE` statements may be tailored to the known dependencies between sub-categories. For example, the OLIB *appman.cl* sub-category file starts as follows:

```
NAME appman

INCLUDE varray.g
INCLUDE p_que.h
INCLUDE p_gen.h
INCLUDE p_file.h
```

[5] Limited information may also be required by other files. For example, a resource file containing command menu items will require the corresponding command manager method numbers.

Note that *appman.cl* requires a reference to *varray.g* since the APPMAN class has a CLEANUP component, and CLEANUP is a subclass of an array class. Although the classes in *appman.cl* depend on *olib.g* and also require the inclusion of *p_std.h* and *p_object.h*, these do not need to be included explicitly. The classes in *varray.cl* also depend on these files, so *varray.g* can be relied on to perform the necessary includes. There is automatic protection in *.g* files against including the same file more than once, so explicit inclusion of these files, although not necessary, is harmless.

One of the major advantages of using sub-categories stems from the observation that a particular C source file almost never needs to include all the category information. Judicious division of a category file into sub-categories to reduce the amount of category data included in each of the source files can significantly reduce the time and, more importantly, the memory usage involved in building an application.

In general, classes should be grouped according to their relationships (which imply dependencies) either from subclassing or from usage as a component. For example, the classes in the OLIB *varray.cl* sub-category file (with one exception) are general-purpose array classes, all of which have VAROOT in their inheritance tree. In contrast, the classes in the *appman.cl* sub-category file are related either by being used as components of the application manager, or by the intimate relationship between the application manager and active objects.

It is perfectly acceptable for a sub-category file to contain a single class definition, if that class is isolated from other classes in the category. For example, the OLIB TIME class has its own sub-category file, *time.cl*. In this case it is particularly worthwhile since the TIME class defines a large number of constants and structures.

Category translation

The category translator tool, *ctran.exe*, is the principal tool used in the creation of a SIBO object oriented program. It generates a number of output files from a single input category file (which may include a number of external category references, header files and sub-category files).

Which output files are generated, and where they are put, is determined by flags passed to *ctran.exe*, whose syntax is:

```
ctran <name> [-e<dir> -x[<dir>] -c[<dir>] -g[<dir>] -a[<dir>] -i[<dir>] -
l[<dir>] -s -k -v]
```

where:

`<name>`	specifies the input category file name, assumed to have a file name extension of *.cat*
`-e<dir>`	specifies the directory from which files are included by the EXTERNAL and INCLUDE category file keywords
`-x[<dir>]`	specifies that an external reference *.ext* file should be generated in the current directory or, if given, the specified directory
`-c[<dir>]`	specifies that a *.c* C language category source file should be generated in the current directory or, if given, the specified directory
`-g[<dir>]`	specifies that one or more *.g* C language include files should be generated in the current directory or, if given, the specified directory
`-a[<dir>]`	specifies that a *.asm* assembly language category source file should be generated in the current directory or, if given, the specified directory. This flag is not normally set in the SDK development environment
`-i[<dir>]`	specifies that one or more `.ing` assembly language include files should be generated in the current directory or, if given, the specified directory. This flag is not normally set in the SDK development environment
`-l[<dir>]`	specifies that a *.lis* human readable class report file should be generated in the current directory or, if given, the specified directory

`-s`	specifies that output is being generated for the SDK. This flag is normally set in the SDK development environment. Omitting this flag causes additional information, irrelevant to the SDK development environment, to be included in the output *.c* and *.g* files
`-k`	specifies that a set of skeleton method function C source files are to be generated. A separate source file is generated for each class in the category file. The name of each file is the same as the class name, and has a *.c* extension. Each file is generated only if a file of that name does not already exist, so there is no danger of accidentally overwriting an existing source file. A warning is given if the file can not be created
`-v`	specifies verbose on-screen progress reports

The various output files are best described with reference to the demonstration category file, listed at the start of this chapter.

The *.ext* external reference file

The *.ext* file publishes the category name and information on the classes it contains. For each class in the category the file publishes:

- the class name
- the method names at their first appearance (when declared with `ADD` or `DEFER`)
- flags indicating if the subclass has additional property and if it supplies method functions

This file should be included (by means of the `EXTERNAL` keyword) in other category files that make external references to the category it describes. There is no need to generate this file if no other category makes such references.

The *demo.ext* file generated from *demo.cat* is as follows:

```
Generated by Ctran from demo.cat
IMAGE demo
CLASS dummy root
{
DECLARE dm_init
DECLARE dm_sub
HAS_METHOD
HAS_PROPERTY
}

CLASS sub dummy
{
HAS_METHOD
}
```

The *.c* C language category source file

This file contains the C language data definitions and initialisations for each class in the category. It is the source of the class descriptors which reside in the same code segment as the method functions (see also the *Object Oriented Programming* chapter of the *PLIB Reference* manual).

This file must always be generated, and must be compiled and linked into the application. Before linking, the object file must be converted, by means of the *ecobj.exe* tool, to move the class descriptor data into the code segment. This is performed automatically by the *ct.bat* batch file described in the *Building an Application* chapter.

The *demo.c* file generated from *demo.cat* is effectively as follows:

```
/* Generated by Ctran from demo.cat */

#include <demo.g>

/* External Superclass References */
#define ERC_ROOT C_ROOT

/* Class dummy */
GLREF_C VOID dummy_destroy();
GLREF_C VOID dummy_dm_init();
GLDEF_D struct
    {
    P_CLASS c;
    VOID (*v[2])();
    } c_dummy=
        {
          {1,(P_CLASS *)ERC_ROOT,sizeof(PR_DUMMY),0,0x6b,2,1},
          {
          dummy_destroy,
          dummy_dm_init
          }
        };

/* Class sub */
GLREF_C VOID sub_dm_sub();
GLDEF_D struct
    {
    P_CLASS c;
    VOID (*v[1])();
    } c_sub=
        {
          {0,(P_CLASS *)&c_dummy,sizeof(PR_SUB),2,0x6b,1,0},
          {
          sub_dm_sub
          }
        };

/* Class Lookup Table */
GLDEF_D P_CLASS *ClassTable[]=
    {
    (P_CLASS *)&c_dummy,
    (P_CLASS *)&c_sub
    };

/* External Category Name Table */
GLDEF_D struct
    {
    UWORD number;
    UBYTE names[1][14];
    } ExtCatTable =
      {1,
        {
        {'o','l','i','b','.','D','Y','L',0,0,0,0,0,0}
        }
      };
```

Note the expansion of the method function names to include the class name, as well as the declared method name. This ensures that, even if a method replaces one supplied by a superclass, the method function names are unique.

The *.g* C language include file

A *.g* include file contains the generated enumerated constants for the categories, classes and methods declared in the category file, together with the generated structures that define the property of each class. In addition it reproduces the auxiliary constant and structure definitions from the various classes.

The category numbers section lists the numbers used to refer to categories from within this category (for example, when creating an instance with `p_new`). This section will always contain at least one entry, with value zero, for the local category. There will be one additional entry for each externally referenced category.

Unless building an assembly language program, a *.g* file must always be generated, and must be `#included` in any C source file that refers to any of the items mentioned above.

Note that the name of a symbolic constant representing a class number is derived from the class name with a preceding `"C_"`, and that of a method number is `"O_"` followed by the declared method name.

The symbolic constant representing a class number is always of the form `CAT_XXXX_YYYY`, where `XXXX` is the name of the local category and `YYYY` is the name of the category to which the number refers. Thus the DEMO category refers to itself via the constant `CAT_DEMO_DEMO` and refers to the OLIB category by `CAT_DEMO_OLIB`.

If the category file includes one or more sub-category (*.cl*) files, an additional *.g* file is generated for each sub-category. The content of each *.g* file corresponds to the content of the relevant category or sub-category file. Note that the *.g* file corresponding to the main category (*.cat*) file does not contain information relating to the content of any of the sub-category files.

The *demo.g* file generated from *demo.cat* is effectively as follows:

```
/* Generated by Ctran from demo.cat */
#define DEMO_G
#ifndef P_STD_H
#include <p_std.h>
#endif
#ifndef P_OBJECT_H
#include <p_object.h>
#endif
#ifndef VARRAY_G
#include <varray.g>
#endif

/* Category Numbers */
#define CAT_DEMO_DEMO 0
#define CAT_DEMO_OLIB 1

/* Class Numbers */
#define C_DUMMY 0
#define C_SUB 1

/* Method Numbers */
#define O_DM_SUB 2
#define O_DM_INIT 1

/* Constants for dummy */
#define DUMMY_BUFFER_SIZE 128
#define DUMMY_GRAN 16

/* Types for dummy */
typedef struct
    {
    TEXT *buf;
    UWORD len;
    } DUMMY_BUF;

/* Property of dummy */
typedef struct
    {
    PR_VAFLAT *array;
    DUMMY_BUF buffer;
    } PRS_DUMMY;
typedef struct pr_dummy
    {
    PRS_ROOT root;
    PRS_DUMMY dummy;
    } PR_DUMMY;

/* Property of sub */
typedef struct pr_sub
    {
    PRS_ROOT root;
    PRS_DUMMY dummy;
    } PR_SUB;
```

The *.asm* assembly language category source file

This is an assembly language version of the *.c* category source file, described above. It contains the assembly language data definitions and initialisations for each class in the category. It is the source of the class descriptors which reside in the same code segment as the method functions (see also the *Object Oriented Programming* chapter of the *PLIB Reference* manual).

This file should only be generated, instead of the corresponding *.c* file, if the application is written in assembly language, in which case it must be assembled and linked into the application.

The *.ing* assembly language include file

A *.ing* file is an assembly language version of the *.g* file, described above. It contains the generated enumerated constants for the categories, classes and methods declared in the category file, together with the generated structures that define the property of each class. In addition it reproduces the auxiliary constant and structure definitions from the various classes.

It should only be generated if the application is written in assembly language, in which case it must be `INCLUDE`d in any assembly language source file that refers to any of the items mentioned above.

If the category file includes one or more sub-category (*.cl*) files, an additional *.ing* file is generated for each sub-category. The content of each *.ing* file corresponds to the content of the relevant category or sub-category file. Note that the *.ing* file corresponding to the main category (*.cat*) file does not contain information relating to the content of any of the sub-category files.

The *.lis* category listing file

A *.lis* file is a plain text listing, by sub-category, of all the classes in a category, together with their inheritance trees and use of component classes. It may be useful to aid a review of how classes are grouped into sub-categories, but is not otherwise required.

The *demo.lis* file generated from *demo.cat* is as follows:

```
Generated by Ctran from demo.cat

IMAGE demo

********** demo **********
dummy
        Derived from root
        References vaflat
        Subclassed by sub
sub
        Derived from dummy,root
```

The *.c* skeleton method function source file

A separate *.c* method function source file is generated for each class defined in either the category file or any included sub-category files. The file name is the same as the class name of the class from which it is generated (but truncated to the first eight characters in the case of long class names). Generating these files avoids the repetitive typing involved in creating method function files by hand. It is expected that a set of method function source files will be generated once only for each application.

A suitable batch file for creating the skeleton method function source files is *ctskel.bat*:

```
@echo off
ctran %1 -e..\include\ -s -k
```

which is used as, for example:

```
ctskel demo
```

This batch file is copied into the *\sibosdk\oopdemo* directory on installation of the optional OOP component of the SDK. It assumes that the SDK is installed on *C:* and should, of course, be modified as necessary for an SDK installed on a different drive.

Each source file contains the required include files and a minimal skeleton for each method function that must be supplied by the corresponding class.

The file generated from *demo.cat* for class `DUMMY` is as follows:

```
/*
dummy.c
Generated by Ctran from demo.cat
*/

#include <demo.g>

#pragma METHOD_CALL

METHOD VOID dummy_destroy(PR_SUB *self)
    {
    }

METHOD VOID dummy_dm_init(PR_SUB *self)
    {
    }
```

and that for class `SUB` is:

```
/*
sub.c
Generated by Ctran from demo.cat
*/

#include <demo.g>

#pragma METHOD_CALL

METHOD VOID sub_dm_sub(PR_SUB *self)
    {
    }
```

The general form of these files is described in the *Method Function Source Files* chapter.

In the generated skeleton files, all method functions are declared as being `VOID` and are supplied with only the first, mandatory, parameter (being the handle of the class instance). It is the responsibility of the programmer to make such modifications as are necessary to match the requirements of the method functions of a particular application.

APPENDIX B

METHOD FUNCTION SOURCE FILES

For each method included in the category file by means of either `ADD` or `REPLACE`, there must be a corresponding method function declared in one of the source files that is compiled and linked into the application.

A set of skeleton source files is generated automatically from the category file if a `-k` flag is passed to *ctran.exe*, as described in the *Category Translation* chapter.

There is no formal requirement as to how method functions should be divided between a number of source files. The most common scheme is to place all the method functions for a particular class in a single source file, using a separate file for each class (the generated skeleton files follow this scheme). Depending on circumstances, however, the method functions of a single class may be divided between two or more source files: alternatively, a single file may contain the method functions of more than one class. A viable alternative to the above scheme is to group all the method functions of the classes of a single sub-category file. The overriding aim should be to enhance the clarity and maintainability of the code.

The name of each method function is constructed by concatenating the class name with an underscore and the method name. Thus, the `va_test` method of the `DIRLIST` class has a method function with the name `dirlist_va_test`. Method function names may not be more than 31 characters long - which is less than the sum of the maximum lengths of the class name (15 characters) and the method name (21 characters).

Method function parameters

A method function may have from one to four 16-bit parameters. These correspond, with the omission of the message number, to the parameters passed to a message sending function (such as `p_send`). Note that the message sending mechanism removes the method number from the list of parameters before calling the appropriate method function.

The first parameter of each method function must be the object handle, that is, the handle of the class instance. This handle, which is a pointer to the heap cell containing the instance property struct, is conventionally given the name `self`. The name of the struct itself is derived by adding a leading `PR_` to the class name. This struct is defined in the *.g* file corresponding to the *.cat* or *.cl* file that contains the corresponding class definition. This *.g* file must therefore be *#include*d in the appropriate source file(s).

Calling conventions and function order

As described in the *Introduction* chapter, a method function should normally be declared with the `METHOD_CALL` calling convention (that is, preceded by a `#pragma METHOD_CALL` statement). The only exception is when a method function is the target of both a message sending function (such as `p_send` or `p_entersend`) and `p_enter`. In this case the method function must be declared with the `CDECL` calling convention.

A method function source file may include any number of local and/or global auxiliary functions. To avoid the need to repeatedly switch between different calling conventions, it is recommended that functions should appear in the file in a standard order, for example:

- local and global auxiliary functions, declared with `LOCAL_C` or `GLDEF_C`
- local and global auxiliary functions that are the target of `p_enter`, preceded by a `#pragma ENTER_CALL` statement
- method functions that are also the target of `p_enter`, preceded by a `#pragma CDECL` statement
- the remaining method functions, preceded by a `#pragma METHOD_CALL` statement

Where possible, auxiliary functions should be written in 'topological' order, that is, a function should appear before any reference to that function. Method functions, however, may appear in any order.

Sometimes, conflicting requirements mean that it is necessary to intermix functions of different calling conventions. If, for example, you need to position an auxiliary function that is the target of `p_enter` between other 'normal' auxiliary functions, you can surround the function in question with the pair of statements:

```
#pragma save, ENTER_CALL

...

#pragma restore
```

APPENDIX C

MECHANISMS

This appendix provides more detail than was presented in the Basic concepts section of the Introduction chapter. In addition to giving further insight into the mechanisms involved in Psion's Object Oriented system, it may prove useful while debugging errant applications.

Classes

A *class* defines the data (property) and behaviour (methods) of a particular type of object.

A class is implemented as:

- a set of method functions
- a class descriptor

The method functions are those functions that implement the class methods. The source code format of these functions is described in *Appendix B*.

Class descriptor

A class descriptor is a data structure that resides in the same code segment as the method functions of that class. It contains information relevant to the class and consists of a header, followed by an array of 16-bit code segment offsets to the method functions.

The C structure for the class descriptor header is defined as:

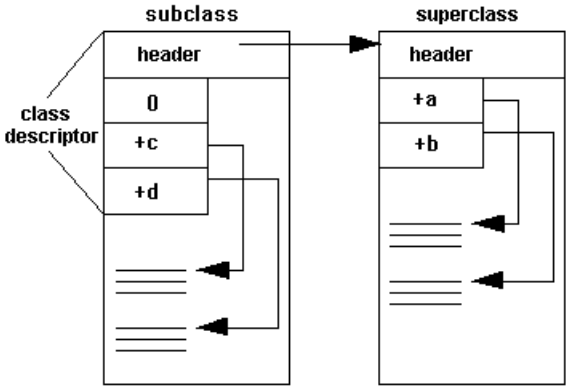
```
typedef
    {
    UWORD cat;
    struct p_class *super; /* superclass class */
    UWORD len; /* length of instance */
    UWORD base; /* base function number */
    UBYTE sig_6b; /* signature - should be 0x6b */
    UBYTE num; /* number of entries in vector table */
    UBYTE ncomp; /* number of component objects */
    } P_CLASS;
```

The contents of a loaded *and dynamically linked* class descriptor is as follows:

`cat`	the handle of the code segment that contains the superclass class descriptor
`super`	the offset of the superclass class descriptor within segment `cat`, or zero if there is no superclass (i.e. this is the class descriptor of a root class)
`len`	the length of an instance of the class, including the lengths of inherited property (used by, for example, `p_new` and `p_newlibh` to create an instance)
`base`	the base method number corresponding to the first entry in the method table that follows the class descriptor
`sig_6b`	a signature (which should be `0x6b`) to guard against a bad class reference
`num`	the number of entries in the method table

`ncomp` the number of component objects to be automatically destroyed

The following method table may contain "holes", represented by zeroes, corresponding to those method functions that are supplied by a superclass. The following diagram illustrates a typical situation:



where +a, +b, +c and +d represent code segment offsets to method functions.

In this example, the superclass provides two methods, whose method functions are at code segment offsets +a and +b. The subclass replaces the second method of its superclass, with a method function at an offset +c in its own code segment. It also adds one new method, with method function code at an offset +d.

The first method of the subclass is supplied by its superclass, as indicated by the zero in the table of offsets. Sending the corresponding message to the subclass will therefore cause the method function at offset +a in the superclass code segment to be executed.

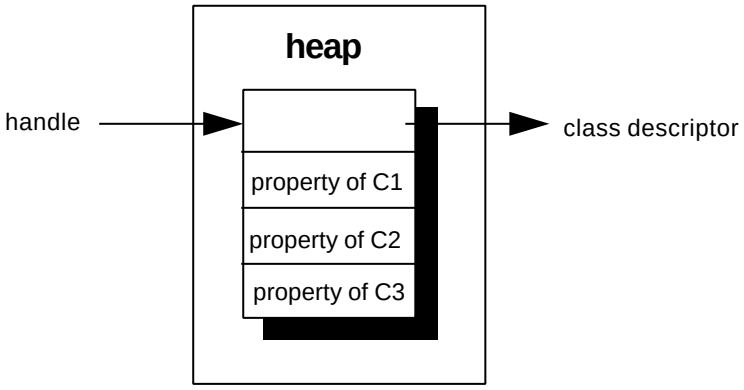
Note that the two classes may be in the same or different code segments. The resolution of the links between classes in different code segments is part of the dynamic linkage mechanism, described later.

Object creation

An *instance* of a class is implemented as a cell in the heap and is created by calling `p_new`, `f_new`, `f_newlibh`, `p_newlibh`, `f_newsend` or `f_newlibhsend`. The returned handle is a pointer to the heap cell.

The first two words of the cell contain the location of the class descriptor of the class of which the object is an instance (in exactly the same way as for the superclass reference in a linked class descriptor).

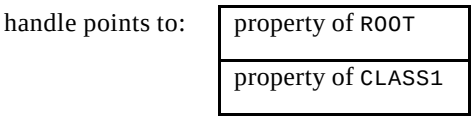
The remainder of the cell contains the *property* (if any) of that class, including any property inherited from its superclasses. The property contribution of a class always follows the property contribution inherited from its superclass, as illustrated in the following diagram.



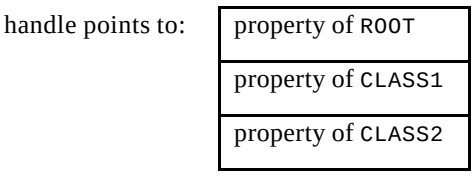
In this example, class `C1` is subclasses the `ROOT` class, `C2` subclasses `C1 and C3` subclasses `C2`. The property of `C3` is made up of the contribution from `C3` itself and the contributions inherited from `C2`, `C1` and the `ROOT` class. The various contributions are ordered within the cell as shown.

The pointer to the class descriptor that heads the cell is set up when an instance of that class is created. It is, in fact, the property of the `ROOT` class, which is the ultimate superclass of all classes.

The property contribution of a class always follows the property contribution inherited from its superclass and this ordering stays the same even if the class itself is susequently subclassed. Suppose, for example, that class `CLASS1` subclasses the class `ROOT`. The property of an instance of `CLASS1` is made up of the contribution from `CLASS1` itself and the contribution inherited from the `ROOT` class, as shown in the following diagram:



If `CLASS2` in turn subclasses `CLASS1`, the property of an instance of `CLASS2` is composed of the contributions from three classes, ordered within the cell as shown below:



All the functions (`p_new` etc) that create an object initialise the property with zeros.

Categories

A *category* is formally defined as being a group of one or more classes. The classes are packaged into a *load module* which, when loaded, occupies a single code segment. A code segment may not contain more than one category. There is, therefore, a one-to-one correspondence between a category and the executable code in a single code segment.[1] Note that this implies that an application that occupies a single code segment may not contain more than one category.

Category code segments are shared - there is only one copy of a particular category in memory, however many processes are executing it.

There are two main groups of categories:

- *Image categories* contain an entry point at offset zero and are used to implement programs. The name of a code segment that contains an image category has the extension *.$sc*. An image category code segment is created by loading an executable using `p_execc` (as described in the chapter *Processes and Inter-Process Messaging* in the *Plib Reference manual*).
- *Dynamic library categories* (DYLs) have no entry point, but contain classes that are referenced from image categories and other DYLs. The name of a code segment containing a DYL has the extension *.dyl*. A DYL code segment is created by loading a DYL load module (which may be a separate file or be embedded in an executable) using `p_loadlib` or `p_loadfilelib` (these functions are described in the *Object Oriented Programming* chapter of the *PLIB Reference* manual).

Category handles and category numbers

A category handle identifies a category code segment, which may be in RAM or the ROM, as follows:

- if the category handle is positive, it is the handle of a moveable RAM-based code segment
- if the category handle is negative, it is the paragraph address of a ROM category code segment

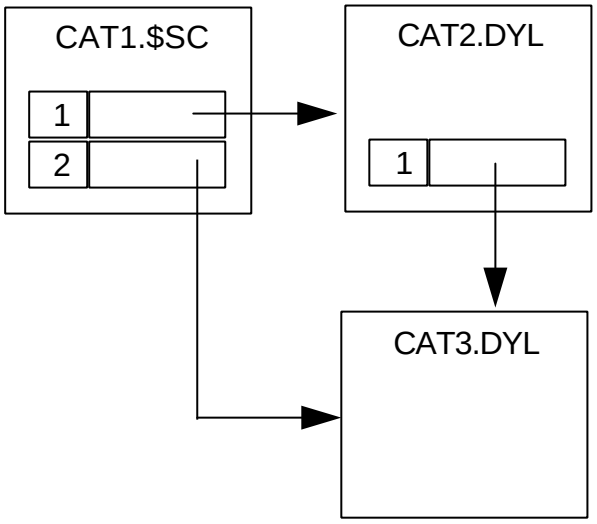
A category code segment may also be identified by a category *number*. This number is known at compile time, whereas category *handles* are only known at run time. A category number is mainly used to create an instance of an object class using `p_new`, `f_new` or `f_newsend` although it is also used by the more obscure functions `p_exactsend`, `p_reclass` and `p_cpycat`.

[1] This is true for all SIBO executables, even if they do not use Object Oriented techniques and thus do not explicitly define a category.

A local category is defined as the category containing the code that makes a category reference; an external category is a category other than the local category. Given these definitions, the local category always has the category number zero. An external category number is the index (from 1) into an array which, at run time will contain handles to the external categories. The array exists in the local category code segment and is generated from the list of `EXTERNAL` category references in a category file.

The value of an external category number thus depends on the composition and order of the external category array in the local category. Different categories will, in general, use different category numbers to refer to the same external category. Because of this fact, a category number should not be passed as a parameter to an external method (for example, to create a component of variable class). When there is a requirement to pass a category as a parameter, the category handle rather than the category number should be used. The category handle may always be obtained from the category number by calling `p_getlibh`.

For example, consider a situation where an image category *cat1.$sc* makes external references to dynamic libraries *cat2.dyl* and *cat3.dyl*, and *cat2.dyl* contains an external reference to *cat3.dyl*.



The above diagram illustrates that in this case the external category number of *cat3.dyl* from *cat1.$sc* is 2, but from *cat2.dyl* it is 1.

Dynamic linkage

A reference to an external category by category number occurs when:

- a class from an external category is subclassed by the local category
- the local category contains code that references an external category by a category number (most likely for the purposes of creating an instance of an external class using `p_new`, `f_new` or `f_newsend`).

At run time, each reference by category number (such as in a call to `p_new`) will generate a reference by category handle. Before this can be done, the calling category must be dynamically[2] linked with all the external categories to which it refers. External categories are referenced by their memory segment names and it follows that, when a category is dynamically linked, *all the referenced categories must be loaded*.

Part of the linking process is the resolution of the reference to a (possibly external) superclass in each class descriptor in the category.

A main image category is linked by calling `p_linklib(0)`. For the predominant case where an application is implemented as a single image category referencing only ROM-based DYLs (which do not need to be explicitly loaded), the image may be linked by calling `p_linklib` at any time before the execution of code involving an external category reference. The call to `p_linklib` is normally early in `main`.

[2] The term *dynamic linkage* is used because the link is made at run time. This is as opposed to normal (static) linkage between code modules, which occurs at compile time (and is used to produce a category load module, such as an executable or, indeed, a DYL).

A DYL referencing only categories that are already loaded may be linked immediately after it is loaded. This may be done by passing a suitable parameter value to the function that loads the DYL (either `p_loadlib` or `p_loadfilelib`). For example, this technique is suitable when an application category loads a DYL that references only the ROM-based DYLs and the application category.

In any case where a category (whether an image category or a DYL) contains references to categories that are not yet loaded, it must not be linked immediately after loading. First, all other referenced categories must be loaded. Only then can the category be linked by calling `p_linklib`.

Referencing by category handle

It is possible for a category to reference an external category by category *handle* - most commonly to create an instance of an external class using `p_newlibh`, `f_newlibh` or `f_newlibhsend` or to call the more obscure `p_reclassbyhandle`.

In this case, the handle is normally obtained independently of dynamic linkage by one of the following means:

- the category handle is passed as a parameter to a method
- from the segment name, by calling `p_findlib` (emulating dynamic linkage)
- by the local category loading a DYL using `p_loadlib` or `p_loadfilelib`

Message passing

In OOP terminology, sending a message to an object means calling a method function of the class or superclass of which that object is an instance.

Method functions are identified by their *method number*, which must be between zero and 255. The method number zero is normally reserved for the method that destroys the object (and its components, if any).

The most common way of sending a message is to use `p_send` (or, more efficiently, one of the `p_sendn` variants). This function must be supplied with the handle of the object instance (the address of a cell in the heap, as returned by, say, `p_new` or `p_newlibh`) and the method number as its first two parameters. Up to three additional parameters may be supplied.

The `p_send` function locates the appropriate method function as follows:

- it locates the class descriptor of which that object is an instance (using the category handle and class segment offset at the beginning of the instance)
- if the method number is in range of the method table that follows the class descriptor, and the corresponding entry has a non-zero value, it calls the corresponding method function
- otherwise it locates the superclass class descriptor and repeats the above

If the process of trying to find a corresponding method in successive superclass class descriptors (sometimes called *superclass chaining*) fails, the sending function panics with panic number 48.

The send will also panic (with panic number 55) if the category handle and class segment offset at the beginning of the instance points to a class descriptor that does not have the correct signature. This catches, amongst other things, the sending of a message to an object that has already been destroyed.

If successfully located, the method function is passed the object handle and the optional parameters (the method number passed to `p_send` is suppressed).

Within method function code (including any auxiliary functions) it is possible to 'send a message' by making a normal function call to another method function, rather than using one of the above message-sending functions. This is only possible when:

- the target method function is in the same category as the sending method
- the target method is *monomorphic*, that is, the functionality does not depend on the class of the instance to which the message is sent
- there are no calls to `p_supersend` in the target method

This technique may be used freely in application-specific classes, since such classes are totally within the control of the application writer. When writing general-purpose library DYLs, one has to be more careful about calling a local method (rather than using a message sending function such as `p_send`). Making a direct call removes any opportunity for subclassers to divert the send to a subclass method. However, in some cases it may be positively desirable to restrict subclassers in this way.

The message sending functions (`p_send` etc) represent the only mechanism for calling methods when:

- the method is *polymorphic* (where a particular send may call different method functions depending on the class of the instance to which the method is being sent)
- the method function is in an external category (for example, a ROM-based DYL)
- the method function contains a call to `p_supersend`

Calling conventions for method functions

A method function that is the target of any of the message sending functions (eg `p_send`, `p_supersend` or `p_entersend`) must use one of the following two calling conventions:

`CDECL`	where the generated code will take the parameters off the stack
`METHOD_CALL`	where the generated code will take the parameters from the registers (which is more efficient)

Note that if you call a method function directly, the prototype must be visible to the caller and must, of course, indicate the correct calling convention.

Recall, from the *Error Handling* chapter of the *PLIB Reference* manual, that the target of a `p_enter` must use one of:

`CDECL`	where the generated code will take the parameters off the stack
`ENTER_CALL`	where the generated code will take the parameters from the registers

Since the `ENTER_CALL` convention is different from the `METHOD_CALL` convention, a method function that is a target of both `p_send` (or any other message sending function, including `p_entersend`) and `p_enter` must be declared as `CDECL`.

Method parameters

In the calling convention of message sending functions, such as `p_send`, the function parameters are passed in registers. In TopSpeed C this means that no more than five parameters may be passed (including the object handle and the method number) with each parameter being limited to a 16-bit value.

Thus the parameters to a method sending function may not include the types `LONG`, `FLOAT` or `DOUBLE`, and structures may not be passed by value. The preferred technique is to pass the (16-bit) address of any of these types of data.

In exceptional cases a `LONG` may be passed as two `WORD` parameters, where the first contains the least significant word and the second contains the most significant word. A very small number of methods in the OLIB dynamic library, for example, use this technique.

INDEX

.rh extension, 15-2
ACLIST resource struct, 8-9
ACLIST_ARRAY resource struct, 8-9
AM_ADD_TASK, 16-3
AM_FINDIMG, 16-2, 16-3
AM_LOAD_RES_BUF, 15-2, 16-3, 17-4
AM_LOAD_RESOURCE, 15-2, 16-2, 17-3
AM_NEW_FILENAME, 16-2
    switching to a new file, 11-2
AM_NOTIFY, 16-4
AM_NOTIFYERR, 16-4
AM_RSCNAME, 16-2, 16-4
    replacing, 15-1
AM_YIELD, 16-3
CHLIST resource struct, 8-6
CHOICE_ITEM resource struct, 8-6
COM_ACCL_CHECK, 17-2
    example of use, 5-8, 5-9
COM_EXIT
    application termination, 11-3
    Shutdown message, 11-3
COM_FILE_CHANGE
    example of use, 11-1, 11-2, 11-3
COM_INIT
    example of use, 5-10
COM_MENU
    example, 5-6
    example of use, 5-9
COMMAN class, 5-1
DatDialogPtr, 8-1
DESTROY, 6-4, 6-10
Dialog
    altering items, 7-19
    bullet symbol, 8-3
    flags, 7-3
    prompt, 8-3
    results, 7-20
    simple, 7-18
    title, 8-2
    title - replacing, 8-5
Dialog controls
    action list, 8-7
    choice list, 8-5
    date/time editor, 8-16
    edit box, 8-9
    file name choice list, 8-20
    file name editor, 8-19
    floating point editor, 8-15
    integer numeric editor, 8-11
    latitude/logitude editor, 8-18
    long numeric editor, 8-10
    pack selector, 8-19, 8-20
    range numeric editor, 8-14
    small action list, 8-8
    text window, 8-2
    word numeric editor, 8-13
DIALOG resource
    example, 8-2
Dialog resource
    example, 7-17
Dialog resource structs, 7-3
DL_CHANGED, 7-15
DL_DIMMED_MESSAGE, 7-12
DL_DYN_INIT, 7-7
    usage, 8-1
DL_FOCUS, 7-15
DL_HANDLE_TO_INDEX, 7-8
DL_INDEX_TO_HANDLE, 7-8
DL_INQ_MINSIZE, 7-11
DL_ITEM_ADD, 7-10
DL_ITEM_APPEND, 7-11
DL_ITEM_DIM, 7-8
DL_ITEM_LOCK, 7-8
DL_ITEM_NEW, 7-15
DL_ITEM_REPLACE, 7-11
DL_KEY, 7-7, 8-1
DL_LAUNCH_SUB, 7-15
DL_SET_ITEM_FLAGS, 7-9
DL_SET_PROMPT, 7-9
    example of use, 17-13
DL_SET_SIZE, 7-12
DL_TAKE_FOCUS, 7-9
DLGBOX class, 7-2
DLGBOX methods, 7-6
DTEDIT resource struct, 8-16
EDWIN resource struct, 8-9
FLTEDIT resource struct, 8-15
FNEDIT resource struct, 8-19
FNSELWN resource struct, 8-21
h2LineConfirm, 17-11
hAppendEllipsis, 17-6
hAtob, 17-5
hAtos, 17-6
hBeep, 17-9
hBusyPrint, 17-9
hConfirm, 17-10
hDestroy, 17-2
hDlgItemDim, 17-20
hDlgItemLock, 17-21
hDlgSense, 17-17
hDlgSenseChlist, 17-18
hDlgSenseDtedit, 17-20
hDlgSenseEdwin, 17-17
hDlgSenseFledit, 17-19
hDlgSenseLledit, 17-19
hDlgSenseNcedit, 17-18
hDlgSensePtedit, 17-20
hDlgSenseRgedit, 17-19
hDlgSenseTwips, 17-22
hDlgSet, 17-12
hDlgSetChlist, 17-13

hDlgSetChlistOn, 17-13
hDlgSetDtedit, 17-16
hDlgSetEdwin, 17-12
hDlgSetFledit, 17-15
hDlgSetLledit, 17-14
hDlgSetNcedit, 17-14
hDlgSetPrompt, 17-13
hDlgSetPtedit, 17-15
hDlgSetRgedit, 17-16
hDlgSetText, 17-12
hDlgSetTitleByRid, 17-17
hDlgSetTwips, 17-21
hDlgTakeFocus, 17-21
HELP_ARRAY resource, 15-3
hEnsurePath, 17-3
hErrorDialog, 17-11
hErrs, 17-4
hGetBBWid, 17-8
hGetBWid, 17-7
hGetSWid, 17-8
hInfoPrint, 17-8
hInfoPrintErr, 17-9
hInitVis, 17-2
hLaunchDial, 17-10
hLoadChlistResBuf, 17-4
hLoadResBuf, 17-3
hLoadResource, 17-3
hSetGFont, 17-7
hSetGStyle, 17-7
hSetGTmode, 17-6
hwim.rh, 15-2
HWIMMAN, 16-1
hWservComSend, 17-2
LG_DRAW, 6-11
LG_SELF_CHECK, 6-11
LG_SENSE_WIDTH, 6-11
LG_SET_ID_POS, 6-11
LG_UPDATE, 6-11
LLEDIT resource struct, 8-18
LNCEDIT resource struct, 8-11
LODGER class
    and dialog control class, 8-1
Menu options
    validity checking, 5-8
MENU resource struct
    and choice list, 8-6
NCEDIT resource struct, 8-12
p_false, 17-2
p_true, 17-2
PUSH_BUT resource struct, 8-8
RGEDIT resource struct, 8-14
s_.hlp, 15-2
s_.rss, 15-2
SE_DTEDIT struct, 8-17
SE_EDWIN struct, 8-10
SE_FLEDIT struct, 8-16
SE_LLEDIT struct, 8-18
SE_LNCEDIT struct, 8-11
SE_NCEDIT struct, 8-12
SE_RGEDIT struct, 8-14
SE_TEXTWIN struct, 8-4
SE_WNCEDIT struct, 8-13
Subdialog, 7-22, 8-3
sx_.ra, 15-2
top-level windows, 6-4
TOPIC_ARRAY resource, 15-3
TXTMESS resource struct, 8-4
w_am, 5-11, 16-1
w_ws, 5-1
WN_CALC_POSITION, 6-6
    and WN_POSITION, 6-6
WN_CONNECT, 6-4
    and WN_POSITION, 6-6
WN_DODRAW, 6-5
WN_DRAW, 6-7, 6-8
WN_EMPHASISE, 6-5, 6-8
WN_INIT, 6-7, 6-10
WN_KEY, 6-6, 7-13
WN_POSITION, 6-6
WN_REDRAW, 6-4
WN_SENSE, 6-7, 7-8
WN_SENSE_HELP, 6-5, 7-13
WN_SET, 6-7, 7-7
    current dialog, 17-12
WN_VISIBLE, 6-5, 6-10
WNCEDIT resource struct, 8-13
WS_ALERT, 16-12
WS_BACKGROUND, 16-13
WS_CHANGE_CLIWIN, 16-7
WS_DIAL_ENV, 16-11
WS_DO_DIAL, 16-7
    example of use, 7-16
WS_DO_HELP, 16-8
WS_DO_SUBMENU
    example of use, 5-11
WS_DYN_INIT, 16-13
WS_EDIT_PDEV_SETUP, 16-12
WS_EDIT_PRINT_CONTEXT, 16-12
WS_ENS_PRINT_CONTEXT, 16-12
WS_ERROR_DIALOG, 16-10
WS_EVAL_ENV, 16-10
WS_EVALUATE, 16-10
WS_FOREGROUND, 16-13
WS_FORMAT_DIALOG, 16-11
WS_FREE_DIAL, 16-8
WS_LOAD_CHLIST_RES, 16-8, 17-4
WS_LOCK, 16-9
WS_QUERY_DIALOG, 16-9
WS_RESET_MENUBAR, 16-9
WS_SENSE_PDEV_TEXT, 16-13
WS_SET_MENUBAR, 16-9
    example of use, 5-11
WS_WRAP_PARA, 16-8
WSERV active object class, 16-5
WSERV window server active object, 6-1