

SIBO 'C' Software Development Kit

PROGRAMMING IN HWIF

Version 2.10

February 3, 1995

(C) Copyright Psion PLC 1994-95

All rights reserved. This manual and the programs referred to herein are copyrighted works of Psion PLC, London, England. Reproduction in whole or in part, including utilization in machines capable of reproduction or retrieval, without express written permission of Psion PLC, is prohibited. Reverse engineering is also prohibited.

The information in this document is subject to change without notice.

Psion and the Psion logo are registered trademarks, and Psion, Psion MC, Psion HC, Psion Series 3, Psion Series 3a and Psion *Workabout* are trademarks of Psion PLC.

TopSpeed is a registered trademark of Clarion Software Corporation. Intel 8086 and 80286 are registered trademarks of Intel Corporation. IBM, IBM XT and IBM AT are registered trademarks of International Business Machines Corp. Microsoft and MS-DOS are registered trademarks of Microsoft Corporation. Apple and Macintosh are registered trademarks of Apple Computer Inc. VAX and VMS are registered trademarks of Digital Equipment Corporation. Brief is a registered trademark of Underware Inc. Psion PLC acknowledges that some other names referred to are registered trademarks.

Contents

1 Introduction to Hwif.....	1
Overview	1
Learning to program in Hwif	1
Comparisons with OPL/w and with OOP	2
Recognising Hwif function calls	2
The connection to the Window Server	3
The general shape of Hwif programs	3
Processing events	3
Types of events	3
Programs with event sources other than keys	4
Active and inactive event sources	5
Active words and status words contrasted	6
uGetKey and uGetKeyA compared	6
Deferred processing	6
Programs with more than one get-event loop	7
Diamond key	8
Menu bar interactions	10
Where _cmds must point	10
Where _mdata must point	11
How to use uPresentMenus	11
The ManageCommand routine	12
Changing menu bar contents dynamically	12
Restrictions on valid accelerators	12
Grey underlining	13
Menu positions	13
Presenting dialogs	13
Checking for run-time errors	13
Items that can be added to Hwif dialogs	13
Longer choice lists	14
Typical dialog usage	15
Dialog underlining	15
Help dialogs	16
Help dialogs (an older alternative)	18
Line editors and multi-line editors	18
Editing features supported	18
Presenting an editor	18
Applications with more than one editor	19
Edit boxes and saved file versions	20
Printing	20
Printing features supported	20
The Print Setup dialog	21
The basic mechanism of printing	21
The PrintLine callback function	22
The location of the print buffer	22
The Print Details dialog	23
Word wrapping during printing	23
Printing the contents of multi-line editors	23
Time-text utility functions	24
Default textual representation	24
Refreshing the format on returning to foreground	24
Date/Time-text utility functions	25
Hwif, the Console, and screen output	25
Three options for graphics output	26
Practical acquisition of graphics techniques	26

Hwif opens the console channel	27
The main window.....	27
Grey	27
Dual mode applications	27
Storing data to file	28
The Dbf file format	29
Dialling telephone numbers	29
Communication with the System Screen	29
Reading the command line	30
Storing the name of the file currently open.....	30
The protocol of messages from the System Screen	30
Some notes on run-time errors.....	30
Some errors to consider	31
Strategies on handling errors.....	31
Reverting to the previous file.....	31
Errors when formatting edit boxes	32
Future developments	32
 2 Worked Examples in Hwif.....	33
How to use the supplied examples	33
The embedded suggestions.....	33
Preview of the example applications.....	33
Getting started: the Query application.....	35
Hello World	35
To build the program	35
Errors during linking.....	36
Running the application from the Series 3 System Screen	36
Further explanation of qu1.c	36
Suggestions for modifying Qu1	36
Introducing a GC	37
Suggestions for modifying Qu2	37
A status window and a menu bar	37
Defining the menu bar.....	40
The contents of ManageCommand.....	40
Suggestions for modifying Qu3	41
Supplying an icon.....	41
First examples in presenting dialogs	41
Suggestions for modifying FileSize.....	43
A date editor in a dialog	43
Choice lists and the time-text functions	44
A note on the start-up heap	46
Further comments on edump.....	46
Suggestions for modifying Qu4	46
From .img to .app.....	46
The floating point emulator sys\$8087.ldd	47
Debugging a .app application	47
Some responsibilities of being a .app.....	48
Menu command look up - by accelerator or by index?.....	48
Example of floating point editor	50
Example of numeric editor	50
Examples of other dialog items	51
Suggestions for modifying Query.....	51
Getting serious: the Tables application.....	51
The state of the application.....	52
Using an environment variable.....	53
Memory consumption by environment variables	53
Complications on reading environment variables	54
Suggestions for enhancing Ta1	54
Laying out information on the screen.....	54
Laying out an action button and its associated text	56
Positioning an action button vertically	56

Animating the action button	57
Suggestions for modifying Ta2	57
Presenting an edit box	57
Generating random numbers	58
Further comments on Ta3	58
Suggestions for modifying Ta3	59
Adding in a timer	59
Drawing the bar gauge	60
Limitation on debugging Tables	61
Suggestions for enhancing Tables	61
The remaining example applications	61
Resource file access with REMIND	61
3 Advanced Use of Hwif	63
Building the Hwif library	63
Extending Hwif	63
Combining Hwif with object oriented code	63
Using a separate DYL	64
Debugging an Hwif DYL	65
Modifying the Hwif category	66
Access to a growing scroll bar from Hwif	67
The application's category file	67
C_DONEWN items in dialogs	68
The SE_DONEWN struct and the WN_SET method of DONEWN	69
The BAR_AO active object	69
Termination of the grow bar dialog	70
General comments	71
4 Hwif Reference Documentation	73
Overview of the Hwif library	73
Two levels within the h-layer calls	73
Two layers within the u-layer calls	73
Groups of functions and naming conventions	73
Return values and error notification	74
Binary counted strings	74
Dialog and menu interactions as a special mode	74
The central functions in the u-layer of Hwif	74
Initialisation common to Hwif applications (uCommonInit)	74
Enable use of grey (uEnableGrey)	76
Read a key, waiting for its delivery (uGetKey)	77
Read a keypress asynchronously (uGetKeyA)	77
Cancel outstanding asynch' keypress request (uCancelGetKeyA)	78
Test if a keypress is outstanding (uKeyPressOutstanding)	79
Convert accelerator into command index (uLocateCommand)	79
Present a set of menus (uPresentMenus)	81
Open a dialog (uOpenDialog)	82
Run a dialog (uRunDialog)	83
Dialogs and flashing cursors	83
Add an action list to a dialog (uAddButtonList)	83
Add a choice list to a dialog (uAddChoiceList)	84
Add a general item to a dialog (uAddDialogItem)	84
Text items in dialogs (H_DIALOG_TEXT)	85
Numeric editors in dialogs (H_DIALOG_NUMBER)	85
Floating point editors in dialogs (H_DIALOG_FLOAT)	85
Time editors in dialogs (H_DIALOG_TIME)	86
Date editors in dialogs (H_DIALOG_DATE)	86
Useful date constants	86
Non-scrolling text editors in dialogs (H_DIALOG_EDIT)	86
Scrolling text editors in dialogs (H_DIALOG_EDIT)	87
Secret data input items in dialogs (H_DIALOG_XINPUT)	87
Filename selectors and filename editors (H_DIALOG_FSEL)	87

Begin a dynamic choice list (uBeginDCL)	88
Add a choice to a dynamic choice list (uGrowDCL)	88
Add a dynamic choice list to a dialog (uAddDCL)	88
Underline menu items with a grey line (uAddGreyUline)	88
Underline dialog components (uSetDialogUline)	91
The auxiliary functions in the u-layer of Hwif	92
Disable/enable Escape processing (uEscape)	92
Find ID of main window (uFindMainWid)	93
Force the application into foreground (uForceToFront)	93
Fail-safe presentation of an error string (uErrorString)	93
Fail-safe presentation of an error message (uErrorValue)	93
Check that a handle is non-zero (uCheckHandle)	93
Convert a ZTS to a BCS (uZTStoBCS)	94
Present a menu-like dialog (uDialogMenu)	94
Present a dialog containing textual information (uDisplayText)	94
Time-text utility functions	95
Open a time-text channel (hTTOpen)	95
Set time-text abbreviation lengths (hTTSetAbbreviations)	95
Set time-text format (hTTSetFormat)	95
Set time-text time and date (hTTSetTime)	96
Sense string from time-text channel (hTTSenseString)	96
Close a time-text channel (hTTClose)	97
Stand alone date/time text editor functions	97
Create a date/time text editor (hDTOpen)	97
Close date/time text editor (hDTClose)	97
Set value in date/time text editor (hDTSet)	97
Retrieve value from date/time text editor (hDTSense)	97
Date/time text editor self-check (hDTSelfCheck)	97
Pass key to date/time text editor (hDTHandleKey)	98
Emphasise date/time text editor (hDTEmphasise)	98
hDTxxx structs (H_DTEDIT)	98
hDTxxx structs (H_SE_DTEDIT)	99
Edit box functions	99
Open an edit box (hEBOpen)	99
Pass a key to an edit box (hEBHandleKey)	100
Formatting in background	100
Complete edit box formatting (hEBCompleteFormat)	100
Sense the text in an edit box (hEBSenseText)	101
Set the text in an edit box (hEBSetText)	101
Emphasise an edit box (hEBEmphasise)	101
When edit boxes lose their cursor	101
Set edit box select and cursor (hEBSetSelect)	102
Sense edit box select and cursor (hEBSenseSelect)	102
Sense edit box clipboard contents (hEBSenseClipText)	102
Set edit box clipboard contents (hEBSetClipText)	102
Change width of edit box (hEBChangeWidth)	102
Set width of edit box cursor (hEBSetCWidth)	103
Insert text buffer into edit box (hEBInsert)	103
Replace selection in edit box (hEBReplace)	103
Evaluate edit box expression (hEBEvaluate)	103
Copy function for edit box (hEBCopy)	103
Paste function for edit box (hEBPaste)	104
Find text within edit box (hEBFind)	104
Clear edit box changed flag (hEBClearChanged)	104
Sense if edit box contents have changed (hEBSenseChanged)	104
Hide or show edit box symbols (hEBShowSymbols)	104
Close an edit box (hEBClose)	105
Return handle of edit box document component (hEBSenseDoc)	105
Set capacity of document component (hEDCapacity)	105
Insert text into document component (hEDInsert)	105
Notify edit box that document has changed (hEBDocChanged)	105
Set word-wrap margin (hEBSetMargin)	106
Sense word-wrap margin (hEBSenseMargin)	106
Convert position to line number and pixel offset (hEBPosToXL)	106

Printing and print support functions	107
Invoke Print Setup dialog (hPrintSetupDialog)	107
Invoke Printer Configuration Setup dialog (hPrinterSetupDialog)	107
Print data (hPrint)	107
Word wrapping during printing	108
Page breaks during printing	108
Limitations during the PrintLine callback	108
Set subsequent indent for printing (hPrintSetSI)	109
Sense page width for printing (hPrintSensePageWidth)	109
Sense printed width of buffer (hPrintSenseBufWidth)	109
Advanced possibilities when printing	109
Miscellaneous functions	109
Crack the command line (hCrackCommandLine)	109
Notify a change in the name of the file (hSetUpStatusNames)	110
Ensure that the path exists for a filename (hEnsurePath)	110
Position a dialog (hDlgPosition)	110
Emit DTMF tones (hDTMFString)	111
Check if database file is compressible (hIsDbfCompressible)	111
Access help engine (hHelpSubSystem)	111
Help resource structures	112
HELP_ARRAY	112
STRING	112
TOPIC_ARRAY	112
Pass handle of Appl.Res.Channel (hDeclareAppRcb)	112
Open channel to resource file (hInitAppRcb)	112
Request insertion of resource file pack (hRequestReplacePack)	113
Set system resource file language (hSetSystemResourceLang)	113
Set choice list to use a variable array (hSetVarrayInChlist)	114
Load applications's DYL (hLoadOwnDyl)	114
Get last key press (hLastSystemKey)	115
Call an object oriented dialog (hOODDialog)	115
The low level h-layer functions	116
Menu bar interactions - overview	116
Dialog interactions - overview	116
Low level initialisation (hIFInit)	117
Open a menu card (hCardOpen)	117
Add an item to a menu card (hCardAdd)	117
Close a menu card (hCardClose)	117
Open a menu bar (hMenuOpen)	117
Add a menu card to a menu bar (hMenuAdd)	117
Run a menu bar (hMenuRun)	118
Close a menu bar (hMenuClose)	118
Open a dialog (hDlgOpen)	118
Run a dialog (hDlgRun)	118
Close down an incomplete dialog (hDlgClose)	118
Add an item to a dialog (hDlgAdd)	118
Open a choice list (hChoiceOpen)	118
Add an item to a choice list (hChoiceAdd)	118
Close an incomplete choice list (hChoiceClose)	119
Open a button action list (hButtonOpen)	119
Add a button to an action list (hButtonAdd)	119
Close an incomplete action list (hButtonClose)	119

1. The first part of the paper discusses the importance of the study of the history of the United States. It is argued that the study of the history of the United States is essential for a full understanding of the country and its people. The author points out that the history of the United States is a complex and multifaceted one, and that it is important to study it from a variety of perspectives. The author also points out that the study of the history of the United States is important for the development of a sense of national identity and pride.

2. The second part of the paper discusses the importance of the study of the history of the United States. It is argued that the study of the history of the United States is essential for a full understanding of the country and its people. The author points out that the history of the United States is a complex and multifaceted one, and that it is important to study it from a variety of perspectives. The author also points out that the study of the history of the United States is important for the development of a sense of national identity and pride.

3. The third part of the paper discusses the importance of the study of the history of the United States. It is argued that the study of the history of the United States is essential for a full understanding of the country and its people. The author points out that the history of the United States is a complex and multifaceted one, and that it is important to study it from a variety of perspectives. The author also points out that the study of the history of the United States is important for the development of a sense of national identity and pride.

4. The fourth part of the paper discusses the importance of the study of the history of the United States. It is argued that the study of the history of the United States is essential for a full understanding of the country and its people. The author points out that the history of the United States is a complex and multifaceted one, and that it is important to study it from a variety of perspectives. The author also points out that the study of the history of the United States is important for the development of a sense of national identity and pride.

CHAPTER 1

INTRODUCTION TO HWIF

Overview

Hwif, the Handheld Wimp InterFace, is a library of user-interface routines for C programmers writing applications for the Psion Series 3, Series 3a and *Workabout*. At the time of writing, no comparable library exists for the HC or MC ranges of SIBO computers.

In this manual, references to the Series 3 should generally be taken also to refer to the Series 3a and the Workabout. Cases where there are differences in behaviour between the various models are mentioned explicitly.

Hwif makes it easy for programmers to present menus, dialog boxes, and edit boxes. Hwif also allows programmers to tap into the abundant printing functionality of the Series 3 - and much more, beside.

Hwif is an *interface* library because it provides access to code that is already present in the Series 3 ROM. Applications which include calls to Hwif will find that the size of their code only increases slightly as a result. This is in marked contrast to the case if an *alternative* user interface system is developed. At the same time, the ROM-based user interface is likely to be more comprehensive and robust than any such alternative user interface. Finally, users will instantly feel at home with the user-interface presented by Hwif applications - since it is the same as that possessed by the applications built into the Series 3.

Going beyond the issues of dialogs, menus, and edit boxes, Hwif assists programmers in writing applications that conform to the *wider responsibilities* of applications on the Series 3 - in terms of communicating information to and from the System Screen and the status window. This adds to the effect of applications which behave just like the built-in ones.

On a more practical issue, Hwif simplifies many of the *initial programming choices* facing would-be Series 3 programmers. One particular method of interfacing to the Window Server is selected over all others, and one particular method of structuring responses to external events is highly recommended. These choices out of the way, programmers can concentrate, in the meantime, upon acquiring familiarity with many of the other aspects of the Series 3 ROM software. In due course, programmers may wish to re-evaluate the choices made for them by Hwif, and may choose to take alternative decisions (for example, a different interface to the Window Server). In the meantime, Hwif helps programmers get off to a flying start.

Learning to program in Hwif

The documentation for Hwif consists of three chapters: overview, worked examples, and reference.

- The current chapter gives an overview of the scope of Hwif, and of the basic programming ideas it embodies; generally speaking, this is the chapter that should be read first
- The chapter *Hwif reference guide* presents a factual documentation of all the functions available in the Hwif library; generally speaking, this is the chapter that should be read last
- The chapter entitled *Worked examples in Hwif* contains a more discursive account of how these library functions can be used, in the context of a series of example programs.

The associated example applications are an integral part of learning to program in Hwif. As well as illustrating access to the built-in menu and dialog system, they also contain many demonstrations of how to use other, wider parts of the rich ROM software the Series 3 provides. To start with, the examples are built up in stages, to make them easier to understand. The very first example is built up in stages right

from the very beginning (an Hwif version of the *Hello World* program), with ample discussion of such topics as programming environment, and how to "build" ("make") an application. The later examples embody considerable sophistication and an extensive use of the Series 3 ROM software - enough to keep even the most avid of would-be Hwif programmers busy for quite some time.

Comparisons with OPL/w and with OOP

There are in fact three choices for programmers wishing to hook into the Series 3 built-in menu and dialog system:

- Write in OPL/w, the Wimp extension of OPL
- Write in traditional C, using routines in the Hwif library
- Write in Psion's proprietary object-oriented extension of C, interacting much more directly with the object classes in the ROM.

The advantages of OPL/w are:

- OPL programs can be written and debugged on the Series 3 itself, without any additional hardware or software tools
- OPL is a *protected* environment, automatically handling runtime errors so that they do not cause an application crash (or even a system crash)
- Some programmers find the syntax of OPL less intimidating than that of C
- OPL provides a convenient layer over the raw database file functionality provided by the OS.

On the other hand, writing in C has the following gains:

- The code executes more swiftly
- C is a richer programming environment, with abstract data structures, pointers, and typedefs
- It is generally much simpler to call routines in the OS from C than from OPL
- C allows access to a wider range of printing and editing support utilities
- Programmers with experience of C have no need to learn OPL
- Code written in C for other products on other hardware can obviously be converted more quickly into C for the Series 3 than into OPL for the Series 3
- Conversely, code written in C for the Series 3 is more likely than OPL code to have parts that are portable to other projects; in this sense, programming in C is a better long-term investment.

As far as functional access to menus and dialogs goes, there is little of substance to choose between OPL/w and Hwif. For greater control over menus and dialogs, programs have to be written in object-oriented C. This is discussed in its own manual.

Compared to the full object-oriented approach, Hwif is a significantly simpler programming system; nevertheless, it allows the creation of powerful and attractive applications. As evidence of this, a series of realistic example applications accompany this manual, each written using Hwif. These applications are all serviceable utilities that are likely to be found useful, either as they stand, or after an element of customisation. But they by no means exhaust the scope of what can be achieved using Hwif. Indeed, each example is deliberately constructed in such a way that an enterprising developer could enhance it considerably. To this end, the discussions on each example contain numerous suggestions as to how the application could be modified and extended.

Recognising Hwif function calls

Hwif library calls all have names starting either with lower case *u* or lower case *h*. For example, `uCommonInit`, `uGetKey`, `hPrintSetupDialog`, and `hDlgPosition`.

In the code fragments below, routines supplied by the application have names starting with upper case letters. For example, `SpecificInit`, `MainLoop`, and `ManageCommand`.

Routines with names starting with lower case *w* or lower case *g* are part of `Wlib`, the Window Server library. `Plib` functions have names starting with *p_* (or sometimes *f_* or *Dbf*).

The connection to the Window Server

The type of connection made by Hwif to the Window Server is actually via the *console device* (discussed further in a later section). Although this will be of little concern to most Hwif programmers, several points about the nature of this connection should be stated (particularly for the sake of those programmers already familiar with the contents of the Window Server manual):

- The connection is made `W_CONNECT_DISABLE_LEAVES` - removing the need to enclose Wlib calls in `p_enter` harnesses. When an error occurs in a Wlib call (such as lack of system memory for the request to be carried out), a simple error value is returned, rather than a call to `p_leave` being made.
- The connection is made `W_CONNECT_PRIORITY` - so that the priority of the application is automatically adjusted, as standard, whenever the application passes into foreground or background.
- The connection creates a window the full size of the Series 3 or Series 3a screen, with its own backed bitmap - so that Hwif programmers have no need to concern themselves with processing redraw requests. Note that the Series 3a screen can emulate the Series 3 screen.

The general shape of Hwif programs

All Hwif programs can have the following for their main routine:

```
GLDEF_C VOID main(VOID)
{
    uCommonInit();
    SpecificInit();
    MainLoop();
}
```

The library routine `uCommonInit` performs initialisation that is common to all Hwif programs, such as connecting to the Window Server, and creating control blocks for subsequent menu and dialog interactions. A routine such as `SpecificInit` would be supplied by the program itself, to carry out initialisation specific to the particular application, prior to getting down to the real business of the program. Finally, a routine such as `MainLoop` (also supplied by the program) is where this real business is handled.

Hwif programs can usefully be viewed as consisting, in their steady state, of a *series of responses to events*. Accordingly, the routine `MainLoop` can always have the general shape

```
LOCAL_C VOID MainLoop(VOID)
{
    FOREVER
    {
        GetEvent();
        ProcessEvent();
    }
}
```

More flesh is placed on this skeleton in the sections immediately following.

The program terminates in response to suitable user-input, probably with a call

```
p_exit(0);
```

inside a routine such as `ProcessEvent`.

Processing events

Types of events

Among the types of events that a program might process are:

- menu command hot keys, of the PSION + X variety

- other keys with special meaning for the program, such as cursor keys or alphanumeric input
- system messages such as notification of passing into foreground or background, or the Series 3 being turned off and then on again
- messages from the System Screen for the application to close down, or to change the file currently being used
- the expiry of timers or alarms
- the receipt of data via the serial channel.

These events can in fact be classified into just two types:

- those received as a result of *reading a keypress*
- *others* - of which only the last two in the earlier list count.

The point is that foreground and background messages, on the one hand, and messages from the System Screen, on the other, are both received by Hwif applications as special sorts of keypresses (ones with specially recognisable values of the keycode). (What is going on here is that the console device is converting all Window Server events to otherwise unused keycodes).

This leads to a particularly simple form for the routine MainLoop:

```
LOCAL_C VOID MainLoop(VOID)
{
    WMSG_KEY key;      /* to receive key press */

    FOREVER
    {
        uGetKey(&key);
        if (key.keycode&W_EVENT_KEY)
        {
            ...        /* process system message */
        }
        else if (key.keycode&W_SPECIAL_KEY)
        {
            ...        /* process menu command hot key */
        }
        else
        {
            ...        /* switch statement on other keys of interest */
        }
    }
}
```

The routine MainLoop can have this form for all applications in which there are no events to process, other than those received as keypresses.

An application can of course omit the test

```
if (key.keycode&W_EVENT_KEY)
```

if it has no special action to take on passing into foreground or background, or on the Series 3 being switched on, or on receipt of any messages from the System Screen.

An application can likewise omit the test

```
if (key.keycode&W_SPECIAL_KEY)
```

if it has no menu bar (for example, if it is only using Hwif in order to access its dialog functionality).

Programs with event sources other than keys

Consider a program in which there are events other than keypresses. For example, a Spy program giving information about all applications currently running on a Series 3 might update its display regularly, on a timer - to ensure that the display keeps up to date with what is happening in all the different applications.

In such a case, the program cannot know in advance, at any one time, which of the two events will occur first: the receipt of a keypress, or the expiry of the timer.

Accordingly, the *synchronous* call `uGetKey` must be replaced by the *asynchronous* call `uGetKeyA`. This latter call takes an additional parameter - the address of a *status word* that is written to when a keypress is in due course received. If on the other hand the timer expires without a keypress being received, it is the status word of the timer that is written to.

The status word is set to `E_FILE_PENDING` when the call `uGetKeyA` is made, and is set to zero when there is a keypress ready to deliver. `MainLoop` in this case acquires the following form:

```
LOCAL_D WORD timstat;
LOCAL_D WORD keystate;
LOCAL_D WORD keyactive=FALSE;
...

LOCAL_C VOID MainLoop(VOID)
{
    WMSG_KEY key;

    FOREVER
    {
        if (keyactive)
            wFlush();      /* flush any outstanding graphics calls */
        else
        {
            uGetKeyA(&keystate,&key);
            keyactive=TRUE;
        }
        p_iowait();        /* wait for something to happen */
        if (keystate==E_FILE_PENDING)
        {
            ...            /* the timer must have expired */
        }
        else
        {
            keyactive=FALSE;
            ...            /* proceed as above */
        }
    }
}
```

This is more complicated than the preceding version (in which there is only one event source) in each of two ways:

- the single line of code with the call `uGetKey` has been replaced by a series of lines that calls either `uGetKeyA` or the Window Server routine `wFlush`, and then in either case calls `p_iowait`
- the code that determines the action appropriate to the keypress that has just been received now has to stand alongside additional code that determines, by means of tests on status words, whether a keypress has indeed been received, or whether it is another sort of event that needs to be processed.

An application with more than two event sources will have a correspondingly enriched set of tests on status words, in order to find the event source which has delivered an event. On the other hand, there is no further complication over the *GetEvent* part of the routine - this remains as in the above example.

Active and inactive event sources

Note that applications using the asynchronous call `uGetKeyA` need (on pain of being panicked) to keep track of whether they already have a so-called *outstanding read* for a keypress. In the above example, this is handled by the variable `keyactive`:

- every time round the main loop, the call `uGetKeyA` should be made only if `keyactive` is `FALSE`
- `keyactive` is initialised as being `FALSE`
- every time `uGetKeyA` is called, `keyactive` is set `TRUE`
- every time a keypress is actually received, `keyactive` is set `FALSE` again.

Similar care must be taken for any other asynchronous event source. Thus the above program would probably have a variable `timactive` too (the code that primes the timer is missing from the above listing). In general, event sources do not take kindly to being asked more than once to deliver an event, without

an event being delivered in the meantime. This is regarded as evidence of faulty program logic, deserving of a panic.

Of course, it would be possible to simplify the above code example, dispensing with the explicit variables `keyactive` and `timactive`: the call `uGetKeyA` could be made at once, when a keypress is received, instead of setting the variable `keyactive` to `FALSE`. However, this is not recommended in general. In practice, as a program grows to contain more event sources, it becomes ever easier to keep track of which are active by using `xxxactive` variables, instead of by ad hoc program logic.

The need to flush the Window Server command buffer at least once each time round the main loop also counsels in favour of locating the call `uGetKeyA` as advised above. (Reading a key, whether synchronously or asynchronously, automatically causes the command buffer to be flushed. It is only when a non-keypress event has been received that an explicit call to flush the buffer is required.)

Active words and status words contrasted

Note carefully that the *active word* and the *status word* of an event source serve two different purposes:

- the active word records whether a *request* has been made to the event source to deliver an event when one is available
- the status word records whether an event has in fact been *delivered*
- the active word is written to *by the application itself*, when it requests the delivery of an event
- the status word is written to *by code other than in the application* - for example, by a timer wait handler routine (when a timer expires), or by the Window Server process (when a key is to be delivered).

`uGetKey` and `uGetKeyA` compared

As may be surmised, the call

```
uGetKey(&key);
```

is effectively equivalent, in programs with no other event sources, to

```
WORD keystat;  
  
uGetKeyA(&keystat,&key);  
p_iowait();
```

However, in programs with more than one event source, the call `uGetKey` will return *only when a keypress has been received* (this includes quasi-keypresses such as coming info foreground), whereas the call `p_iowait` will return when *any* event is ready to be serviced.

Accordingly, what `uGetKey` strictly corresponds to is

```
WORD keystat;  
  
uGetKeyA(&keystat,&key);  
p_waitstat(&keystat);
```

since `p_waitstat` returns only when the event associated with the passed status word has indeed expired - regardless of whether other events expire in the meanwhile.

Deferred processing

An independent way in which the structure of a program's `MainLoop` can be developed is via a call which simply checks if a keypress has been received, without actually delivering it. In case no keypress has been received yet, the program might continue with some intensive processing, whereas if a keypress is outstanding, that processing might be deferred until the keypress has been dealt with.

For example, suppose an icon is being re-positioned on the screen by cursor keystrokes, and that drawing the icon in its new position is time-consuming. To increase performance, a programmer might decide to draw the icon in its current position only if the user has ceased pounding on the cursor keys. Thus

```

UpdatePending=FALSE;
FOREVER
{
  uGetKey(&key);
  switch (key.keycode)
  {
    ...      /* may set UpdatePending */
  }
  if (UpdatePending && !uKeyPressOutstanding())
  {
    DrawIcon();      /* time consuming */
    UpdatePending=FALSE;
  }
}

```

Programs with more than one get-event loop

Many programs possess more than one *mode*. In different modes, various keypresses can have different meanings. Thus an icon designer program might have a special mode in which cursor keys reposition a selected portion of the screen, whereas ordinarily, cursor keys might simply reposition the current drawing point. Again, a database program may have one mode in which records are being found, and another in which records are being added or updated. As another example, an agenda program may have one mode in which a month view is presented, and another in which a day view is presented.

There are in fact two different approaches to programming in more than one mode:

- have different get-event loops for each mode
- just have one get-event loop, and keep state variables to decide the appropriate response to various incoming events.

To illustrate the first approach, consider again the case of an icon designer application which enters a special mode on receipt of a designated menu command. This could be programmed as follows:

```

LOCAL_C MainLoop(VOID)
{
  FOREVER
  {
    /* outer get-event loop */
    ...
    if (...)      /* designated menu command received */
    {
      SetUpSpecialMode();
      FOREVER
      {
        /* an inner get-event loop */
        ...
      }      /* exit this loop on certain conditions */
      TidySpecialMode();
    }
    ...
  }
}

```

In practice, the code responding to the designated menu command would probably be isolated in its own separate subroutine.

Again, an agenda application with two different modes might in theory be structured as follows:

```

LOCAL_C VOID MonthViewMainLoop(VOID)
{
  FOREVER
  {
    ...
  }      /* exits loop only when user transitions out of month view */
}

```

```
LOCAL_C VOID DayViewMainLoop(VOID)
{
    FOREVER
    {
        ...
    } /* exits loop only when user transitions out of day view */
}

LOCAL_C VOID OverallMainLoop(VOID)
{
    FOREVER /* start up in month view */
    {
        MonthViewMainLoop();
        DayViewMainLoop();
    }
}
```

To compare the two approaches to programming in more than one mode:

- Programming with more than one get-event loop is generally easier: relevant state variables (such as might be initialised in a routine like `SetupSpecialMode` referenced above) can be kept on the stack
- However, it is only possible to go so far with more than one get-event loop; as programs become more complicated, just having one get-event loop becomes an ever better design decision.

In particular, a program with more than one get-event loop may founder on account of the large overhead of maintaining *shared common processing* between the different loops. Thus testing for special events such as

- a message from the System Screen to terminate the application
- notification that the Series 3 has been switched on
- the expiry of timers or the receipt of data via the serial channel

could well be largely independent of which mode the application is in. As a result, logic would have to be duplicated at the tops of the various possible get-event loops.

Incidentally, it is sometimes appropriate to *disable* some kinds of events when transiently entering a certain mode. Thus messages from the System Screen to terminate the application or to change the currently open file can be disabled by the simple line of code

```
DatLocked=TRUE;
```

in which case, in the System Screen, the user will be informed that the application is "busy" on any attempt to close it down or to change the file being used.

Diamond key

The following discussion on the diamond key applies exclusively to the Series 3a.

In the previous section, the idea of programs possessing more than one *mode* was discussed.

The concept is used in the built-in applications such as *agenda* and *data*.

Commonly, applications switch to a particular mode when the corresponding key press event occurs. For example, the *agenda* built-in application switches to Week View mode when the PSION+SHIFT+W keypress is received.

On the Series 3a, the diamond key is used in the built-in applications to cycle around the various modes. Further, the built-in applications also allow the user to select which modes are to be included in the cycle.

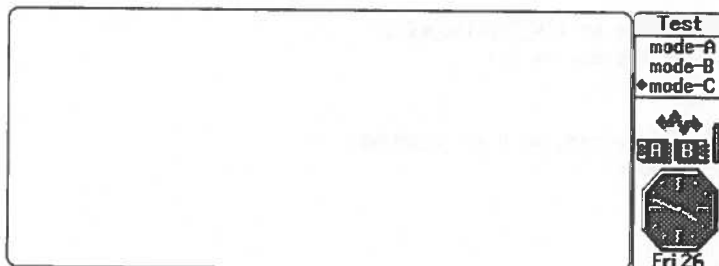
An Hwif program can also make use of the diamond key in this way simply by checking for the diamond key in its main event loop (or the appropriate event loop if more than one is used) and taking suitable action.

If a status window is displayed, the icon can be replaced by a list of modes. Optionally, a diamond symbol can be used to highlight the mode which is currently active and can be a useful visual aid.

This is achieved by using the window server functions `wsSetList` and `wsSelectList` (see the *General Window Server Functions* chapter in the *Window Server Reference* manual).

The following simple example resizes the main console window and displays a permanent status window alongside together with a list of three modes. Pressing the diamond key successively, causes the position of the diamond symbol in the status window to be shifted to lie alongside "mode-B" and then "mode-C" and then back to "mode-A" again.

Aside from the visual confirmation of a mode switch, the actual meaning and implementation of the switch is application dependent.



```
#include <p_std.h>
#include <wlib.h>
#include <hwif.h>

GLREF_D UWORD _UseFullScreen;

LOCAL_D WORD   keystat;
LOCAL_D INT    gc;

LOCAL_D UINT   modepos = 0;
LOCAL_D TEXT * modelist[] =
{
    "mode-A",
    "mode-B",
    "mode-C",
};

LOCAL_C VOID ReduceMainWin(VOID)
{
    P_EXTENT StatusExtent;
    W_WINDATA wd;

    wInquireStatusWindow(W_STATUS_WINDOW_BIG,&StatusExtent);
    wd.extent.tl.x = 0;
    wd.extent.tl.y = 0;
    wd.extent.width = 480 - StatusExtent.width;
    wd.extent.height = 160;
    wSetWindow(uFindMainWid(),W_WIN_EXTENT,&wd);
}

LOCAL_C VOID SpecificInit(VOID)
{
    ReduceMainWin();
    wStatusWindow(W_STATUS_WINDOW_BIG);
    gc = gCreateGC0(uFindMainWid());
    gBorder2(W_BORDER_TYPE_0,W_BORD_CORNER_4);
    wsSetList(3,&modelist[0],modepos);
}

LOCAL_C VOID SwitchMode(VOID)
{
    modepos++;
    modepos %= 3;
    wsSelectList(modepos);
    /* ... application specific mode switch code ... */
}
```

```
LOCAL_C VOID MainLoop(VOID)
{
    WMSG_KEY key;

    FOREVER
    {
        uGetKeyA(&keystat,&key);
        p_iowait();
        if (key.keycode & W_SPECIAL_KEY)
        {
            key.keycode &= (~W_SPECIAL_KEY);
            if (key.keycode == 'x')
                p_exit(0);
        }
        else if (key.keycode == W_KEY_DIAMOND)
            SwitchMode();
    }
}

GLDEF_C VOID main(VOID)
{
    _UseFullScreen = TRUE;
    uCommonInit();
    SpecificInit();
    MainLoop();
}
```

Note that setting `_UseFullScreen` to be `TRUE` indicates that this code is intended to be run on either the Series 3 or the Series 3a. For a full explanation of the significance of the variable `_UseFullScreen`, see the description of the function `uCommonInit` in the *Hwif Reference Documentation* chapter.

Menu bar interactions

The way an Hwif application initiates a menu bar interaction is simply to make the call

```
uPresentMenus()
```

where there are no explicit parameters.

The actual contents of the menu bar are communicated implicitly via the static variables `_cmds` and `_mdata` which various Hwif library calls expect to access. The application must ensure that these point to appropriate tables.

In brief, `_cmds` is the address of a table of the text names and accelerators of all the current menu commands of the application, and `_mdata` is the address of another table giving the names of the menu tiles, and how many commands there are in each tile.

Where `_cmds` must point

For example:

```
LOCAL_D TEXT *cmds[] =
{
    "nNew file",
    "oOpen file",
    "aSave as",
    "iInsert",
    "cCopy",
    "fFrame off",
    "xExit",
    NULL
};

GLDEF_D TEXT ** _cmds=(&cmds[0]);
```

This defines a menu bar currently with 7 menu items.

Note carefully:

- The text for each menu item starts with its accelerator
- The entire list is terminated by a NULL.

Where `_mdata` must point

For example, suppose that the above commands are split up into a "File" menu (the first three), an "Edit" menu (the next two), and a "Special" menu (the last two). Then the following definitions would be appropriate

```
LOCAL_D H_MENU_DATA mdata[] =
{
    "File",3,
    "Edit",2,
    "Special",2,
    NULL
};

GLDEF_D H_MENU_DATA * _mdata=&mdata[0];
```

Note that:

- Again, the data is terminated by a NULL
- The individual entries (one per menu tile) are given in the form of an `H_MENU_DATA` struct, which simply contains a `TEXT*` followed by a `UWORD`.

How to use `uPresentMenus`

If the user cancels, or if an error has occurred (eg out of memory), `uPresentMenus` returns 0. Otherwise, it returns the accelerator of the item chosen.

Accordingly, `uPresentMenus` is normally called in the following context:

```
LOCAL_C VOID TryExecuteCommand(INT keycode)
{
    keycode=uLocateCommand(keycode);
    if (keycode>=0)
        ManageCommand(keycode);
}

LOCAL_C VOID MainLoop(VOID)
{
    INT ret;

    FOREVER
    {
        uGetKey(&key);
        if (key.keycode&W_SPECIAL_KEY)
            TryExecuteCommand(key.keycode&(~W_SPECIAL_KEY));
        else if (key.keycode==W_KEY_MENU)
        {
            ret=uPresentMenus();
            if (ret>0)
                TryExecuteCommand(ret);
        }
        else ...
    }
}
```

That is, a menu bar interaction is initiated in response to receipt of a MENU keypress. The result, if positive, is dispatched to the Hwif library routine `uLocateCommand`, which converts the accelerator to an *index into the current table of menu commands* (as pointed to by `_cmds`). Finally, inside the routine `ManageCommand`, a switch statement (or equivalent) on the index is performed.

Note that there are *two* routes to the routine `ManageCommand`:

- the route via receipt of the MENU key and the presentation of the menu bar

- the route whereby the hot key is received directly, in the form PSION+ACCELERATOR.

In the latter case, the bit `W_SPECIAL_KEY` is set in `key.keycode`, so that the user's presumed intention of invoking a menu command via its accelerator can be detected early inside any get-event loop. This bit has to be masked out before the contents of the keypress is analysed by the routine `uLocateCommand`. In case the user has typed PSION together with an accelerator that does not currently exist for the application, the routine `uLocateCommand` returns -1. Otherwise, as stated above, it returns an appropriate index into the table of menu commands.

The ManageCommand routine

The `ManageCommand` routine of an application is one of its most important ones, ranking alongside `MainLoop` and (to a lesser extent) `main` itself as the locus of the controlling logic of the program.

For clarity, it seems best if the contents of `ManageCommand` just call other routines where the real work of each of the different menu commands is executed. This leaves `ManageCommand` as, primarily, an extended switch statement, branching on the command index number.

Changing menu bar contents dynamically

There are at least two methods of dynamically changing the contents of the menu bar.

In the first method, two different sets of tables could be declared, and the values of `_cmds` and `_mdata` be changed when required. This might be appropriate for an application with two modes that differ distinctly from each other.

Alternatively, for a more modest change, code such as the following suffices:

```
cmds[5]=(show_frame? "fFrame off": "fFrame on");
```

Restrictions on valid accelerators

On the Series 3, the allowed values for accelerators are the 26 lower case letters 'a' through 'z', together with the four arithmetic operator keys, '+', '-', '*', and '/' (though some of the last four values may change to something else on a non-English keyboard version).

On the Series 3a and *Workabout*, however, the allowed values for accelerators are all those allowed for the Series 3 *plus* the upper case letters 'A' through 'Z'.

It is not possible to specify a menu command *without* an accelerator. This means that at any one time, an application is limited to 30 first level menu commands on the Series 3 and 56 first level menu commands on the Series 3a and *Workabout*.

To handle upper case accelerators correctly, the `MainLoop` code fragment above could be changed as follows:

```
LOCAL_C VOID MainLoop(VOID)
{
    INT ret;
    INT code;

    FOREVER
    {
        uGetKey(&key);
        if (key.keycode & W_SPECIAL_KEY)
        {
            code = key.keycode & (~W_SPECIAL_KEY);
            if (key.modifiers & W_SHIFT_MODIFIER)
                code = p_toupper(code);
            TryExecuteCommand(code);
        }
        else if (key.keycode == W_KEY_MENU)
        {
            ret = uPresentMenus();
            if (ret > 0)
                TryExecuteCommand(ret);
        }
        else ...
    }
}
```

Grey underlining

Built in applications have the ability to add grey lines underneath menu items. This serves to group related menu items and can be a useful visual aid if used sparingly.

Note that grey need not be enabled for the main console window in order for this to work.

This feature is available on the Series 3a and *Workabout*, but not on the Series 3. It is discussed more fully in the *Hwif Reference Documentation* chapter.

Menu positions

The position of the menu item currently selected can be recorded by making use of the global variable `_MenuPositions`.

This is particularly useful in an application where more than one menu bar is used. The position of the menu item selected in each menu bar can be recorded so that the previously selected menu item can be highlighted when a menu bar is re-displayed.

This feature is available on the Series 3a and *Workabout*, but not on the Series 3. It is discussed more fully in the *Hwif Reference Documentation* chapter.

Presenting dialogs

Presenting a dialog consists of the following steps:

- one call to `uOpenDialog`, to begin building up the dialog contents
- one or more calls to `uAddDialogItem`, `uAddButtonList`, and/or `uAddChoiceList`, to add items to the dialog
- possibly, a call to `hDlgPosition`, to position the dialog to one side or corner of the screen
- one call to `uRunDialog`, to await the user's response.

Checking for run-time errors

The programmer ought to bear in mind that an out-of-memory error can occur at any of the above stages, in which case the flow of execution must be terminated at once. In practice, this is very simple to do, since the Hwif `uxxx` calls all automatically inform the user if any error arises, and automatically clean up any temporary dialog resources that are no longer required.

For example, to present a dialog with a title (*Use*) and a choice list (with prompt *Clipboard*):

```
LOCAL_C VOID ChangeClipboardUsed(VOID)
{
    if (uOpenDialog("Use"))
        return; /* out of memory */
    if (uAddChoiceList("Clipboard",&using,"1","2","3","4",NULL))
        return; /* out of memory */
    if (uRunDialog()<=0)
        return; /* out of memory, or user cancelled */
    WriteClipboardText();
}
```

Items that can be added to Hwif dialogs

These are precisely the same as in OPL/w, namely:

- choice lists
- action lists of buttons
- plain text items
- numeric editors
- floating point editors
- time editors and date editors

- text editors (both scrolling and non-scrolling)
- secret data input boxes
- filename selectors and filename editors.

In most of these cases, the item is associated with a so-called *live variable*, which specifies the initial value of the item (when the dialog is made visible), and which may be changed when the dialog is successfully completed. The type of this live variable varies from item to item.

Thus in the above example, the (global) variable `using` is the live variable for the choice list.

Longer choice lists

In some cases, the list of choices presented in a particular choice list may be too long for a choice list to be conveniently defined simply by one call to `uAddChoiceList`.

The three functions `uBeginDCL`, `uGrowDCL`, and `uAddDCL` exist to help out with these so-called *dynamic* choice lists (the name reflects the fact that the contents of the choice list are built up over a few lines of code, rather than just being defined statically; in some cases, the contents in the list may change between different invocations of the dialog, to reflect changing run-time circumstances).

For example, the following routine builds up a choice list whose contents are the twelve month names:

```
LOCAL_C INT AddMonthChoiceList(UWORD *pmonno)
{
    H_DI_CHOICE ch;
    TEXT mon[32];
    INT i;

    if (uBeginDCL(&ch))
        return(-1);          /* report failure to caller */
    for (i=0; i<12; i++)
    {
        p_nmmon(&mon[0],i);
        if (uGrowDCL(&ch,&mon[0]))
            return(-1);      /* report failure to caller */
    }
    return(uAddDCL("Month",pmonno,&ch));
}
```

Alternatively, the function `hSetVarrayInChlist` can be used to build up a choice list, especially if the list might contain more than 255 items.

For example, by modifying the above code, the same effect can be achieved as shown below:

```
#define C_VASTR 6
#define O_VA_APPEND 7
#define OLIB_CAT 1

LOCAL_C INT AddMonthChoiceList(INT item_no,INT *pmonno)
{
    UWORD used = 0;
    VOID *pvarray;
    TEXT mon[32];
    INT i;

    if (uAddChoiceList("month",&used,NULL))
        return(-1);          /* failure */
    pvarray = p_new(OLIB_CAT,C_VASTR);
    for (i = 0; i < 12; i++)
    {
        p_nmmon(&mon[0],i);
        p_send3(pvarray,O_VA_APPEND,&mon[0]);
    }
    hSetVarrayInChlist(item_no,(*pmonno),pvarray);
}
```

Note that error handling in this code is incomplete.

Typical dialog usage

Applications will in many cases wish to call dialogs *in a loop*, as follows:

```
LOCAL_C VOID OpenFile(VOID)
{
    H_DI_FSEL open;
    TEXT openbuf[P_FNAME_SIZE+2];
    VOID *newfcb;

    SetupFilename(&openbuf[0]);
    open.flags=H_FILE_PICK_SELECTOR;
    open.fname=&openbuf[0];    /* initialise filename selector */
    do
    {
        if (uOpenDialog("Dump"))
            return;
        if (uAddDialogItem(H_DIALOG_FSEL,"File:",&open))
            return;
        if (uRunDialog()<=0)
            return;
        openbuf[1+openbuf[0]]=0;
    } while (TryOpenFile(&openbuf[1],&newfcb));    /* until open succeeds */
    ChangeOverTo(&openbuf[1],newfcb);
}
```

Schematically:

```
InitialiseLiveVariables();
FOREVER
{
    if (No memory for dialog)
        break;
    if (User cancels)
        break;
    if (Dialog choices validate okay)
    {
        PerformAction();
        break;
    }
}
```

In such a case, relevant live variables need to be initialised *before entering the loop*. Then if the user mistakenly selects (eg) the wrong file from a directory, this choice will remain in the dialog when it is presented again (along with an appropriate error message), so that it is easy for the user to adjust the choice to what was intended. At the same time, users will be able to see what they typed into the dialog the first time.

It is possible to discover the last key press and key modifiers handled by the system on the application's behalf. This is particularly useful on exit from `uRunDialog`.

A number of key press combinations cause a dialog to terminate; amongst others, they include ESC and HELP. Knowing which key press combination caused the dialog to terminate, enables the application to take appropriate subsequent action.

For more information, see the description of the `hLastSystemKey` function in the *Hwif Reference Documentation* chapter.

Dialog underlining

Built in applications have the ability to add or remove a solid underline to components in a dialog. This serves to group related dialog components and can be a useful visual aid if used sparingly.

Hwif programs can also do this by calling the function `usetDialogUline` and specifying the position of the dialog component within the dialog and indicating whether underlining is to be added or removed.

This feature is available on all machines, although there is a difference in behaviour on the Series 3 as opposed to the Series 3a and *Workabout*.

Further detail on this can be found in the *Hwif Reference Documentation* chapter.

Help dialogs

Hwif contains support for accessing the same Help engine as used by the in-built applications on the Series 3.

The function `hHelpSubSystem` can be called in response to suitable key-presses from the user (eg the HELP keypress).

The Help engine is resource-based and any serious user of `hHelpSubSystem` must create a resource file (using the tool `RCOMP.EXE`) containing resources defining the hierarchy of Help text.

Typically, a `MainLoop` routine might contain the following code fragment:

```
...
if (key.keycode == W_KEY_HELP)
    hHelpSubSystem(QUERY_HELP, QUERY_HELP_INDEX);
...
```

where the two parameters are the resource IDs of the "top-level" Help resource and the "index" set of Help resources respectively.

Like `uPresentMenus` and `uRunDialog`, `hHelpSubSystem` only returns when the entire Help operation has been completed by the user. It does its own error handling internally, automatically presenting suitable error messages where necessary.

Help Resources

As stated above, the function `hHelpSubSystem` requires two resource IDs as parameters. These reference instances of the `HELP_ARRAY` resource (defined in the *Hwif reference documentation*). These, in turn, may reference instances of `STRING`, `TOPIC_ARRAY` and yet other `HELP_ARRAY` resources, forming a potentially complex hierarchy.

Note that in the definition of the resource struct `HELP_ARRAY`, the three fields must always appear in the given order. When declaring an instance of the resource struct `HELP_ARRAY`, the fields can come in any order (true for any resources declared in a resource file). The reason that it may be natural to rearrange the fields is that the displayed form of a general Help screen is:

1. "topic" text in the title line (in bold).
2. Any `STRING`s in "strlst" come next.
3. any associated topics defined by "topic_id" come last (bulleted and in bold).

The "topic_id" field of any `HELP_ARRAY` resource, if present, must always reference an instance of the `TOPIC_ARRAY` struct (defined in the *Hwif reference documentation*). In turn the values in "id_lst" refer to further `HELP_ARRAY`s as shown schematically below:

```
HELP_ARRAY (topic_id)--> TOPIC_ARRAY
                        |
                        (id_lst item)--> HELP_ARRAY ... etc
                        |
                        (id_lst item)--> HELP_ARRAY ... etc
                        |
                        ...
```

For example, the following definition has resource ID `world_basics` and contains all three possible types of field - "topic", "strlst" and "topic_id".

```
RESOURCE HELP_ARRAY world_basics
{
    topic = "World";
    strlst =
    {
        STRING {str = "To find a city, type first few letters";}
        STRING {str = "";}
        STRING {str = "World button locks to cities in one country";}
    }
    topic_id = world_extra;
}
```

However, it is more common to omit at least one of these fields in any one `HELP_ARRAY`.

In practice, Help hierarchies tend to be built from what can be described as "top-level" variations of `HELP_ARRAYS`, where the "strlst" fields are omitted, and "bottom-level" variations, where the "topic_id" fields are omitted

The "top-level" variations result in the presentation of bold, bulleted lists of further topics on which help can be had. For example:

```
RESOURCE HELP_ARRAY query_help
{
    topic = "Query";
    topic_id = query_help_data;
}
```

The "bottom-level" variations result in the presentation of UNbold, UNbulleted text strings. For example:

```
RESOURCE TOPIC_ARRAY query_help_data
{
    id_lst =
    {
        query_list,
        query_func,
        query_memories,
        query_percent,
        query_tips
    };
}
```

In turn, `query_memories`, for example, is the ID of the resource:

```
RESOURCE HELP_ARRAY query_memories
{
    topic = "Memories M0-M9";
    strlst =
    {
        STRING {str = "Top line shows current memory";},
        STRING {str = "'M In', 'M+' etc work on it";},
        STRING {str = "";},
        STRING {str = "Change or set with 'Change memories'";}
    };
}
```

Thus, calling `hHelpSubSystem` with first parameter `QUERY_HELP` displays `QUERY` in the title line of the Help display followed by a list of bold, bulleted items of which `MEMORIES M0-M9` is the third. Selecting this item from the Help menu causes the above text to be displayed.

Note that neither the system code at run-time, nor the resource compiler offers any assistance in word-wrapping strings of Help text. The developer must ensure that individual lines do not become so wide that a run time "Too wide" error message is given.

The `HELP_ARRAY` identified by the index resource ID parameter to `hHelpSubSystem` (ie the second parameter) is only referenced when the user requests an "Index" of all available Help topics, either by pressing `CONTROL+HELP` inside the Help subsystem or by selecting the "Index" item and pressing `ENTER`. Either way, the system code constructs a sorted alphabetical list of topics from two sources:

1. the help index at resource 112 in the "system" resource file (which the system code automatically opens on behalf of all Hwif applications).
2. the `HELP_ARRAY` resource referenced by the caller's index resource ID parameter (if non-zero).

In constructing the list, the system code ignores any "strlst" and "topic" fields in the top-level resource.

A measure of context sensitive help can be achieved by allowing the values passed to `hHelpSubSystem` to depend on the program state.

Finally, whenever a Help screen is constructed, the system code automatically appends a reference to "Help on help" and "Index" provided that these are not already provided by the developer.

Help dialogs (an older alternative)

In previous versions of Hwif, it was not possible to access the built-in Help subsystem from an Hwif application. However, two utility routines were available (and still are!) in the Hwif library that allowed the presentation of a broadly comparable but possibly less satisfying Help dialog suite.

The routine `WDialogMenu` allows the presentation of a menu-like dialog broadly equivalent to the top level in a Help dialog suite. Typically, an application would launch this dialog when the `HELP` key is pressed. The application supplies a range of topic titles, and the user cursors up and down in standard manner to select a topic of further interest. The value returned from `WDialogMenu` informs the application which topic was selected.

The application can then make use of the routine `WDisplayText` to present a dialog containing up to seven lines of additional textual information (for the body of the help topic selected). When this is exited, the application can, if desired, present the top-level dialog again.

Line editors and multi-line editors

In addition to the various editors that are available as items in dialogs, the Hwif library also allows access to single- and multi-line editors, outside of the context of dialogs.

Such editors allow the user to alter data, without having to invoke a dialog box for this purpose. Advantages include:

- compared to a corresponding (scrolling) editor in a dialog, a multi-line editor allows the user to see more text at one time, and the text can contain embedded carriage returns
- any Find text (for example) can remain permanently visible on the screen, in its own edit box, instead of being visible only when the user requests the presentation of a Find Dialog
- if the entries in for example a Diary are laid out on screen, the user can edit them *in place*, rather than having to use a dialog and lose sight of the overall screen layout in the meantime.

Invoking these editors via Hwif library calls does not significantly add to the total size of the code of an application, since the Hwif library calls merely provide access to editing functionality that is already present in the Series 3 ROM.

Editing features supported

Hwif editors allow applications to make use of the following features:

- in multi-line editors, text is automatically word-wrapped at the visible right margin specified; users can start new paragraphs by pressing the `ENTER` key
- in multi-line editors, the display automatically scrolls vertically when required
- the user can show or hide paragraph ends and spaces
- the text can be displayed bold, italicised, underlined, monospaced, and/or double height (though the font style cannot vary from word to word inside any one editor)
- the width of an editor can be changed dynamically on request, for example if the user hides or shows a permanent status window
- it is possible to display a side cursor in the left margin; the width of the main (flashing) cursor can also be controlled (and can be set to zero)
- the functions cut, copy, paste, evaluate, find, and replace are all available.

Presenting an editor

The minimum steps required to include an editor in an application are the following:

- create the editor, using the call `HEBOpen`
- when appropriate, emphasise it, using the call `HEBEmphasise`, so that it displays a flashing cursor
- from time to time, pass it relevant incoming keypresses, using the call `HEBHandleKey`
- when the application needs to know the current contents of the editor, the call `HEBSenseText` can be made.

The call `heBOpen` passes a filled-in `H_EDIT_BOX` struct, to customise the editor. This specifies (among other things) the position of the top left of the editor on the screen, the maximum number of characters that the editor is to accept, the width and height of the editor, the font style to use, and the spacing between lines.

For example, the following code fragment creates an editor four lines deep, of width 12 pixels less than the current screen width, and which supports clipboard functionality and a left cursor:

```
H_EDIT_BOX heb;

heb.win=MainWid;           /* ID of main window */
heb.maxchars=255;
heb.vulen=ScreenWidth-12;
heb.vislines=4;
heb.pos.x=5;
heb.pos.y=36;
ebH=heBOpen(H_EDIT_BOX_VISLINES|H_EDIT_BOX_LEFT_CURSOR|H_EDIT_BOX_CLIPBOARD,&heb);
```

The return value `ebH` (the *handle* of the edit box) should be used to identify this particular editor (as opposed to others) in subsequent `heBxxx` calls. The call returns zero if there was insufficient memory to create the editor.

In the above example, the variable `ScreenWidth` has previously been set up by the application to the current width of the screen (this can vary depending on whether a permanent status window is visible). The variable `MainWid` is the ID of the main screen window, as discussed below in connection with the Console.

The call `heBOpen` creates an editor without any text in it. Text can be placed into the editor in a variety of ways, chief amongst them the call

```
heBSetText(VOID *ebH,TEXT *pb,INT blen)
```

In using the calls `heBSetText` and `heBSenseText`, it is important to understand that Hwif editors store text internally in one contiguous buffer. The buffer itself is always terminated by a zero. Paragraph ends are recorded using the character 13 (`\n`). Thus text set into an editor should not contain any zero (except, possibly, after `blen` characters - in which case it is harmless). For example,

```
heBSetText(ebH,"Hello world\nThis is Hwif",24);
```

The call `heBSenseText` simply returns a `TEXT*` pointer to the editor's own copy of the text being edited. Clearly, this has to be treated with care. A copy of the text may have to be made (using, for example, `p_scpy`) before closing down the editor or otherwise changing its contents.

As well as the *textual* content of an editor initially being zero, the *cursor* position also starts off at zero; likewise there is, by default, no *select region*. These settings can be overridden by means of the call `heBSetSelect`. If required, there is also a corresponding call `heBSenseSelect`.

Word wrapping can, effectively, be turned off by using the `heBSetMargin` function.

For more details of these and other editing calls, see the reference section, or the worked example applications.

Applications with more than one editor

Applications with more than one editor can present an impressive appearance to users.

What the user sees is two or more regions of the screen - such as the Find Window and the Record Window in a database application - each enclosed in its own border, and each housing an editor. The user typically navigates between these windows using the TAB key to switch focus, and editing keys are directed to whichever editor currently has the focus.

To implement such a set up, an Hwif programmer needs to take the following steps:

- the sizes and positions of the various screen components have to be carefully calculated (see the example applications for some guidance on this)
- the requisite number of editors have to be created, and their handles stored
- a variable has to be dedicated to recording which of the editors currently has the focus
- for each editor, a graphics call such as `gBorderRect` has to be made, to produce an appropriately shadowed boundary (with a heavier shadow for the editor possessing the focus).

On receipt of the TAB key, the application switches focus, by means of two calls to `gBorderRect` (one to remove the shadow from the editor that is losing the focus, and one to apply a shadow to the editor that is gaining it), and two calls to `HEBEmphasise`. The effect of these latter calls is to control whether a flashing cursor is displayed, and whether a select region is highlighted.

Thus part of the code in the `MainLoop` of an application with three editors could be as follows:

```
switch (key.keycode)
{
    ...
case W_KEY_TAB:
    RotateFocus(key.modifiers&W_SHIFT_MODIFIER);    /* forwards or backwards */
    break;
case W_KEY_RETURN:
    if (emph!=2)
        break;    /* else fall through */
default:
    hEBHandleKey(edit[emph],key.keycode,key.modifiers);
}
```

In this application, the handles of the three edit boxes are stored in `edit[0]` through `edit[2]`, the variable `emph` records which currently has the emphasis, and of the three editors, only the third is multi-line (hence the test in the `W_KEY_RETURN` branch). The effect of `SHIFT+TAB` is to rotate the focus in the opposite direction to plain `TAB`.

Edit boxes and saved file versions

One important difference between restricting users to edit data via dialogs, and allowing them to edit the data in place, in single- or multi-line editors, is that menu commands can arrive at any time in the middle of editing in the second case (but not in the first). The point is that access to the menu bar is impossible while a dialog is in place; not so when text in an edit box is being edited.

Applications which use edit boxes may therefore have to check, before carrying out any menu command, that their own record of the contents of the edit boxes is up to date.

To this end, the routine `HEBSenseChanged` exists, which reports whether or not an edit box has had its contents changed. There is also a routine `HEBClearChanged`, which clears the internal "changed" flag maintained by the edit box.

When file-based data is being edited via a dialog, an application can often choose to write any changes to file, immediately the dialog completes (especially in the case of a database application). But when data is being edited via an edit box, the application obviously cannot write out any changes *every time a keypress is received*. This leads to there being a potential difference between the *file* version of some data, and the current *in-memory* version. It is up to applications to keep careful track of this difference, and to decide when changes should be committed to file.

Printing

Hwif library routines provide support for applications:

- invoking the standard Print Setup dialog suite (specifying page size, margins, headers and footers, and so on)
- actually printing.

Printing features supported

Any printing from an Hwif application automatically conforms to the parameters specified by the user via the Print Setup dialog, and is automatically directed to the file or device specified by the user in the Printer Setup dialog in the System Screen. As the printing takes place, the standard Printing dialog is presented, informing the user of the page currently being printed, and containing a Cancel button to allow the printing to be abandoned.

Rom resident code performs the pagination and any word-wrap required, and ensures that any headers and footers specified by the user are appropriately positioned. That is, including Hwif printing library calls in an application does not significantly add to the total size of the code of an application, since these library calls merely provide access to functionality that is present in the Series 3 ROM.

All that an application needs to provide is a *callback* function which provides the data for each new line (or series of lines, if word-wrap is required) to be printed.

The Print Setup dialog

In order to enter the Print Setup dialog suite, an application only needs to include the single line of code

```
hPrintSetupDialog();
```

There is no requirement for the application to record the values chosen by the user in this dialog suite, as ROM resident code does this automatically.

The basic mechanism of printing

As an example of how to print, consider an application which stores its data as a series of records accessed via an index, `table[]`. These records in turn consist of a date and time, and a text string, all defined by the struct

```
typedef struct
{
    ULONG sdate;      /* time and date */
    UWORD tlen;       /* length of text string */
    TEXT *pb;         /* address of text string */
} ENTRY;
```

The following two routines suffice to print the application's data:

```
LOCAL_D UWORD PrintRec;      /* entry currently being printed */
LOCAL_D UWORD PrintState;    /* which PART of the entry is currently being printed */
LOCAL_D TEXT PrintBuf[H_TIME_LN_DATE_STRING];

LOCAL_C INT PrintLine(H_PRINT *pr)
{
    ENTRY *pent;

    pent=&table[PrintRec];
    switch (PrintState++)
    {
        case 0:
            pr->blen=SdateToBuf(&PrintBuf[0],&pent->sdate);
            pr->buf=&PrintBuf[0];
            pr->style|=H_PRINT_STY_BOLD;
            break;
        case 1:
            pr->buf=pent->pb;
            pr->blen=pent->tlen;
            break;
        case 2:
            pr->blen=0;
            PrintState=0;
            PrintRec++;
            return(PrintRec!=count);
    }
    pr->flags|=H_PRINT_KEEP;
    return(TRUE);
}

LOCAL_C VOID PrintAll(VOID)
{
    if (!count)
    {
        winfoMsg("Nothing to print");
        return;
    }
    PrintRec=0;
    PrintState=0;
    hPrint(PrintLine);
}
```

Of these two routines, the latter (`PrintAll`) is the controlling one. The general form of this routine is actually as follows:

```
CollectPrintDetails();
CheckIfReallyNeedToPrint();
InitialisePrintStateVariables();
hPrint(PrintLine);
PostPrintMessage();
```

In all cases, the centre-piece of a `PrintAll` routine is the line

```
hPrint(PrintLine);
```

or equivalent. When the flow of program execution reaches this line, it remains inside the `hPrint` routine until printing has completed (either naturally, or on account of the user terminating, or on account of some kind of error). Thus any following line of code, such as a call to `PostPrintMessage`, is executed only *after* printing is complete. In this sense, `hPrint` is similar to the calls `uRunDialog` and `uPresentMenus`; a lot can take place on the screen without the program progressing any further through application code. However, in one crucial respect, `hPrint` differs from these other calls; the application-supplied *callback function* `PrintLine` is repeatedly visited from *inside* the `hPrint` call.

The `PrintLine` callback function

Each time system code calls the `PrintLine` function, the application has to supply the data for the next line (or series of lines) to print.

It is the responsibility of the application to keep an independent record of how far the printing has progressed - so that the appropriate data can be passed each time. This record - the so-called *Print state variables* - has to be in some static data (or in an alloc cell whose *handle* is a static variable). In the above example, the variables `PrintState` and `PrintRec` play this role: `PrintRec` counts through the records (from 0 through `count-1`), and `PrintState` records which *part* of each record is currently being printed:

- the text corresponding to the date and time
- the main text of the record
- a possible blank line underneath the record, to separate it from the following one.

Each time `PrintLine` is called, the application has to fill in parts of a passed `H_PRINT` struct. The parts that an application is most likely to want to write to are:

<code>blen</code>	the length of the text for the line (or group of lines)
<code>buf</code>	the address of a buffer containing the text to print
<code>style</code>	possible further embellishment of the font style selected by the user in any Print Setup dialog
<code>flags</code>	the application may wish to <i>or</i> in the bit <code>H_PRINT_KEEP</code> , to request the system software to keep this line (or group of lines) together on the same page with the following line, if possible; another potentially useful flag is <code>H_PRINT_PAGE</code> , to force the emission of a form feed before the line is printed.

Additionally, the return value from `PrintLine` has the following significance:

- the application should return `FALSE` when it has no more data to print
- otherwise, the application should return `TRUE`.

The location of the print buffer

Note that `pr->buf` must point to a buffer *that will continue to exist* after the routine `PrintLine` has returned.

In the above example, the main body of text can be printed simply by setting `pr->buf` equal to the application's own pointer to where this text is stored. However, the text corresponding to the time and date is another matter, since the application evidently only stores this in the form of a `ULONG`. The application routine `sdateToBuf` converts the time and date from a `ULONG` into text form (presumably using the `Hwif` time-text utility functions, discussed later). However, it would be a severe error to declare a buffer to hold this textual representation as an automatic on the stack inside `PrintLine`.

The Print Details dialog

Although the above example lacks such a dialog, it is possible for an application to present its own dialog prior to proceeding with a print.

The purpose of this dialog - if not only to confirm that the user wishes printing to go ahead - could be to collect additional parameters affecting the way the printing is done. For example, the user could be asked to choose whether all records should be printed, or only a selected set (say only those records which are somehow *tagged*). Again, the dialog could control whether each new record should begin on its own new page.

Note that any such Print Details dialog differs in function from the Print Setup dialog available via `hPrintSetupDialog`:

- the Print Setup dialog is common to all applications, whereas Print Details dialogs differ from application to application
- system code takes care of recording and implementing the choices made from the Print Setup dialog, but it is up to applications to record and implement choices made in any Print Details dialog.

Word wrapping during printing

In most cases, applications have no need to be aware of any word-wrapping that may take place during printing. Whether or not some text is printed on one line or over more than one line depends, after all, on choices made by the user in the Print Setup dialog - such as the page size, the left and right margins, and the default printing font. In all cases, the ROM printing code automatically ensures that the specified margins are respected, with excess text being placed on the following line.

Occasionally, however, an application may wish to specially indent subsequent lines of a wrapped paragraph. This can be achieved by using the Hwif library call `hPrintSetSI` - which sets the *Subsequent Indent* for any wrapped lines.

At the same time, *any* line can be arbitrarily indented, by means of writing to the indent field of the passed `H_PRINT` struct.

The units any such indents are expressed in vary considerably from printer to printer. To guide the application as to suitable values, two additional calls are available:

<code>hPrintSenseBufWidth</code>	returns the width, in printer units, of a specified buffer (when printed in the selected default printer font)
<code>hPrintSensePageWidth</code>	returns the width, in printer units, of the paper being printed on (minus its margins).

Printing the contents of multi-line editors

Recall that Hwif multi-line editors use the character 13 (`\n`) to record paragraph ends.

Accordingly, if data produced in multi-line edit boxes is to be printed, it should first be scanned for embedded `\n`'s, along the following lines:

```
case 2:
    PrintBuf=index[PrintRec].note.buf;
    PrintLen=index[PrintRec].note.len;
default:
    pr->buf=PrintBuf;
    ind=p_bloc(PrintBuf,PrintLen,'\n');
    if (ind>=0)
    {
        pr->blen=ind;
        PrintLen-=ind+1;
        PrintBuf+=ind+1;
    }
    else
    {
        pr->blen=PrintLen;
        PrintState=0;
        PrintRec++;
    }
```

Time-text utility functions

The Hwif time-text functions provide a convenient way of generating textual representations of times and/or dates, that reflect the user's preferences as given in the Formats dialog in the *Time* application.

In general, a textual representation of time and/or date consists of a combination of some of the following components:

- numerical representations of the day in the month, the month in the year, the year, and the century
- numerical representations of the hour, the minute, and the second
- the name of the day in the week, and the name of the month in the year - each of which may be abbreviated
- a suffix (such as *th* or *rd*) after the day number in the month
- an *am* or *pm* indicator
- time and date separators (such as colons and slashes).

The `HTTxxx` functions provide textual representations as combinations of the above, taking data from:

- a time and/or date that has previously been specified, using the `HTTSetTime` call (which accepts any of the `P_DAYSEC`, `P_DATE`, or system-time representations)
- formatting preferences previously indicated by the application, using the calls `HTTSetFormat` and, possibly, `HTTSetAbbreviations`
- the user's current preference for whether time should be *am/pm* or 24 hour, for whether the month should come before or after the day, and for what the date and time separators should be.

The resultant text itself is obtained by a call to `HTTSenseText`.

Each of the above calls has to pass a *handle* that has previously been obtained by a call to `HTTOpen`. This allocates resources that the subsequent calls access. When these resources are no longer needed, they can be freed by a call to `HTTClose`.

Default textual representation

In the absence of a call from `HTTSetFormat`:

- any time string generated consists of hours, minutes, and seconds (all expressed numerically), together with the current time separators
- any date string generated consists of the day in the month, the month in the year, the year, and the century (again, all expressed numerically), together with the current date separators.

In the absence of a call from `HTTSetAbbreviations`, any day and month names generated are given in full, without being abbreviated.

Refreshing the format on returning to foreground

Ideally, every time an application making long-term use of the time-text functions is brought into foreground, it ought to reset the format of any time-text channels it currently has open. This ensures that any changes made by the user in the meantime (when the application was in background) are picked up.

For example, part of the `MainLoop` of an application displaying time in textual form might be

```
if (key.keycode & W_EVENT_KEY)
{
    if (key.keycode == CONS_EVENT_FOREGROUND)
    {
        UpdateTimeFormat();
        Display();
    }
}
else
...
```

If the application displays time without any seconds, and displays the month name, the day name, and a suffix after the day number, the contents of `UpdateTimeFormat` could be simply

```
LOCAL_C VOID UpdateTimeFormat(VOID)
{
    hTtSetFormat(ttH, H_TIME_FORMAT_NO_SECONDS|H_TIME_FORMAT_SUFFIX|
                H_TIME_FORMAT_DAY_NAME|H_TIME_FORMAT_MONTH_NAME);
}
```

This call re-asserts the application's requirements, but allows the user's latest preferences on matters of time and date separator, 12 or 24 hour clock, and month coming before or after date, to be picked up too.

Date/Time-text utility functions

These are a small set of functions which construct and manipulate stand-alone date/time text editors. As such, they need not exist within a dialog. In some respects they are similar to the time-text functions but have fewer date/time formats.

The textual representation of date is limited to the ten characters DD/MM/YYYY. For example: 20/07/1993.

The textual representation of time can be in any of the following formats:

- HH:MM:SS representing either a time duration *or* a time of day in hours, minutes and seconds. If it represents a time of day then, depending on system settings, the hours can be in either the 24-hour or the 12-hour format. If the latter, the string will be followed by the characters am or pm.
- HH:MM representing either a time duration *or* a time of day in hours and minutes. If it represents a time of day then, depending on system settings, the hours can be in either the 24-hour or the 12-hour format. If the latter, the string will be followed by the characters am or pm.

The formatting can be done either at creation time in a call to `hDTOpen` or when setting a value in a call to `hDTSet`.

The value and formatting information can be retrieved by a call to `hDTSense`.

A call to `hDTSelfCheck` performs a validation on the value currently held.

A call to `hDTHandleKey` handles the current keypress, assuming that there is suitable code which can capture keypress events.

A call to `hDTEmphasise` can switch the emphasis for the current date/time editor. In other words, depending on whether the second parameter is set to `TRUE` or `FALSE`, text highlighting can be turned on or off. The emphasis can also be set at creation, i.e. on a call to `hDTOpen`.

Each of the above calls must pass a *handle* that is obtained by the call to `hDTOpen`. This allocates resources that the subsequent calls access. When these resources are no longer needed, they can be freed by a call to `hDTClose`.

It is worth a reminder that the position and width of the text editor on the screen can also be specified at creation, i.e. on a call to `hDTOpen`.

Hwif, the Console, and screen output

The visual output of an Hwif application generally consists of:

- intermittently, menus and dialogs
- more permanently, single- or multi-line editors
- a temporary or permanent status window
- additional graphics and text effects.

These additional graphics and text effects are achieved by a mixture of Wlib calls such as `wInfoMsg`, `gPrintBoxText`, `gBorder`, `gInvObloid`, `wsEnable`, `gClrRect`, `wScrollRect`, and so on.

In comparing these graphics calls with simpler console routines such as `p_puts`, three possible difficulties emerge:

- there is a significant learning curve of new functions and new function names
- console routines, being *text-based*, automatically take care of positioning the cursor, whereas more painstaking calculation is, inevitably, required with graphics functions
- before making any of the graphics calls, a significant amount of initialisation has to be undertaken, involving *connecting to the Window Server* as well as creating *windows* and *graphics contexts* when required.

Three options for graphics output

In general, there are essentially *three* options for application screen output:

- stick strictly to console services such as `p_printf` and `p_puts`
- use the routines in the Window Server library, but using *windows with backed-up bitmaps*, to avoid the need to undertake the conceptually more taxing burden of *do-it-yourself window redrawing*
- embrace the full philosophy of window redrawing.

Hwif applications fall decidedly into the second of these three options. Routines such as `p_printf` produce an output that is clearly inferior, in graphics quality, to that of the menus, dialogs, and editors in the Hwif arsenal.

The initialisation routine `uCommonInit` hides the complications of actually connecting to the Window Server - thus alleviating one of the possible disadvantages mentioned earlier, as regards using Window Server calls. Once the call `uCommonInit` is complete, the connection to the Window Server has already been established (there is no need for any independent call such as `wStartup`), and a backed-up window the size of the screen has been created. Similarly, the routines `uGetKey` and `uGetKeyA` hide the complications of receiving and decoding events from the Window Server.

At the same time, it is just as simple to invoke graphics routines such as `wInfoMsg` (which produces an information message at the bottom right corner of the screen) and `wSetBusyMsg` (which produces a flashing *busy* message) as it is to invoke `p_printf` and `p_puts` - provided, that is, that you have learned of the existence (and the names) of these graphics routines.

On the other hand, there is no requirement to go the full extent of adopting the window redraw mechanism that ultimately gives the best performance from the window server. Just as console i/o is *inappropriately primitive*, compared to the i/o of Hwif dialogs, edit boxes, and menus, the full redraw mechanism is *inappropriately advanced*.

Indeed, the three stages of graphics output listed above correlate closely with the three general stages through which programmers can develop, in acquiring fuller mastery of the Series 3 software:

- use only of console routines is appropriate to an initial encounter with the Series 3, and for applications whose user interface is unimportant
- use of the graphics calls (but not of the intricacies of doing your own redraws) matches applications at the Hwif level of sophistication
- application programmers for whom high performance is vital must adopt not only the concept of redrawing windows but also the Psion proprietary object-oriented development system.

Practical acquisition of graphics techniques

As far as acquiring familiarity with the varied Window Server library calls is concerned, it is intended that the accompanying Hwif example applications will prove useful in this regard (as well as in regard to illustrating the Hwif library calls). These provide sufficient illustrations for would-be graphics programmers to develop enough confidence to subsequently branch out, by themselves, into the wider reaches of the Window Server reference manual.

Two of the main types of graphics displays on the Series 3 are amply treated within these examples: a series of edit boxes (each with their own border), and a scrolling vertical list. Hiding and showing permanent status windows is also covered more than once.

Hwif opens the console channel

As a matter of implementation, the call `uCommonInit` which prepares the ground for all subsequent Hwif calls itself involves opening the console device (if it is not already open). As is standard, the handle of the console control block is written into the static `winHandle`.

Almost all of the time, Hwif programmers can be oblivious of this implementation decision, and need make no reference to `winHandle`. The supplied routines `uGetKey`, `uGetKeyA`, and `uKeyPressOutstanding` hide some of the details of the interaction with the console.

However, any console program can, by default, be terminated at any time by the user simply pressing PSION+ESC. In case this is undesirable, an application should make the call

```
uEscape(FALSE);
```

during its `SpecificInit` routine.

An important point to note is that if the global variable `_UseFullScreen` is zero on entry to the call to `uCommonInit`, the console device is initialised in compatibility mode; in other words, the behaviour and appearance of the console on the Series 3a or the *Workabout* emulates that of the console on the Series 3.

The main window

At the end of the call to `uCommonInit`, a graphics window exists, the size of the whole screen - 240 by 80 pixels on the Series 3, 480 by 160 pixels on the Series 3a and 240 by 100 pixels on the *Workabout* (in full Series 3 compatibility mode on the *Workabout* there is a gap of 10 pixels both above and below the window).

In practice, it is rarely necessary for Hwif programs explicitly to create any additional windows.

Many graphics calls need to know the window ID of the window they are to operate upon. The Hwif routine `uFindMainWid` returns this ID.

Typically, Hwif applications include a line such as

```
MainWid=uFindMainWid();
```

early in their `SpecificInit` routines.

Grey

On the Series 3a and the *Workabout*, drawing can be done not only in black & white but also in grey. To draw grey in the main console window, it must first be enabled.

This is done by calling the function `uEnableGrey` which is more fully described in the *Hwif Reference Documentation* chapter.

Ideally, if grey is to be used, it should be enabled early in the life of the application, preferably during the initialisation phase. On no account should grey be used before being enabled - the application is liable to fail.

Note that `uEnableGrey` changes the ID of the main console window. Therefore, ensure that `uFindMainWid` is called to fetch the new ID.

Dual mode applications

In general, applications which are intended to run on the Series 3 as well as on the Series 3a and/or *Workabout* often need to be able to distinguish the type of machine on which they are running.

For example, grey is available on the Series 3a and *Workabout* but not on the Series 3. Applications that are to run on more than one type of machine may wish to make use of such additional features if they are available.

Global variables such as `_UseFullScreen` can be used to help an application differentiate between the different situations. This is discussed in more detail in the description of the function `uCommonInit` in the *Hwif Reference Documentation* chapter.

One point that needs to be discussed here though is the question of application icons, since the form of icon required for the Series 3 is different from that needed for the Series 3a and *Workabout*. For

applications that are intended to run on the Series 3 and either or both of the other machines, a *.pic* file can be created to hold two separate icons.

In this situation, the *.pic* file will first contain a bitmap, 24 pixels wide by 24 pixels deep, for the Series 3 icon. This is immediately followed by two bitmaps, each 48 pixels wide by 48 pixels deep, for the Series 3a/Workabout icon. This is discussed more fully in the *Series 3 Programming Overview* chapter in the *Series 3/3a Programming Guide*.

In order to ensure that the appropriate icon is used, the function `hCrackCommandLine` must be called *before* the call to `uCommonInit`. This must be done even though the application might have no interest in the contents of the command line.

Thus, the general shape of the `main()` of an application might look like this:

```
GLDEF_C VOID main(VOID)
{
    _UseFullScreen = TRUE;
    hCrackCommandLine();
    uCommonInit();
    SpecificInit();
    MainLoop();
}
```

This ensures that all the relevant Epoc statics are correctly initialised by the time the application connects to the window server (ie inside `uCommonInit`); this is when the window server decides which icon to use for the application.

If an application is also interested in the value returned by `hCrackCommandLine`, then it should save the returned value at this point. It must NOT call `hCrackCommandLine` a second time.

Storing data to file

Applications which manipulate significant quantities of data will in general wish to allow users to save this data to file. Such programs need to address the following points:

- the user interface allowing the user to specify which file(s) to open, save, merge, ...
- the format of the data, as saved on file
- the mechanism of reading/writing the data to and from file
- possible special requirements of writing files in a manner that makes best use of the Flash storage medium
- keeping the System Screen and the status window informed as to which file is currently being used.

The final topic is discussed in the section following this one. On the question of user interface, Hwif dialogs allow the inclusion of either of two types of filename specifier:

- *filename selectors* constrain the user to select a file that already exists - as is appropriate for commands such as *Open* and *Merge*
- *filename editors* allow the user to type in the name of a file that may or may not already exist - as is appropriate for commands such as *Save as* and *New*.

In either case, users can bring up the full filelist, simply by pressing TAB. Or they can press CONTROL+TAB for ease of swifter navigation to more distant files. Again, in either case, the dialog supplies an associated *disk selector*, without the programmer having to explicitly arrange for this. If the user has enabled *Remote Link*, drives on REM:: automatically become available for selection, alongside the local ones. All this happens just by virtue of a filename selector or editor being added to a dialog, being taken care of by ROM resident code on behalf of the Hwif programmer.

Given also that there is a wide range of flags allowing further customisation of the exact behaviour of these dialog items, Hwif programmers should find all their needs amply catered for, as regards the user interface of choosing files.

The Dbf file format

The Plib file i/o functions can be used for any variety of data formats on file, and Hwif programmers can choose whatever they feel most comfortable with. (There is some special treatment for text files.)

However, much can be said in favour of the Dbf file format that is used by, among other applications, the built-in *Database* and *Agenda*:

- It is designed with Flash-friendliness as a high priority, with incremental file modification as individual records are updated
- rom-resident code provides a rich set of services to simplify access to files of this format
- services such as random and sequential access are both highly optimised
- other services such as merging and compressing databases are easy to use.

The Hwif library itself has very little to add to these Dbf services (there is a utility function, `hIsDbfCompressible`, to determine whether a given Dbf channel supports file compression). More important is the fact that the Hwif example applications illustrate clearly the use of these Dbf functions.

Dialling telephone numbers

Typical contents of databases include telephone numbers.

The Hwif library includes a function, `hDTMFString`, to emit DTMF tones corresponding to a passed string.

This function uses the tone lengths and pauses as specified by the user in the *World* application (or otherwise), and reverts to system defaults in the absence of any such setting.

Communication with the System Screen

An important aspect of the Series 3 is the way all the built-in applications communicate with the System Screen:

- This name of any file currently open is displayed in bold in the file list in the System Screen
- This name is also displayed in any status window shown
- On a request from the System Screen, an application can close itself down tidily, saving any changes to file as appropriate
- Alternatively, applications can be requested to *switch files*, to change which file they are currently editing.

Applications use two mechanisms to communicate to the Series 3 OS their preferences concerning file switching, as well as the name of the file they are currently editing:

- some data is written *at compile time* into a *shell data (shd) file* that is linked into the application's *.app* file; this data includes the expected extension of any files to be edited, and the default top-level directory, as well as the more basic point of whether the application is file-based at all
- other data can be written *at run time* to various *reserved Epoc statics*; these include the full path name of the file currently being edited.

There are routines in the Hwif library to take care of keeping the various Epoc statics up to date. However, applications programmers may need to know about two of these statics directly:

UWORD <code>DatLocked</code>	this should be set to TRUE whenever, over a potentially extended period, the application is unable to respond to a <i>Switchfile</i> or <i>Shutdown</i> message from the System Screen
TEXT <code>*DatUsedPathNamePtr</code>	this points to a buffer giving the full path name of the file currently being edited.

For full details on the interaction between Series 3 applications and the System Screen, see the chapter *Communicating with the System Screen* in the *Series 3/3a Programming Guide*.

Reading the command line

In addition to being able to respond to requests of the *Switchfile* or *Shutdown* varieties, file-based applications on the Series 3 should also be able to read the command line they are sent when they start. This has a special form which can, however, be interpreted by means of the Hwif call `hCrackCommandLine`.

File-based applications would ordinarily include a call to `hCrackCommandLine` as part of their initialisation. One of the consequences of this call is that the Epoc static `DatUsedPathNamePtr` is initially pointed to an appropriate zero-terminated string in the body of the command line.

Storing the name of the file currently open

Initially, the name of the open file is part of the command line. However, when this has to be changed - either as a result of an *Open* or *Save as* command inside the application, or in response to a *Switchfile* request from the System Screen - a new buffer has to be used for this purpose. (The command line buffer is sized to precisely the right length needed for the initial file.)

Typically, file-based applications will maintain a permanent buffer, of length `P_FNAME_SIZE`, to store any change in the name of the file open. Once the new name has been copied into this buffer, the call `hSetUpStatusNames` should be made, to adjust all Epoc statics as appropriate, including `DatUsedPathNamePtr`.

The protocol of messages from the System Screen

A keycode with value equal to `CONS_EVENT_COMMAND` means that the System Screen wishes to communicate with the application. In case there is any doubt as to what the message is, it can be determined by making a call to `wGetCommand`.

For more details, see *Communicating with the System Screen* in the *Series 3/3a Programming Guide*.

Some notes on run-time errors

Errors arising from Hwif calls include the following types:

- programmer errors, such as making calls with unsuitable parameters (or disregarding earlier errors) - these may well result in the application being panicked
- as a special case of programmer error, menus or dialogs may turn out too wide to display properly on the screen; this is generally indicated by a return value `E_GEN_TOO_WIDE` from a call such as `uRunDialog` (there is also an associated error `E_GEN_TOO_MANY`) and the user will see a *Too wide* error alert
- shortage of memory in the application data space, generally indicated by a return value `E_GEN_NO_MEMORY`
- shortage of memory in the Window Server data space - also indicated by the same return value.

A well-written application needs to be able to recover from an out-of-memory (OOM) error, without falling over in the process or corrupting or losing any data. In general, an application should always test the return value of Hwif calls, to see whether they have succeeded, or whether they have failed with OOM.

However, there are some cases when an application can legitimately assume that a call always succeeds:

- if the call is part of the initialisation of the application
- *and* if the application has specified its start-up heap appropriately (this is done as a line in the *.pr* project file that orchestrates linking)
- *and* if the call only uses resources in the data space of the application (as opposed to resources in the data space of the Window Server).

Note that in no case can an application legitimately assume the success of a call which requires Window Server resources.

Application writers can make use of the *Spy* application to discover how much heap an application requires in order to start.

Some errors to consider

Other errors which application writers may need to consider include:

- running out of SSD space when writing to a file
- not being able to find a specified file (because the relevant SSD has been removed)
- the SSD being removed part way through a file operation
- (perhaps the least obvious) the failure of a remote link connection to another filing system - say because the user has shut the connection down since opening a file on that filing system.

Strategies on handling errors

Any error during application initialisation is generally fatal. The user should be informed of what has happened and the application terminated.

For example:

```
LOCAL_C VOID SpecificInit(VOID)
{
    INT command;

    MainWid=uFindMainWid();
    CreateGC();
    command=hCrackCommandLine();
    if (ObeySystemCommand(command,DatUsedPathNamePtr,&dH))
        p_exit(0);
    ReduceScreenSize();
    gBorder(W_BORD_CORNER_4);
    wsEnable();
    DisplayStart();
}
```

In the above example, notifying the user of the error takes place inside the routine `ObeySystemCommand`. In other cases, an application may wish to take advantage of the fact that if it calls `p_exit` with a negative parameter, the OS will automatically present a notifier on its behalf. The text in the notifier is the system error message corresponding to the parameter to `p_exit`.

Another straightforward case to handle is an error, such as OOM, while building up a dialog. In fact, all that needs to be done in this case is to follow the procedure given in several of the earlier examples: test the results of calls such as `uOpenDialog` and `uAddDialogItem`, and simply break out of the general stream of program flow when an error is detected. The user will see the error (presented by `Hwif` library code), can opt to free some memory by shutting other applications down, and then retry the command by invoking the same menu choice as before.

A similar approach can often be taken for cases such as errors when writing to an SSD. Alternatively, applications may wish, in these cases, to provide their own retry loop, and may even wish to insist that users successfully conclude the loop before allowing them to continue. This is appropriate for file-based applications in which the file must always be kept up to date.

Sometimes, indeed, all that an application can do, on detecting an error, is to notify the user accordingly, and then terminate the application. For example, if a user removes an SSD containing an open file, and an application tries to read data from this file, the following sequence of events will occur:

- the OS will present its own notifier, requesting that the SSD be reinserted
- this notifier has two options: Retry and Fail
- if the user selects Fail, the OS returns the error `E_FILE_ABORT` to the application.

In such a case, there is little an application can do, apart from terminating gracefully.

Reverting to the previous file

Ideally, a file-based application should aim, where possible, at being able to recover from failing to switch files (in response either to a menu command, or to a request from the System Screen) by means of reverting to the previously open file.

Thus suppose an application currently has file `name1` open, on file channel `fcbl`, and that the user requests that file `name2` be opened instead. Suppose further that, for one reason or another, `name2` cannot be loaded

into the application (it may be the wrong type of file, it may currently be locked by another application, or whatever). Then the user could see one of two things:

- an error notifier is presented, and then the application terminates
- an error notifier is presented, and then the application reverts to its previous state.

Evidently, the latter is preferable. It allows the user the opportunity to make due amends (for example, closing down another application which has the file already open) and then retry.

A simple approach to this end is to keep the first file open until the second file is successfully loaded. Only when this is complete are the resources associated with the first file freed.

For this reason, a file-based application will often possess a routine `ChangeOverTo` (with parameters such as the new filename and its new control block) that has the role of finally closing down the previous file, and then altering the application's records of the name of the current file. See the example given earlier, in the section *Typical dialog usage*.

Errors when formatting edit boxes

Single- and multi-line edit boxes pose an additional type of problem, in handling OOM errors.

In making a change to an edit box, OOM can occur in either of two ways:

- there is insufficient memory to increase the *contents* of the edit box (eg to add another character)
- the contents can be grown but the *layout* cannot be recalculated.

The point is that layout information (the location of all line breaks due to word-wrap, and so on) is dynamically allocated; consequently, the calculation of layout can fail.

Paradoxically, it turns out that the best reaction an application can make to the second kind of error is usually to ignore it. Hwif library code will ensure that the user is informed of the shortage of memory, and the edit box will only be partially redrawn. Despite only being able to redraw itself partially, the edit box will not crash, and will hold on to all its contents in the meanwhile. Once additional memory becomes available, the edit box will recalculate its layout, and then redraw itself correctly.

Future developments

In summary of the foregoing, it can be said that Hwif fulfils two separate roles with regard to applications programmers:

- in its own right, it supports the development of a large variety of significant and potent applications, with an agreeable user interface
- in a broader context, it serves as a critical stepping stone towards a full mastery of the Series 3 API, *including* object orientation, low-ram window redraws, and the `p_enter/ p_leave` mechanism that removes most of the clutter from error handling.

Whether programmers who learn Hwif will be content to stop there, and exploit the considerable avenues this opens up in its own right, or whether they will wish to press on in due course to master the full Series 3 API, will obviously vary from programmer to programmer. Both choices make good sense.

CHAPTER 2

WORKED EXAMPLES IN HWIF

How to use the supplied examples

The supplied example applications serve two purposes:

- *tutorial*, with embedded suggestions for further exploration
- *reference pool*, with varied illustrations of many parts of the Series 3 ROM software.

In neither case is there any need for would-be Hwif programmers to examine the code for *all* of the example applications provided with this manual; nor is there any need to digest all the associated discussion this chapter contains. Instead, the expectation is that prospective applications developers will work through items in the tutorial that they find of interest, and will merely skim through the other parts.

In this way, prospective applications developers will familiarise themselves with at least the *general contents* of the example applications. Then when they are planning their own application, they may well recall that one of the example applications does something similar (with a dialog, say) to something planned in their own application. In that case, the application developer can look up the relevant piece of source code, and consult the associated documentation, to discover how to produce the desired effect.

Strangely enough, it is often going to be *unhelpful* for Hwif application writers to think that a certain feature of the user interface of, for example, the built-in *Agenda* application ought to be duplicated in their own application. For even were the relevant source code available for perusal, that code would almost certainly be laden with proprietary object oriented techniques to the extent of being well out of the grasp of the Hwif application writer. On the other hand, familiarity with the supplied Hwif example applications is likely to provide a much more appropriate set of models to follow. Quite probably, there will be something in one of these applications that does *essentially* the same job as in the desirable feature of the built-in application (albeit possibly not so elegantly). In contrast to the code of the built-in application, the code of the example Hwif application *is* suitable for being copied into the developer's own application.

The embedded suggestions

In all forms of learning, practice makes perfect. This is as true for programming in a new system (such as Hwif), as it is for learning in general.

Accordingly, this chapter is regularly punctuated with "Suggestions" sections. These have been provided so that the would-be Hwif programmer who feels a bit unsure about some topic can have plenty scope for practising in that area. Trying out a few of the suggestions should increase understanding and boost confidence. Additionally, many of the suggestions provide a foretaste of topics to be discussed shortly afterwards. Others give hints on ideas that Hwif programmers might like to develop independently.

Of course, readers are free to think up their own ideas for how to modify the various examples presented. However, especially in the earlier stages, ideas that *seem* perfectly straightforward extensions of the examples given might turn out to be significantly more involved than at first thought. Careful thought has been applied to the lists of suggestions given in the text, to avoid precisely this problem.

Preview of the example applications

The first example application, *Query*, focuses almost exclusively on the basic menu and dialog functionality of Hwif (use of the Hwif time-text utility functions is also illustrated). Examples are given of each of the possible items that can be included in dialogs. There are only a few graphics calls, and

these are completely straightforward. There is no file-handling (likewise in fact for all of the first four example applications). The application itself provides answers to questions that a user may wish to pose, such as the conversion of centigrade values into Fahrenheit or miles into kilometres, the next occurrence of a certain date combination (eg Friday the 13th), the size of a specified file, and the encryption and decryption (using a supplied key) of given text messages.

The next example, *Tables*, introduces the important idea of reading a key asynchronously. This is because it involves the user typing in the answer to a multiplication problem (such as "4 times 9") before a timer that is ticking away on screen runs out completely. The application also introduces use of a simple Hwif edit box (with double height characters), and includes examples of several useful graphics techniques - with an animated action button and a growing gauge display *outside of the context of a dialog*. On exit, the state of the application is recorded in an environment variable, which is used to re-initialise the application the next time it is run.

The application *Remind* functions as a sort of half-way house between the built-in *Agenda* application and the built-in *Time* application. It allows users to set alarms with text strings, which on expiry can, if desired, be "snoozed" by specified time intervals (but still allowing access to the remainder of the Series 3 in the meantime). Evidently, this application illustrates access to the alarm server device. As such, it demonstrates more of the important concepts concerning asynchronous i/o. *Remind* also introduces another large subject: printing and access to the Print Setup dialog. The main screen display is a scrolling list of all reminders scheduled by the user, automatically sorted into chronological order. The graphics calls employed demonstrate how to achieve smooth scrolling (without undue screen flicker). Finally, on receipt of the keypress CONTROL+MENU, *Remind* hides or shows a permanent status window, and adjusts the rest of its display accordingly.

Notes is another application that functions as a half-way house between the functionality of two of the built-in applications: it straddles some of the characteristics of the built-in *Database* and *Word* applications. The screen is divided up into three edit windows - Title, Notes, and Find - with the data for a series of notes being directly edited in place (as opposed to being manipulated only through dialogs). As well as providing a wide-ranging survey of the editing facilities available via Hwif, the *Notes* application also gives a further example of printing.

The next example, *Dump*, produces a Hex dump of a nominated file. This is displayed on the screen in the first instance, but a selected portion of the dump can be written out to a nominated file. The dump can be scrolled in any direction, and it is possible to search it, either for strings of text, or for byte streams. The application introduces file-handling, both of the file to be dumped (initially chosen from the System Screen), and of the file to receive a written record of the dump. Finally, the use of a special mode is illustrated, in which repeated cursor keypresses may cause the display to be drawn in its new position only when there is a suitable delay in receiving keys.

Then comes a couple of applications illustrating different uses of the Dbf database subsystem. These applications differ from both *Notes* and *Remind* in that they make permanent copies of their data on file (whereas *Notes* and *Remind* just operate with in-memory data). Both of these applications deal with *Shutdown* and *Switchfile* messages from the System Screen.

The first of these Dbf applications, *Tele*, is a simple example of a fixed field database, with fields for a name, a department code, and a telephone extension number. Records can be added, deleted, updated, searched for, and even sorted (using quicksort). A couple of other Dbf file options are also illustrated: Merge and Copy. Telephone numbers, once found, can have corresponding DTMF tones emitted. The main screen display is straightforward, involving double height characters.

The application *Days* illustrates a Dbf database that stores peoples' birthdays (or other days of interest), along with a notes field. The application maintains an in-memory index which sorts the entries by date to facilitate a more meaningful presentation of the contents of the database. The main screen view is somewhat elaborate: it can be toggled between a series of edit boxes (as in *Notes*) and a scrolling list (as in *Remind*). Alteration of the data takes place by direct manipulation in the edit boxes.

Many secrets of the inner workings of the Series 3 are revealed by the application *Spy*, which presents a list of all the processes running at any one time, together with specified information, such as the number of cells in the allocator heap of that process, and the watermark on its stack. This list can be refreshed on a timer, so the application also provides an additional example of asynchronous keyboard reads. For more details on what can be done using *Spy*, see the chapter *Using Spy.app* in the *Series 3/3a Programming Guide*.

Associated with *Spy* is a maverick application called *Joker*, whose main role is to instantiate all the special cases tested for by *Spy*. For example, *Joker* can corrupt its allocator heap on demand, in a variety of different ways.

Finally, the application *Iconed* is a full-blown icon editor, which can be used to design icons for new applications (and to improve the icons shipped with the example applications). This illustrates a whole variety of more advanced graphics calls, as well as another set of possibilities in file-based applications.

Getting started: the Query application

Hello World

Consider the following program:

```
#include <p_std.h>
#include <wlib.h>
#include <hwif.h>

GLDEF_C VOID main(VOID)
{
    WMSG_KEY key;

    uCommonInit();
    wSetBusyMsg("Hello world",W_CORNER_BOTTOM_LEFT);
    uGetKey(&key);
    uGetKey(&key);
    p_exit(0);
}
```

This can be typed into your favourite programming editor (either inside or outside the TopSpeed programming environment invoked by the *ts* command). Or, to save time, it can be loaded from disc: this program is supplied in the directory `\sibosdk\hwdemo\` as *qu1.c* (with the name of this file indicating that it is the *first* version of the *Query* application).

The meaning of various lines in this program is as follows:

<code>#include <p_std.h></code>	This is the standard Psion header file, containing the function prototypes for all "simple" Plib routines (such as <code>p_exit</code>), as well as the typedefs for the likes of <code>GLDEF_C</code> and <code>VOID</code>
<code>#include <wlib.h></code>	This is the Wlib header file, containing the function prototypes for all Wlib routines, such as <code>wSetBusyMsg</code> , as well as the definitions of constants like <code>W_CORNER_BOTTOM_LEFT</code> and structs like <code>WMSG_KEY</code>
<code>#include <hwif.h></code>	This is the Hwif header file, containing the function prototypes for all Hwif routines, such as <code>uCommonInit</code> , as well as the definitions of numerous constants and structs
<code>uCommonInit();</code>	The first statement in all Hwif programs; miss this out and the above program will panic when it reaches the next line
<code>wSetBusyMsg(...);</code>	Display the indicated text as a flashing message at the bottom left corner of the screen
<code>uGetKey(&key);</code>	Wait for a keypress event to be delivered (see below on why this line appears <i>twice</i> in the above program)
<code>p_exit(0);</code>	Exit the application cleanly.

To build the program

The next stage is to convert the above *.c* source file into a *.img* executable file. Any such conversion is governed by a TopSpeed *project* file, with extension *.pr*. (The *.pr* file is involved whether or not compilation takes place inside the *ts* system.)

In fact, all but two of the Hwif demo applications can use the project file *unnamed.pr* that can be found in the same directory as the source modules. The exceptions are:

- *Spy* which has three different source modules and therefore its own *.pr* file (*spy.pr*)
- *Remind* which has its own *remind.pr*

For more details on *.pr* files, and on the associated housekeeping batch files such as *make.bat*, see the chapter *Building an Application* in the *General Programming Manual*.

To create *qu1.img*, simply type `make qu1`.

Note that the *unnamed.pr* in `\sibosdk\hwdemo` has one extra line than the copy in `\sibosdk\demo`:

```
#pragma link(hwif.lib)
```

to ensure that the Hwif library is pulled into the link.

Errors during linking

However the *.img* file is built, a spurious pair of warnings may, regrettably, be issued by the linker:

```
_ClassTable is duplicated, files involved are hwif and rlib
_ExtCatTable is duplicated, files involved are hwif and rlib
```

These warnings (which, at the time of writing, cannot be disabled) should be ignored. It is, however, important that the HWIF and RLIB libraries are linked in the correct order. The RLIB library is automatically included in the link and should not be explicitly included in any application's *.pr* file.

Needless to say, any other reported errors should be carefully attended to.

Running the application from the Series 3 System Screen

Once the file *qu1.img* has successfully been built, it can be copied (using Remote Link on the Series 3 and McLink on the PC) into a top-level `\img\` directory on a Series 3. The name *Qu1* will now appear under the *RunImg* icon in the System Screen. (Update the System Screen display, if need be, by pressing SYSTEM; you can press CONTROL+SYSTEM to position to the *RunImg* icon; if perchance you have removed this icon, re-install it with the *Install standard* menu command.)

Cursor down onto the name *Qu1* and press ENTER. The screen will clear completely, except for a message *Hello world* flashing at the bottom left hand corner. On any keypress, the application terminates.

For an explanation of how to *debug* an application such as *Qu1*, see in the first instance the chapter *Building an Application* in the *General Programming Manual*, and for more details, the *SIBO Debugger* manual.

Further explanation of qu1.c

The reason why *qu1.c* contains two calls to `uGetKey` can now be clarified. Whenever any application comes into the foreground, it is sent notification of this fact. For Hwif programs, this notification takes the form of a special keypress. The value of this keypress is `CONS_EVENT_FOREGROUND`, that is 0x401 (consult the header file *p_cons.h* - which is automatically `#included` by *hwif.h*.) This (generalised) keypress is sent to the application as soon as it starts, since when it starts, it comes into foreground. Hence the need for a *second* call to `uGetKey`, to make the program wait until another keypress is received, before terminating.

Incidentally, the code

```
GLDEF_C VOID main(VOID)
{
    ...
    p_exit(value);
}
```

is of course equivalent to

```
GLDEF_C INT main(VOID)
{
    ...
    return(value);
}
```

However, *qu1.c* uses the former method since it is actually rare for Hwif applications to terminate at the bottom of their main routines. Instead, they usually terminate, with a call to `p_exit`, as soon as a suitable menu command is received (but after first carrying out any necessary checks and/or saving data to file); in-line calls to `p_exit` are also common when errors arise during application initialisation.

Suggestions for modifying Qu1

- Replace the two calls to `uGetKey` with a loop that repeatedly waits for keypresses, and which terminates the program only on a designated keypress
- Inside this loop, test for other chosen keypresses, and make calls to `wSetBusyMsg` with parameters that depend on what the keypress is

- Include, on various keypresses, calls to `wInfoMsg` as well as to `wSetBusyMsg`; also try calls to `p_sound`, `wsAlertW`, and `wClientPosition` (to position the application to background)
- Instead of displaying *Hello world*, display the current time and/or date
- Deliberately call `p_exit` with a negative parameter, to see what the effect is.

Introducing a GC

There are many graphics calls in the Wlib library that cannot be made unless a GC (*graphics context*) exists. This includes the function `gBorder` that provides the standard Series 3 framing for regions of the screen.

Accordingly, one step up from *qu1.c* is for `main` to become (as in *qu2.c*)

```
GLDEF_C VOID main(VOID)
{
    WMSG_KEY key;

    uCommonInit();
    CreateGC();
    gBorder(W_BORD_CORNER_4);
    wSetBusyMsg("Hello world", W_CORNER_BOTTOM_LEFT);
    FOREVER
    {
        uGetKey(&key);
        if (key.keycode==(W_SPECIAL_KEY|'x'))
            p_exit(0);
    }
}
```

with the application-supplied routine `CreateGC` being

```
LOCAL_C VOID CreateGC(VOID)
{
    INT hgc;

    hgc=gCreateGC0(uFindMainWid());
    if (hgc<0)
        p_exit(hgc);
}
```

The GC created has default characteristics (hence the call `gCreateGC0` as opposed to the more general `gCreateGC`, which takes more parameters). The one parameter that has to be passed to `gCreateGC0` is the ID of the window the GC is attached to. The Hwif call `uFindMainWid` returns, as its name implies, the ID of the window filling the screen, that was created by the call to `uCommonInit`.

Note that the call `gCreateGC0` can fail. This is because it requires the Window Server to allocate additional resources, and it may be the case, if there are many other applications running on the Series 3, some of which have many windows, that the Window Server cannot satisfy this request.

Suggestions for modifying Qu2

- Now that a GC has been created, try out the effect of other Wlib calls, such as `gDrawLine`, `gDrawBox`, `gBorderRect`, `gClrRect`, `gPrintText`, `gPrintBoxText`, and `gInvObloid` - all (possibly) in response to the receipt of suitable incoming keypresses
- Use `gCreateGC` instead of `gCreateGC0`, and experiment with the other parameters (eg the font and style fields of the `G_GC` struct required)
- Use `gsetGC` to change the nature of the GC dynamically on designated keypresses (there will need to be a permanent record of the handle `hgc` of the GC created)
- Experiment with graphics calls that, being non GC-based, require to be passed the ID of the relevant window: `wScrollWin`, `wDrawTextCursor`, `wMakeInvisible`, and `wsCreateClock`.

A status window and a menu bar

To look more like a genuine Series 3 application, the display should, where possible, sport a status window. This requires two steps:

- calling the Wlib function `wsEnable` to display a permanent status window

- resizing the main graphics window, so that it no longer obscures the region where the status window is displayed.

Provided the call to `gBorder` is delayed until after the screen is resized smaller, the border will automatically appear with an appropriately reduced width.

Note that the call `wsEnable` has no visible effect unless the main window of the application has been resized appropriately: permanent status windows always come *at the back* of the window order.

The routine to resize the main graphics window smaller is as follows:

```
LOCAL_C VOID ReduceScreenSize(VOID)
{
    W_WINDATA wd;

    wd.extent.tl.x=wd.extent.tl.y=0;
    wd.extent.width=189;
    wd.extent.height=80;
    wSetWindow(MainWid,W_WIN_EXTENT,&wd);
    if (uErrorValue(wCheckPoint()))
        p_exit(0);
}
```

Note the following points:

- the static variable `MainWid` stores the result of a prior call to `uFindMainWid`
- unexpectedly, it is possible for the resize to fail on account of lack of Window Server memory; this is because the backed-up bitmap for the reduced window size is created before the one for the old window size is finally discarded
- the call to `wSetWindow` is not flushed straightaway because it does not return a value; in other words, the call may not be executed immediately. However, as the above point notes, it is possible that the call could fail when it does eventually execute. To make sure that the call is executed immediately, it is necessary to flush the Window Server command buffer immediately after the call is made and to check that no error is reported as a result; this is the point of the otherwise little-used call `wCheckPoint`.
- The width and height values are suitable for the Series 3 screen, or for the Series 3a or *Workabout* in Series 3 emulation mode. For a Series 3a in native mode, or a *Workabout* in full-screen emulation mode, alternative values may be more appropriate.

The lines of code

```
if (uErrorValue(wCheckPoint()))
    p_exit(0);
```

are equivalent to

```
INT ret;

ret=wCheckPoint();
if (ret)
    p_exit(ret);
```

Both methods result in the user being notified of any error (with an appropriate error string being presented), and in the application being terminated in response to the error.

The routine `ReduceScreenSize` is added into the developing *Query* application in the file *qu3.c*. *Qu3* also adds in the presentation of a menu bar. There is now enough initialisation to create a separate initialisation routine

```
LOCAL_C VOID SpecificInit(VOID)
{
    MainWid=uFindMainWid();
    ReduceScreenSize();
    CreateGC();
    gBorder(W_BORD_CORNER_4);
    wsEnable();
    wInfoMsg("Hello world");
}
```

and main now simplifies to what is, in Hwif applications, its standard form:

```
GLDEF_C VOID main(VOID)
{
    uCommonInit();
    SpecificInit();
    MainLoop();
}
```

This leaves the new routine MainLoop:

```
LOCAL_C VOID MainLoop(VOID)
{
    INT ret;
    WMSG_KEY key;

    FOREVER
    {
        uGetKey(&key);
        if (key.keycode&W_SPECIAL_KEY)
            ManageCommand(key.keycode&(~W_SPECIAL_KEY));
        else if (key.keycode==W_KEY_MENU && !(key.modifiers&W_CTRL_MODIFIER))
        {
            ret=uPresentMenus();
            if (ret>0)
                ManageCommand(ret);
        }
    }
}
```

which is an elaboration of the lines

```
FOREVER
{
    uGetKey(&key);
    if (key.keycode==(W_SPECIAL_KEY|'x'))
        p_exit(0);
}
```

from *qu2.c*.

Note the following points:

- the above code ignores the key combination CONTROL+MENU which often toggles the visibility of any permanent status window; for this application, because the main display area (apart from the status window) is so empty, there is no special merit in allowing the permanent status window to be hidden
- only positive return values from uPresentMenus are of interest; other values correspond to the user cancelling out of the menu bar by pressing ESCAPE (as well as to cases where the menu presentation failed due to lack of memory)
- the program has to provide two passages to the routine ManageCommand: one following the presentation of the menu bar, and the other following interception of a menu accelerator when no menu is showing.

It is worth mentioning a potential problem with the MainLoop routine. On the Series 3 only the lower case characters 'a' to 'z' plus the four characters '+', '-', '*' and '/' (in English language versions) are valid accelerators and the above code will work unambiguously. The Series 3a and *Workabout*, however, permit the uppercase alphabetic characters 'A' to 'Z'. To cater for this situation, MainLoop could be changed as shown below:

```
LOCAL_C VOID MainLoop(VOID)
{
    INT code;
    INT ret;
    WMSG_KEY key;
```

```

FOREVER
{
    uGetKey(&key);
    if (key.keycode & W_SPECIAL_KEY)
    {
        code = key.keycode & (~W_SPECIAL_KEY);
        if (key.modifiers & W_SHIFT_MODIFIER)
            code = p_toupper(code);
        ManageCommand(code);
    }
    else if (key.keycode==W_KEY_MENU && !(key.modifiers&W_CTRL_MODIFIER))
    {
        ret=uPresentMenus();
        if (ret>0)
            ManageCommand(ret);
    }
}
}

```

The function `p_toupper` is needed to ensure that the keycode is in uppercase before being passed to the `ManageCommand` routine. Note, however, that no change is required to the code concerned with selecting a command by highlighting its menu item and pressing ENTER (implemented by the call to `uPresentMenus`).

Defining the menu bar

The top of *qu3.c* is as follows:

```

LOCAL_D TEXT *cmds[] =
{
    "iInches/Centimetres",
    "mMiles/Kilometres",
    "lPounds/Kilogrammes",
    "pPints/Litres",
    "fFahrenheit/Centigrade",
    "dDay of week",
    "cCombinations",
    "tTime difference",
    "hHoroscope",
    "nNow",
    "wPassword",
    "eEncrypt",
    "uDecrypt",
    "sSignificance",
    "zFile size",
    "xExit",
    NULL
};

LOCAL_D H_MENU_DATA mdata[] =
{
    "Conversions",5,      /* first five of above commands form the Conversions menu */
    "Calendar",5,        /* next five form the Calendar menu */
    "Secret",3,          /* then three for the Secret menu */
    "Special",3,         /* then three for the Special menu */
    NULL
};

GLDEF_D TEXT **_cmds=(&cmds[0]);
GLDEF_D H_MENU_DATA *_mdata=(&mdata[0]);

```

with the statics `_cmds` and `_mdata` defining the contents of the menu bar and of each pulldown menu.

The contents of `ManageCommand`

Evidently, the real core of the application is to be found in `ManageCommand`, and in routines called therein.

As far as *qu3.c* is concerned, the contents of `ManageCommand` are just as follows:

```
LOCAL_C VOID ManageCommand(INT keycode)
{
    INT index;
    TEXT buf[40];

    switch (keycode)
    {
        case 'x':
            p_exit(0);
        default:
            index=uLocateCommand(keycode);
            if (index>=0)
            {
                p_atos(&buf[0],"You chose '%s'",cmds[index]+1);
                wInfoMsg(&buf[0]);
            }
    }
}
```

As can be seen, only the Exit command functions properly. All other commands give rise to an information message of the form

You chose 'Fahrenheit/Centigrade'.

Note the use of the Hwif library routine `uLocateCommand` to convert between the accelerator of a menu command (which is what is returned by `uPresentMenus`) and the index of the menu command in the table identified by `_cmds`. This provides a simple means of recovering the text for the chosen menu command. Note also the need to test whether the passed keycode corresponds to *any* of the menu commands (ie the test on whether the return value from `uLocateCommand` is non-negative).

Suggestions for modifying Qu3

- Provide code to display the current day name, in response to the *Day of week* menu command (use eg `wInfoMsg`)
- Display the current time, in response to the *Now* menu command
- In response to the *File size* menu command, display the size of *qu3.img*, determined by a run-time call (the full path name of *qu3.img* will be stored, as a zero terminated string, at the Epoc static `DatCommandPtr` - as can be verified inside the Debugger)
- Provide code to toggle the permanent status window on receipt of `CONTROL+MENU`.

Supplying an icon

Qu3 suffers from having an empty hole, in its status window, where an icon should be. (The "empty hole" is actually the default icon.) This shortcoming is in fact shared by *Qu1* and *Qu2*, in that any temporary status window displayed while they are in foreground also lacks a proper icon. The problem is solved by *Qu4*. The critical difference is that *Qu4* has its own *.afl* file.

The content of *qu4.afl* is just the single line

```
query.pic
```

When *Qu4* is being linked, the fact that there is an *.afl* file is picked up by the TopSpeed make system (in the part specially customised for the Epoc system), and any files listed in this file are joined together, with the ordinary outcome of linking, to produce the final *.img* file. In this case, a copy of the icon file *query.pic* is built into the final executable *qu4.img*.

First examples in presenting dialogs

Qu4 advances from *Qu3*, not only by having a proper icon, but also by having some proper dialogs - one each for the commands *File size*, *Day of week*, and *Now*.

ManageCommand accordingly grows:

```
LOCAL_C VOID ManageCommand(INT keycode)
{
    INT index;
    TEXT buf[40];

    switch (keycode)
    {
        case 'd':
            DayOfWeek();
            break;
        case 'n':
            TimeNow();
            break;
        case 'z':
            FileSize();
            break;
        case 'x':
            p_exit(0);
        default:
            index=uLocateCommand(keycode);
            if (index>=0)
            {
                p_atos(&buf[0],"You chose `%s'",cmds[index]+1);
                wInfoMsg(&buf[0]);
            }
    }
}
```

and the code for FileSize is

```
LOCAL_C VOID FileSize(VOID)
{
    TEXT fname[P_FNAME_SIZE+2];
    H_DI_FSEL fsel;
    P_INFO info;
    TEXT buf[30];

    fname[0]=0;
    fsel.fname=&fname[0];
    fsel.flags=H_FILE_PICK_SELECTOR;
    FOREVER
    {
        if (uOpenDialog("Find file size"))
            return;
        if (uAddDialogItem(H_DIALOG_FSEL,"File:",&fsel))
            return;
        if (uRunDialog()<=0)
            return;
        fname[1+fname[0]]=0;
        if (!uErrorValue(p_finfo(&fname[1],&info)))
        {
            p_atos(&buf[0],"File size is %lu bytes",info.size);
            wInfoMsg(&buf[0]);
        }
    }
}
```

The call that works out, amongst other things, the size of the specified file, is

```
p_finfo(&fname[1],&info);
```

with the size, in bytes, being written to the size member of the passed P_INFO struct. The code that *displays* the file size is

```
p_atos(&buf[0],"File size is %lu bytes",info.size);
wInfoMsg(&buf[0]);
```

After the user completes the dialog, the file size is displayed, and the dialog is presented again, for the user to choose another file. The `FOREVER` loop in `FileSize` only terminates

- if the user presses `ESCAPE` to cancel the dialog - in which case `uRunDialog` returns zero
- if there is insufficient memory for any of the calls defining or presenting the dialog - in which case the corresponding call to `uOpenDialog`, `uAddDialogItem`, or `uRunDialog` returns a negative number.

The filename chosen by the user is written to the buffer `fsel.fname` as a leading byte counted string (BCS). However, the function `p_finfo` requires a zero terminated string (ZTS). Hence the conversion

```
fname[1+fname[0]]=0;
```

before the call to `p_finfo`. Since a filename as written by a filename selector can have, in general, up to `P_FNAME_SIZE` (128) bytes, and since the above code writes an extra byte beyond the end of this, the result is that `fname` has to be declared to be at least `P_FNAME_SIZE+1` bytes long. In the interests of even byte alignment, it has actually been declared as `P_FNAME_SIZE+2` bytes long.

It is necessary to add the line

```
#include <p_file.h>
```

to the top of `qu4.c`, since this is where the definitions of the constant `P_FNAME_SIZE` and the struct `P_INFO` are to be found. This header file also contains the function prototype for `p_finfo`.

The call `uErrorValue` made around the result of `p_finfo` presents a suitable error message, if the size of the file cannot be found. There is no need for corresponding calls around the results of `uOpenDialog`, `uAddDialogItem`, and `uRunDialog`, since these latter functions have error-notification built into them. Error notification is standard for all but the most primitive of the Hwif library routines; however, because of the generality of the `Plib` and `Wlib` functions, such error-notification code is not supplied in their case.

In fact, it would be a very rare case indeed for the above call to `p_finfo` to fail. This is because the filename selector item in the dialog automatically validates its contents, before allowing the dialog to terminate. Nevertheless, it is theoretically possible for the file to be deleted *in between* the calls `uRunDialog` and `p_finfo` (bear in mind the multi-tasking nature of the Series 3; more likely, an SSD might be removed, or, for the case of a file selected on `REM::`, a remote link might become broken). Hence the call to `uErrorValue`.

In the above code, the filename is initialised to have zero length, by the code

```
fname[0]=0;
```

This means that the filename selector will position itself initially to the *default path* of the application. Since nothing has been done to set this up, the file shown in the dialog, when it first appears, will most likely be something in the root directory of the default drive - perhaps the file `sys$stub.img`. If there are *no* files in this directory, the filename selector will say so, and will not allow the user to terminate the dialog (apart from cancelling it) until transitioning to a directory in which files do exist.

Suggestions for modifying FileSize

- Display the time the file was last modified, instead of its size
- To see the effect of the `uErrorValue` call, introduce a `p_sleep` before the call to `p_finfo` and use this delay to pull out an SSD on which a filename has been selected.

A date editor in a dialog

The routine `DayOfWeek` contains an example of a date editor in a dialog:

```
LOCAL_C VOID DayOfWeek(VOID)
{
    P_DAYSEC ds;
    H_DI_DATE date;
    TEXT buf[40];
    H_DI_TEXT txt;
```

```

    SetUpDiDate(&date,&ds);
    do
    {
        p_nmday(&buf[1],p_wkday(ds.day));
        buf[0]=p_slen(&buf[1]);
        txt.str=&buf[0];
        txt.type=H_DTEXT_ALIGN_LEFT;
        if (uOpenDialog("Find day of week"))
            return;
        if (uAddDialogItem(H_DIALOG_DATE,"Date",&date))
            return;
        if (uAddDialogItem(H_DIALOG_TEXT,"Day of week",&txt))
            return;
    } while (uRunDialog()>0);
}

```

In contrast to `FileSize`, which displays its result as an information message separate from the dialog, (using the call `wInfoMsg`), `DayOfWeek` displays its result inside the dialog, as a text item included in the dialog:

```

    buf[0]=p_slen(&buf[1]);
    txt.str=&buf[0];
    txt.type=H_DTEXT_ALIGN_LEFT;
    ...
    if (uAddDialogItem(H_DIALOG_TEXT,"Day of week",&txt))
        return;

```

The name of the day in the week is determined, as a ZTS, by the calls

```
p_nmday(&buf[1],p_wkday(ds.day));
```

and the string is converted into the BCS form required by the `H_DI_TEXT` struct by the code

```
buf[0]=p_slen(&buf[1]);
```

The day itself is stored, in the `ULONG ds.day`, as a day number since the beginning of 1900. This is the form required by both the call `p_wkday` and the struct `H_DI_DATE`. It is initialised to *today* by the routine `SetUpDiDate`:

```

LOCAL_C VOID SetUpDiDate(H_DI_DATE *pdate,P_DAYSEC *pds)
{
    ULONG sdate;

    sdate=p_date();
    p_sttods(&sdate,pds);
    pdate->value=(&pds->day);
    pdate->low=0;
    pdate->high=H_LAST_DAY;
}

```

It is necessary to add the line

```
#include <p_date.h>
```

to the top of `qu4.c`, since this is where the definitions of the struct `P_DAYSEC` and the function `p_sttods` are to be found.

Choice lists and the time-text functions

The routine `TimeNow` contains examples of choice lists in a dialog - five choice lists in all, in fact all just with the two choices "No" and "Yes" and added into the current dialog by the utility function `AddNoYesChoiceList`:

```

LOCAL_C INT AddNoYesChoiceList(TEXT *pmt,UWORD *pval)
{
    return(uAddChoiceList(pmt,pval,"No","Yes",NULL));
}

```

```

LOCAL_C VOID TimeNow(VOID)
{
    INT values;
    H_DI_TEXT txt;
    TEXT buf[48];

    txt.str=&buf[0];
    do
    {
        if (uOpenDialog(NULL))
            return;
        uZTstoBCS(&buf[0],"Time is now");
        txt.type=H_DTEXT_ALIGN_CENTRE;
        if (uAddDialogItem(H_DIALOG_TEXT,NULL,&txt))
            return;
        values=0;
        if (ttMonth==2)
            values|=H_TIME_FORMAT_MONTH_NAME;
        if (ttDay==2)
            values|=H_TIME_FORMAT_DAY_NAME;
        if (ttSuffix==2)
            values|=H_TIME_FORMAT_SUFFIX;
        if (ttCentury==1)
            values|=H_TIME_FORMAT_NO_CENTURY;
        if (ttSeconds==1)
            values|=H_TIME_FORMAT_NO_SECONDS;
        hTTSetFormat(ttH,values);
        hTTSetTime(ttH,H_TIME_SET_NOW,NULL);
        buf[0]=hTTSenseString(ttH,H_TIME_SENSE_BOTH,&buf[1]);
        txt.type=H_DTEXT_ALIGN_CENTRE|H_DTEXT_UNDERLINE;
        if (uAddDialogItem(H_DIALOG_TEXT,NULL,&txt))
            return;
        if (AddNoYesChoiceList("Give month name",&ttMonth))
            return;
        if (AddNoYesChoiceList("Give day name",&ttDay))
            return;
        if (AddNoYesChoiceList("Use date suffix",&ttSuffix))
            return;
        if (AddNoYesChoiceList("Show century",&ttCentury))
            return;
        if (AddNoYesChoiceList("Show seconds",&ttSeconds))
            return;
    } while (uRunDialog(>0);
}

```

The live variables for the five choice lists are five statics defined at the top of *qu4.c*, and all given initial values reflecting the defaults built into the Hwif time-text utility functions:

```

LOCAL_D VOID *ttH;
LOCAL_D UWORD ttMonth=1;
LOCAL_D UWORD ttSuffix=1;
LOCAL_D UWORD ttDay=1;
LOCAL_D UWORD ttCentury=2;
LOCAL_D UWORD ttSeconds=2;

```

The textual form of the time is generated by the time-text channel *ttH* opened by the following call at the end of *SpecificInit*:

```
ttH=hTTOpen();
```

The text is actually generated by the calls

```

hTTSetFormat(ttH,values);
hTTSetTime(ttH,H_TIME_SET_NOW,NULL);
buf[0]=hTTSenseString(ttH,H_TIME_SENSE_BOTH,&buf[1]);

```

where *values* has been built up as a combination of the present values of the *ttxxx* variables.

Whereas the results of *FileSize* and *DayOfWeek* only persist until the user cancels the dialog, the result of *TimeNow* persists throughout the lifetime of the application. This is because the *ttxxx* variables are statics.

A note on the start-up heap

There is no special need to test for the result of the call `htOpen` made in `specificInit`. No Window Server resources are required for a time-text channel; the only allocating that needs to be performed for it is out of the application's own heap. Since the call is made during application initialisation, it can be assumed that, if the application has been allowed by the OS to run at all, there will be sufficient memory for the call to succeed.

This point is worthy of some further explanation. Using either the Debugger or the *Spy* application, it can be seen that, when running, *Qu4* has only around 0x300 bytes allocated from its heap. However, the default value for the minimum heap of an application is 0x80 paragraphs, ie 0x800 bytes.

Further investigation will reveal that the minimum *segment size* for *Qu4* is some 0x19c0 bytes, made up as follows:

0x1000	stack
0x800	minimum heap
0x1c0	static data

These figures may be confirmed by running the tool *edump* on *qu4.img*:

```
edump qu4
```

When the OS is instructed to try to run *Qu4*, it first has to allocate the data segment of 0x19c0 bytes. If it fails to do so, the application is not allowed to run, and an out-of-memory notifier is presented. But if it succeeds, the 0x19c0 bytes are guaranteed to remain available throughout the lifetime of the application. Hence the guarantee that the call to `htOpen` will never fail.

Clearly, *Qu4* is an extremely anti-social application, hogging much more heap (not to mention much more stack) than it needs. Such behaviour would be unacceptable in any commercial application. One penalty the application incurs, upon itself, is that the OS will sometimes refuse to run it, even though there is sufficient memory available for its *actual* requirements - the point being that there is insufficient memory available for its *stated* requirements.

Incidentally, the start-up heap for an application can be customised by means of including a line such as

```
set heapsize=0x40
```

in the *.pr* project file governing how the application is built. The stack can be specified by means of a different value of `epocinit`.

Further comments on edump

Another piece of information that *edump* gives is the size of any additional files built into the specified image. Thus the result of running *edump* on *qu4.img* includes the line

```
Add 1 offset,len = 0040 (bytes), 0074 (bytes)
```

whereas no such line is given for *qu3.img*. This additional file, of size 0x74 bytes, is of course the copy of the icon *query.pic*.

Suggestions for modifying Qu4

- Make the results of `FileSize` and `DayOfWeek` persistent in the same way as the result of `TimeNow` is
- For some dates (eg Wednesday 26th September), the textual representation generated in `TimeNow` can end up too wide to fit properly within the widest dialog that is allowed; look out for such cases and abbreviate the text suitably (use abbreviated versions of the day and/or month names)
- Produce a customised project file *qu4.pr* including a line defining the start-up heap more appropriately; confirm the result using *edump*.

From .img to .app

Although *Qu4* has an icon built into it, it is not yet able to be installed in its own right as an application in the System Screen. For this to be possible, an application also needs to have a *shd* (*shell data*) file built into it.

For *Query*, the source of the *shd* file is *query.ms*, which consists solely of the line

```
Query
```

This is actually an abbreviated form of a three-line file:

```
Query
0
```

in which the third line gives the *type* of the application. A type of zero means that the application is non file-based, and consequently has no associated files.

The file *query.shd* can be produced from *query.ms* by the command

```
makeshd query
```

and then the file *query.shd* is joined into the final executable by being listed in *query.afl*.

Up to three files can be specified in an *.afl* file. Whereas the icon of an application can be placed into any of the three slots, the shell data has to be placed into the third slot. Thus the contents of *query.afl* become

```
query.pic
query.rsc
query.shd
```

where *query.rsc* is any small file (preferably a zero-length file).

Running *edump* on *query.img* produces the following three lines of output (among others)

```
Add 1 offset,len = 0040 (bytes), 0074 (bytes)
Add 2 offset,len = 00C0 (bytes), 0000 (bytes)
Add 3 offset,len = 00C0 (bytes), 0024 (bytes)
```

Whereas an application without shell data is usually copied to an *\img* directory on the Series 3, one with shell data is usually copied to an *\app* directory, and renamed from *.img* to *.app* at the same time. Thereafter, the application can be installed, using the *Install application* command in the System Screen.

Once installed, it can be run in the same way as any of the built-in applications is. Further, an application button such as CONTROL+CALC can be assigned to it, if desired.

The floating point emulator sys\$8087.ldd

Before *Query* can be run successfully, the Series 3 needs to be able to locate the *floating point maths emulator*, *sys\$8087.ldd*. This is because *query.c* contains lines such as

```
DOUBLE fahr;

fahr=32;
```

which, innocent as it may seem, requires the presence of *sys\$8087.ldd*.

The simplest way to ensure the Series 3 can locate this emulator is to place a copy of it in the same directory as the application itself. Thus if *query.img* is copied to *m:\app\query.app* on the Series 3, a copy of *sys\$8087.ldd* could be copied into this same directory, *m:\app*. A copy may be found in *\sibosdk\lib* on the PC.

In fact, of the example applications, *Query* is the only one which requires the presence of the emulator.

The built-in applications avoid requiring to use the emulator, since they replace the likes of the above lines of code by the following

```
DOUBLE fahr;
WORD temp;

temp=32;
p_itof(&fahr,&temp);
```

which although it looks more cumbersome, actually produces leaner code overall.

Debugging a .app application

The mechanism for debugging a *.app* application is virtually the same as debugging a *.img* application. In neither case is there any need to copy the application onto the Series 3 by hand.

The only complication concerns the need to pass a suitable command line to *file-based* applications. This is considered later. However, non file-based applications, such as *Query*, can be run without any command line being passed to them.

Some responsibilities of being a .app

In general, an application intended to be capable of being installed in the System Screen should always make a call to `hCrackCommandLine` in its `SpecificInit` routine (or equivalent). This is true whether or not the application is file-based. If no call to `hCrackCommandLine` is made, the EPOC static `DatStatusNamePtr` will be left at its default value of zero, and it will, accordingly, be fruitless for a user to assign an application button (such as `CONTROL+CALC`) to this application.

However, any application that calls `hCrackCommandLine` must explicitly test for system messages of (at least) the *Shutdown* variety (assuming the application has not added in 4000 to its shell data type, to prevent such messages ever being sent). This means that the top of `MainLoop` in *query.c* has to have the form

```
LOCAL_C VOID MainLoop(VOID)
{
    WMSG_KEY key;

    FOREVER
    {
        uGetKey(&key);
        if (key.keycode & W_EVENT_KEY)
        {
            if (key.keycode == CONS_EVENT_COMMAND)
                p_exit(0);
        }
        else ...
    }
}
```

Menu command look up - by accelerator or by index?

Query differs from *Qu4* in the way the switch statement in `ManageCommand` is constructed: in place of

```
LOCAL_C VOID ManageCommand(INT keycode)
{
    switch (keycode)
    {
        ...
    case 'd':
        DayOfWeek();
        break;
    case 'n':
        TimeNow();
        break;
    case 'z':
        FileSize();
        break;
    case 'x':
        p_exit(0);
    }
}
```

there is, effectively,

```
LOCAL_C VOID ManageCommand(INT index)
{
    switch (index)
    {
        ...
    case 5:
        DayOfWeek();
        break;
    case 9:
        TimeNow();
        break;
    case 14:
        FileSize();
        break;
    case 15:
        p_exit(0);
    }
}
```

and instead of `ManageCommand` being called in the simple context

```
LOCAL_C VOID MainLoop(VOID)
{
    INT ret;
    WMSG_KEY key;

    FOREVER
    {
        uGetKey(&key);
        if (key.keycode&W_SPECIAL_KEY)
            ManageCommand(key.keycode&(~W_SPECIAL_KEY));
        else if (key.keycode==W_KEY_MENU && !(key.modifiers&W_CTRL_MODIFIER))
        {
            ret=uPresentMenus();
            if (ret>0)
                ManageCommand(ret);
        }
    }
}
```

there is now one extra layer to navigate between `MainLoop` and `ManageCommand`:

```
LOCAL_C VOID TryExecuteCommand(INT keycode)
{
    keycode=uLocateCommand(keycode);
    if (keycode>=0)
        ManageCommand(keycode);
}

LOCAL_C VOID MainLoop(VOID)
{
    INT ret;
    WMSG_KEY key;

    FOREVER
    {
        uGetKey(&key);
        if (key.keycode&W_SPECIAL_KEY)
            TryExecuteCommand(key.keycode&(~W_SPECIAL_KEY));
        else if (key.keycode==W_KEY_MENU && !(key.modifiers&W_CTRL_MODIFIER))
        {
            ret=uPresentMenus();
            if (ret>0)
                TryExecuteCommand(ret);
        }
    }
}
```

The two mechanisms are obviously equivalent in general terms. However, the latter approach has been adopted throughout all the example applications. The following points can be cited in its favour:

- quite often, several commands can be grouped together and executed more efficiently, passing as a parameter to a common routine the command index (possibly less some base value)
- the switch statement on index is completely dense, and hence compiles much more leanly than a switch statement on accelerator
- the accelerator of a menu command is a less central aspect of it than its position in the menu bar; it is better to switch on a variable of greater importance than on one which is virtually an accident
- this method is *language-independent*: the accelerators can be changed for a foreign-language version, without having to re-compile the `ManageCommand` routine.

In practice, the numerical values of the command indices do not appear explicitly in code; rather, they are hidden through a sequence of `#defines`. See *query.c* for the details.

Example of floating point editor

The routine `Temperatures` called from `ManageCommand` to implement the conversion between Fahrenheit and Centigrade demonstrates floating point editors in dialogs:

```
LOCAL_C VOID Temperatures(VOID)
{
    DOUBLE fahr;
    DOUBLE cent;
    H_DI_FLOAT ffahr;
    H_DI_FLOAT fcent;
    INT index;

    fahr=32;
    cent=0;
    ffahr.value=&fahr;
    ffahr.low=(-17968);
    ffahr.high=18032;
    fcent.value=&cent;
    fcent.low=(-10000);
    fcent.high=10000;
    FOREVER
    {
        if (uOpenDialog("Convert temperature"))
            return;
        if (uAddDialogItem(H_DIALOG_FLOAT,"Fahrenheit",&ffahr))
            return;
        if (uAddDialogItem(H_DIALOG_FLOAT,"Centigrade",&fcent))
            return;
        index=uRunDialog();
        if (index<=0)
            break;
        if (index==2)
        {
            cent=(fahr-32)*5/9;
            Clip(&cent);
        }
        else
        {
            fahr=32+cent*9/5;
            Clip(&fahr);
        }
    }
}
```

As the variable names suggest, the current value in Fahrenheit is stored in `fahr`, and the current value in Centigrade is stored in `cent`. There are two floating point editors, with `fahr` and `cent` being the live variables. Appropriate maxima and minima are set up in each case.

The variables `fahr` and `cent` are initialised to 32 and 0 respectively. Each time the user presses ENTER, one or other of these variables is sensed, and the other is recalculated. Which is which depends on where the user has left the highlight in the dialog. Thus if the user has cursoried the highlight down to the Centigrade line and typed in a new value there, before pressing ENTER, the call `uRunDialog` returns 3 (the counting starts at 1 for the title line in the dialog) and hence `fahr` is calculated anew, from the latest value of `cent`.

Example of numeric editor

The routine `clip` alters the calculated value of eg `fahr` or `cent` so that it only features a specified number of decimal points. (Currently, there is no Hwif mechanism for having floating point editors perform such a *clipping* themselves.) The number of decimal points is governed by the static variable `ndp`, which is initially 2. The *Significance* menu command allows the user to alter this:

```

LOCAL_C VOID ChangeNdp(VOID)
{
    LONG lndp;
    H_DI_NUMBER num;

    if (uOpenDialog("Level of significance"))
        return;
    lndp=ndp;
    num.value=&lndp;
    num.low=0;
    num.high=4;
    if (uAddDialogItem(H_DIALOG_NUMBER,"Decimal places",&num))
        return;
    if (uRunDialog()<=0)
        return;
    ndp=(WORD)lndp;
    CalcSmall();
}

```

Note the requirement to have a LONG variable for the live variable of the numeric editor. This explains why a copy of ndp has to be made in the automatic variable lndp.

The routine CalcSmall recalculates some constants that are used in calls to Clip.

Examples of other dialog items

See the following routines in *query.c* for examples of other types of items in dialogs:

time editors	TimeDifference
action buttons	Horoscope, Combinations
secret input boxes	EnterPassword
text editors	EncryptMessage (non-scrolling), DecryptMessage (scrolling).

Suggestions for modifying Query

- Add at least one more conversion routine.
- Call `hdlgPosition` to position at least one dialog other than in the screen centre.
- Eliminate the need for the floating point emulator, by using routines such as `p_fadd` instead of direct manipulation of floating point numbers. Compare the size of the executable produced with that of the original *query.app*.
- Try to improve on the rather crude scheme in `MakeReadable` and `MakeUnreadable`, called respectively by `EncryptMessage` and `DecryptMessage`, to convert between a short, totally unreadable string of characters in the complete range of values 0 to 255 (as returned by `p_encrypt`), and a longer string with values in the range 32 to 111.

Getting serious: the Tables application

Whereas *Query* contains a collection of dialogs with little unifying principle, *Tables* contains a collection of dialogs all working around a common aim. This aim is to produce a revision aid for someone trying to learn some multiplication tables.

What the dialogs allow to be altered is the following aspects of the state of the application:

- how much time the user has in which to answer any multiplication question posed
- whether the tables end at 12 (as in 3 times 12, 7 times 12, and so on), or at 10, or wherever
- whether the questions posed all come from the same multiplication table, or from a variety, and in the latter case, the range of tables covered
- the running total score of correct answers can be reset to zero.

As well as containing the code to present these dialogs, *Tables* contains code to:

- record the state of the application in an environment variable on exit
- initialise the application appropriately, on start up, from this environment variable
- calculate and pose random multiplication questions
- present an edit box to receive the user's response
- simultaneously, count down a timer and progressively fill in a bar gauge display
- present feedback to the user on whether the answer proffered is correct.

Ta1 contains the dialogs and the environment variable code, but is otherwise devoid of any significant screen display. *Ta2* adds the display of the score so far and the range of values being tested; an animated action button resides in the middle of the remainder of the screen. *Ta3* actually poses random multiplication problems, and provides an edit box to receive the user's response. *Tables* itself adds in the timer, and presents the animated bar gauge display of the time elapsed.

The state of the application

This is recorded in a static instance, *state*, of the following struct:

```
typedef struct
{
    UWORD TableEnd;    /* where tables end */
    UWORD MaxTable;    /* maximum table to test */
    UWORD Tested;      /* number of questions since last reset */
    UWORD Correct;     /* number of correct answers since last reset */
    UWORD Timing;      /* number of seconds allowed for an answer */
    UWORD Mode;        /* which table is currently being tested */
} TSTATE;
```

with values being initialised, the very first time, by the statement

```
LOCAL_D TSTATE state={12,12,0,0,5,1};
```

The value 1 for the *Mode* field has the special meaning that *all* tables are to be tested (from 2 up to *MaxTable*).

The *Tested* and *Correct* fields are reset to zero, provided the user responds affirmatively to a query dialog, in the routine *ResetScore*.

The *TableEnd* and *MaxTable* fields are presented for editing, using numeric editors, in the routine *ChangeLimits*.

Another numeric editor, in the routine *ChangeTiming*, allows the user to alter *Timing*.

The routine *ChangeMode* allows the *Mode* field to be changed. This uses a choice list whose contents are dynamically defined - they vary from "2 times table" up to "n times table", where *n* is the current value of *MaxTable*, but also always include "All tables":

```
LOCAL_C VOID ChangeMode(VOID)
{
    H_DI_CHOICE ch;
    INT j;
    TEXT buf[20];

    if (uOpenDialog("Mode"))
        return;
    if (uBeginDCL(&ch))
        return;
    if (uGrowDCL(&ch,"All tables"))
        return;

    for (j=2; j<=state.MaxTable; j++)
    {
        p_atos(&buf[0],"%d times table",j);
        if (uGrowDCL(&ch,&buf[0]))
            return;
    }
}
```

```

    if (uAddDCL("Test which tables",&state.Mode,&ch))
        return;
    uRunDialog();
}

```

Using an environment variable

Instead of simply calling `p_exit` on receipt of the *Exit* menu command (in the manner of *Query*), the following code is executed:

```

LOCAL_C VOID ExitApplication(VOID)
{
    uErrorValue(p_setenvir(EnvName,ENV_NAME_LEN,&state,sizeof(TSTATE)));
    p_exit(0);
}

```

The code in `MainLoop` that responds to *Shutdown* messages from the System Screen also has to change to call `ExitApplication`:

```

LOCAL_C VOID MainLoop(VOID)
{
    WMSG_KEY key;

    FOREVER
    {
        uGetKey(&key);
        if (key.keycode==W_EVENT_KEY)
        {
            if (key.keycode==CONS_EVENT_COMMAND)
                ExitApplication();
        }
    }
    ...
}

```

There are actually two sorts of environment variables on the Series 3:

- those whose names and values are each ZTSs
- those whose name or values are other than a ZTS.

Correspondingly, there are two sets of routines for reading, writing, searching, or deleting environment variables. (In fact, the ZTS-related routines just layer above the more general ones.)

In this case, what is being stored in the environment variable is the content of the state struct - which is bound to contain embedded zeros, and so the more general routine `p_setenvir` has to be used, as opposed to the notionally simpler `p_setenv`. Accordingly, both the name and the value of the environment variable have to be passed in the form `buf, len`.

Since the name of the environment variable is obviously the same whether the variable is being set (on application exit) or being read (on application start-up), this has been hidden away using the static variable `EnvName` (statically initialised) and the `#define ENV_NAME_LEN`.

Memory consumption by environment variables

The only error that needs to be considered, on writing the environment variable, is lack of memory - either because the limit of 4K allocated for environment variables has already been reached, or because system memory is generally exhausted. The call `uErrorValue` around `p_setenvir` above informs the user should this transpire.

As a general principle, applications should only make sparing use of environment variables (otherwise they may even detract from the performance of some of the built-in applications). In order to preserve larger amounts of data between different invocations of an application, the data should be written to file - either with or without the explicit knowledge of the user.

Note that the name of an environment variable should not, unless authorised by Psion, contain a '\$' character. See the *Environment variables on the Series 3* section of the *Series 3 Programming Overview* chapter of the *Series 3/3a Programming Guide* for further information on this important naming convention for environment variables.

Complications on reading environment variables

Programs using environment variables should always bear in mind that it is theoretically possible for another application to trash the variable - as a result of using another environment variable of the same name. In particular, the *length* of the variable may turn out to be other than what is expected. For this reason, the buffer to receive the environment variable should always be declared as having (at least) `P_ENVMAX` (256) bytes. (In order to access the definition of `P_ENVMAX`, a program must `#include` the file `p_sys.h`.) Hence

```
LOCAL_C VOID ReadState(VOID)
{
    INT len;
    UBYTE buf[P_ENVMAX];

    len=p_getenviron(EnvName,ENV_NAME_LEN,&buf[0]);
    if (len<0 || len!=sizeof(TSTATE))
        return; /* make do with the statically initialised values */
    p_bcopy(&state,&buf[0],sizeof(TSTATE));
}
```

The case when `len` is returned negative corresponds to the environment variable not existing - as will be the case the first time the application is run.

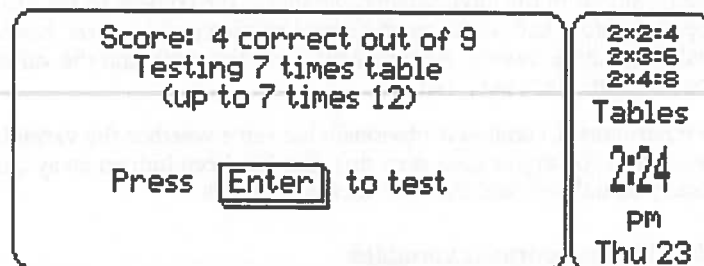
Suggestions for enhancing Ta1

- When `MaxTable` is large, the amount of memory required by the dynamic choice list in the dialog in `ChangeMode` can become considerable. Using either the Debugger or *Spy*, verify that this is the case, and try to re-design the dialog to require less RAM.
- If the user presses `PSION+ESCAPE`, *Ta1* is terminated without having any chance to save its state to an environment variable. Prevent this from happening.
- Strictly speaking, the code in `ReadState` above can be caught out by a rogue program which writes its own environment variable, with the same name and with the same length of data, but with inappropriate values for the individual fields. Write such a rogue program to demonstrate this fact, and consider amending `ReadState` to take better precautions.

Laying out information on the screen

Ta2 goes beyond *Ta1* in that it lays out its current state on the screen, for the user to see.

For example



There are no hard and fast rules for designing such a layout, but there certainly are easier and harder ways of going about *achieving* a given layout (once one has been decided upon). The following discussion may be read as an example of how to achieve a layout such as that shown in the above screen dump.

The three lines of text at the top of the screen share the following features:

- they are centred in the main window (apart from the status window)
- they need to be *smoothly updated* when there is a change in any of the values shown.

Both these reasons argue in favour of using `gPrintBoxText` to draw the lines. Not only can this function automatically centre text, it also takes care of the smooth screen update.

To explain the latter point more fully, consider what has to happen to the display of the second line down when the user changes from testing the 12 times table to testing the 7 times table. Not only does new text have to be drawn, some areas just outside the limits of the new text have to be cleared - since the new text is slightly narrower than the old.

Naively, the way to accomplish the above would be as follows:

- first, clear the area of screen where the old message was drawn
- second, draw the new text.

However, this can give rise to a noticeable and annoying screen flicker. This can be especially annoying if, as often happens, the text is updated when there is actually no change in it.

The routine `gPrintBoxText` avoids these problems by *simultaneously* clearing pixels and drawing to them, sweeping along in a horizontal pass. Pixels are written to if new text is to appear there, and are otherwise cleared.

The three top lines are written by a common routine:

```
LOCAL_C VOID DrawLine(INT j, TEXT *pb)
{
    P_RECT box;

    box.tl.x=4;
    box.br.x=189-4;
    box.tl.y=4+9*j;
    box.br.y=box.tl.y+9;
    gPrintBoxText(&box, 8, G_TEXT_ALIGN_CENTRE, 0, pb, p_slen(pb));
}
```

which is called as follows

```
LOCAL_C VOID DisplayScores(VOID)
{
    TEXT buf[40];

    p_atos(&buf[0], "Score: %u correct out of %u", state.Correct, state.Tested);
    DrawLine(0, &buf[0]);
}
```

and

```
LOCAL_C VOID DisplayMode(VOID)
{
    TEXT *pb;
    TEXT buf[40];

    if (state.Mode==1)
        pb="Testing all tables";
    else
    {
        p_atos(&buf[0], "Testing %d times table", state.Mode);
        pb=&buf[0];
    }
    DrawLine(1, pb);
}
```

and

```
LOCAL_C VOID DisplayLimits(VOID)
{
    TEXT buf[40];

    p_atos(&buf[0], "(up to %d times %d)",
        (state.Mode==1? state.MaxTable: state.Mode), state.TableEnd);
    DrawLine(2, &buf[0]);
}
```

In all cases, the box drawn to has height nine pixels. This allows one pixel of leading between lines, in addition to the font height of eight pixels. Since at least one pixel has to be reserved for descenders (such as the bottom pixel in the *p* of *up*), the maximum allowed value for the ascent parameter to `gPrintBoxText` is 8 - as in the above routine. This means in fact that the extra pixel of leading goes *above* the corresponding line of text (not that it matters in this case).

Collectively, the text respects a border of four pixels all around. The value of 189 is the full screen width (240 pixels) of the Series 3 and *Workabout*, or of the Series 3a in Series 3 emulation mode, less the width of the status window (51 pixels).

Since there is no need for a margin parameter in this case, it is left at zero.

To take advantage of the increased screen width of the Series 3a in "native" mode (480 pixels), the sizing and positioning of the text would need to be changed and the above calculations reworked.

Laying out an action button and its associated text

The text *Press Enter to test* is rather harder to position, because of the embedded action button.

Start by considering the horizontal direction. 38 pixels is a good width for the action button (this is the width of the action buttons in dialogs). There are five characters in *Press*, say six to include one trailing space. This translates to around 36 pixels, allowing 6 characters per pixel. Likewise, there are seven characters in *to test*, which ends up as around 48 pixels. Thus the width of the entire display is $36 + 38 + 48$, ie 122 pixels. Centring this within 189 pixels gives an x-offset of around 34 for *Press*.

Vertically, there are some $80 - (4 + 3 \times 9) - 4$ pixels to play with, ie 45 pixels. With 8 pixels for the height of the text, this leaves 18 pixels clear above the top of the text, translating into a vertical offset of $4 + 3 \times 9 + 18 + 7$ pixels to the baseline of the text, ie 56 pixels.

This means that the code to display the middle line can be written as

```
LOCAL_C VOID DisplayPressButton(VOID)
{
    gPrintText(34,56,"Press",5);
    DrawButton(FALSE);
    gPrintText(34+36+38+6,56,"to test",7);
}
```

where *DrawButton* (discussed below) draws the button itself, in either its normal or its depressed state (depending on the parameter passed to it).

The above discussion again assumes a maximum screen width of 240 pixels which is valid for the Series 3 and *Workabout*, or the Series 3a in Series 3 emulation mode. The Series 3a in native mode has a maximum screen width of 480 pixels. Therefore, to take advantage of the larger screen size of the Series 3a, the above calculations would need to be reworked.

Positioning an action button vertically

As for the vertical positioning of the button, bear in mind that it requires at least 6 pixels for its "edge effects" at top and bottom:

- one pixel for the top line
- one clear pixel underneath that
- one pixel each for the two bottom lines
- one clear pixel between the bottom lines, and one above the upper of these lines.

For text in the system font, which has height 8 pixels, this means that the total height of the button should be at least 14 pixels. With a height of fourteen pixels, the baseline of the text comes $1 + 1 + 7$ pixels below the top of the box.

Since this must match the baseline of the accompanying text *Press* and *to test*, it follows that the top of the box should be at 56-9 pixels.

This the code for *DrawButton* is

```
LOCAL_C VOID DrawButton(INT state)
{
    P_RECT box;

    box.tl.x=34+36;
    box.br.x=34+36+38;
    box.tl.y=47;
    box.br.y=47+14;
    wDrawButton(&box,"Enter",state);
}
```

Animating the action button

When the user does indeed press ENTER, the button should visibly depress, before any multiplication question is posed. The code to animate the button is as follows:

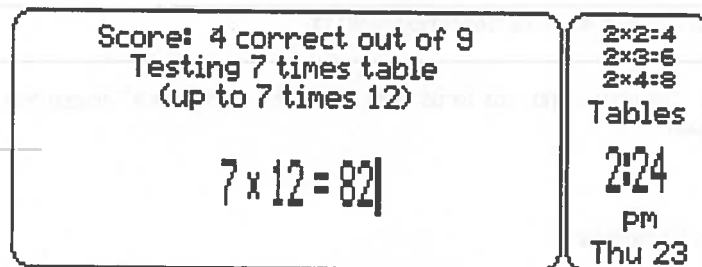
```
LOCAL_C VOID MakeButtonDance(VOID)
{
    DrawButton(TRUE);
    wFlush();
    p_sleep(2);
    DrawButton(FALSE);
    wFlush();
    p_sleep(2);
}
```

Suggestions for modifying Ta2

- Display the top line in bold
- Improve the above calculation so that it takes into account the fact that not all characters have widths of six pixels, and thereby position the test *Press Enter to test* yet more centrally
- Consider how to incorporate displaying the value of `state.Timing` too
- Experiment by removing the `wFlush` and/or the `p_sleep` from `MakeButtonDance`, to ensure that you understand their role in this routine.

Presenting an edit box

In *Ta2*, when the user presses ENTER, all that happens is that the score is incremented. In *Ta3*, the screen alters to the following form:



The left hand part is just the result of one more call to `gPrintBoxText`, for the text (in this case) `7 x 12 =`. The right hand part is an edit box for the user to enter the answer. There is a flashing cursor in the edit box.

The logic of positioning the text display and the edit box is somewhat similar to that above for positioning the action button and *its* surrounding text. The logic for presenting the edit box itself is new.

First, the edit box has to be created:

```
LOCAL_C VOID *CreateEditor(VOID)
{
    H_EDIT_BOX heb;

    heb.maxchars=4; /* allow up to four characters to be typed */
    heb.vulen=30; /* the width is 30 pixels (enough for 4 characters plus the cursor) */
    heb.pos.x=113;
    heb.pos.y=45;
    heb.win=MainWin; /* use the main (screen) window */
    heb.font=WS_FONT_BASE; /* use the standard font */
    heb.style=G_STY_DOUBLE; /* but with double height */
    return(HEBOpen(H_EDIT_BOX_FONT,&heb));
}
```

The meaning of the flag `H_EDIT_BOX_FONT` passed is that the font and style fields of the passed `H_EDIT_BOX` struct are significant.

After creating the edit box, it has to be instructed to display its flashing cursor:

```
HEBEmphasise(heb, TRUE);
```

Next, all suitable keypresses have to be diverted in its direction:

```
FOREVER
{
    uGetKey(&key);
    if (BadKey(key.keycode))
        Beep();
    else if (key.keycode!=W_KEY_RETURN)
        hEBHandleKey(ebH,key.keycode,key.modifiers);
    else
        ...
}
```

When the user again presses ENTER, the contents of the edit box have to be sensed:

```
LOCAL_C INT SenseNumberTyped(VOID *ebH)
{
    TEXT *pb;
    WORD num;

    pb=hEBSenseText(ebH);
    p_stoi(&pb,&num);
    return(num);
}
```

If the answer is as expected, the score is incremented, and the user is returned to the base state of the application. If the answer is incorrect, the user is given the opportunity either to retry, or to be told the correct answer. This interaction takes place via a couple of dialogs. If the user opts to retry, the last answer proffered is redisplayed, but completely highlighted so that any typing deletes it at once:

```
LOCAL_C VOID SelectAll(VOID *ebH)
{
    hEBSetSelect(ebH,0,p_slen(hEBSenseText(ebH)));
}
```

In all cases, when the application returns to its base state, the resources allocated for the edit box are freed by making the call

```
hEBClose(ebH);
```

Generating random numbers

The multiplication questions are generated very easily:

```
LOCAL_C INT FindRandom(INT low,INT high)
{
    INT range;

    range=high-low+1;
    return(low+(INT)(p_randl(&seed)%range));
}

LOCAL_C VOID FindFactors(WORD *pa,WORD *pb)
{
    *pa=(state.Mode==1? FindRandom(2,state.MaxTable): state.Mode);
    *pb=FindRandom(2,state.TableEnd);
}
```

The seed for the random variable generator is initialised by making the following call from `specificInit`:

```
seed=p_date();
```

Further comments on Ta3

Note the following sequence of calls, to print the multiplication question in double height:

```
SwitchStyle(G_STY_DOUBLE);
gPrintBoxText(&box,15,G_TEXT_ALIGN_RIGHT,0,&buf[0],p_slen(&buf[0]));
SwitchStyle(G_STY_NORMAL);
```

Switching the style back to normal again is clearly important, since otherwise, the next time any other call to `gPrintBoxText` is made, that text will end up in double height too.

When the editor is active, *Ta3* enters a special inner mode, with another get-event loop. Access to the menu bar is ruled out until the user has responded to the question in hand. Since no attention is paid to *Shutdown* messages during this inner mode, the Epoc static `datLocked` is set `TRUE` before entering this loop, and is cleared again on exiting it.

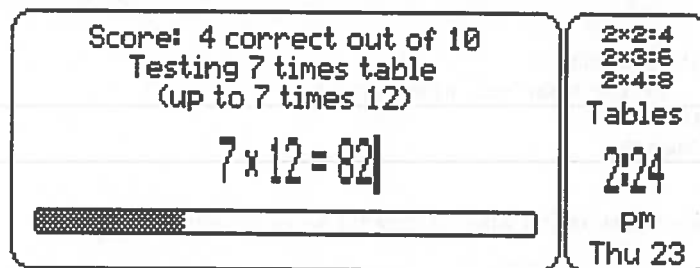
One difference between the edit box used in *Ta3* and those used in dialogs is that the former has double height display, for special emphasis purposes. This is of course not possible in dialogs. Another important difference is that handling the interaction with the editor directly, as in *Ta3*, allows the whole interaction to be terminated when a timer expires - as happens in the next step up from *Ta3*, namely *Tables* itself.

Suggestions for modifying *Ta3*

- Improve the inner get-event loop to respond to *Shutdown* messages from the System Screen
- Add another action button, with the text *Press Enter to confirm*, while the editor is displayed; to make room for this, change from using double height style to bold style
- Replace the code handling the edit box and its associated text with some invoking a suitable dialog (albeit with single-height lines); note how simpler the code is in this case
- Keep track of which questions the user answers incorrectly, and modify the code generating the questions so as to make these questions more likely to be asked again in the future.

Adding in a timer

When the user presses `ENTER` in the base state of *Tables*, the screen alters to the following:



The edit box and its accompanying text have moved up, and a bar gauge has appeared. This is incremented as time passes, and users have to complete their answer before the bar fills completely.

If the total time allowed is less than five seconds, the display updates once every half second; otherwise, it updates itself once a second.

In `specificInit`, a timer channel is opened, with the call

```
p_open(&timH, "TIM:", -1);
```

When an edit box is about to be displayed, the timer and some associated state variables are prepared for action by the routine

```
LOCAL_C VOID InitialiseTimer(VOID)
{
    if (state.Timing<=4)
    {
        timint=5;
        timcount=2*state.Timing;
    }
    else
    {
        timint=10;
        timcount=state.Timing;
    }
    timtotcnt=timcount;
    QueueTimer();
}
```

The variables `timcount` and `timtotcnt` are used in drawing the bar gauge.

Evidently, part of this preparation stage is to prime the timer:

```
LOCAL_C VOID QueueTimer(VOID)
{
    CancelTimer();
    p_ioa4(timH,P_RELATIVE,&timstat,&timint);
    timactive=TRUE;
}
```

The reason why the routine `QueueTimer` starts off with a call to `CancelTimer` is to cater for the case when the user retries an answer.

`CancelTimer` protects itself against cancelling an event that does not exist, by means of checking the variable `timactive`:

```
LOCAL_C VOID CancelTimer(VOID)
{
    if (timactive)
    {
        p_iow2(timH,P_FCANCEL);
        p_waitstat(&timstat);
    }
    timactive=FALSE;
}
```

As can be seen, `timactive` is set `TRUE` whenever any `p_ioc` call is made for the timer. It is set back to `FALSE` again, either inside `CancelTimer`, or in the get-event loop, whenever the expiry of the timer is detected:

```
p_iowait();
if (keystat==E_FILE_PENDING)
{
    /* the timer must have expired */
    timactive=FALSE;
    IncrementBarChart();
    ...
}
```

The routine `CancelTimer` is also called when exiting the inner get-event loop

```
...
DatLocked=FALSE;
hEBClose(ebH);
ClearBottomArea();
DisplayPressButton();
CancelTimer();
}
```

The call to `p_waitstat` inside `CancelTimer` is vital since, as for all the `P_FCANCELS` in Epoc, the timer `P_FCANCEL` does not stop the timer from completing (and thereby signalling). Rather, it *precipitates* the completion (if it has not already taken place).

Drawing the bar gauge

The outside of the gauge is drawn by a call to `gBorderRect`:

```
box.tl.x=8;
box.br.x=189-8;
box.tl.y=61;
box.br.y=61+8;
gBorderRect(&box,W_BORD_CORNER_1);
```

The grey pattern inside is drawn by calls to `gFillPattern`, using the built-in grey bitmap:

```
LOCAL_C VOID IncrementBarChart(VOID)
{
    P_RECT box;

    timcount--;
    GetBarChartRect(&box);
    box.br.x=9+171*(timtotcnt-timcount)/timtotcnt;
    gFillPattern(&box,WS_BITMAP_GREY,G_TRMODE_REPL);
}
```

The rectangle returned by `GetBarChartRect` is the same as that used to draw the outside of the gauge, except that it is inset by one pixel all around.

Limitation on debugging Tables

Due to a limitation in some earlier versions of the Series 3 ROM, applications such as *Tables*, which use asynchronous keyboard reads, may find they are unexpectedly panicked with panic 73, while debugging. This can arise in the following situations:

- the program has stopped at a break point when it has a keyboard read outstanding (ie the program has broken following a timer event), and a key is pressed on the Series 3
- or, the program has stopped at a break point when a timer has been queued, and the timer expires when the program is broken.

In either case, the panic will not be immediate, but will occur later as the result of a signal being mis-identified. (Another problem that can occur, for the same reason, is that the timer will never complete.)

Suggestions for enhancing Tables

- Allow access to the *Timing* menu command only when a suitable password is supplied; this password could be set (via another menu command) only by a "supervisor", and the "student", without knowing the password, would be unable to alter the time allowed for each question
- Currently, the Tested and Correct fields can become arbitrarily high; impose some kind of limit
- Consider a mode in which questions are posed repeatedly, without the user needing to press ENTER between every question; ask up to n questions repeatedly, where n has been set in advance by the user
- Currently, the timer is reset for each question; allow users to answer as many questions as possible during a total amount of time specified; give points for correct values and deduct points for incorrect answers.

The remaining example applications

Much could be said about the remaining example applications which cover a wide variety of different function calls and programming ideas. However, any readers who have managed to follow the discussion so far in this chapter will be well placed to unravel the contents of these other applications by themselves.

One possible exception is the use of a resource file in the *Remind* example application (This feature was not present in earlier versions of this application).

Resource file access with REMIND

See the chapter on *Resource Files* in the *Additional System Information* manual for background information about the value and use of resource files generally.

In *Remind*, all text has been removed from the source module *remind.c* and has been placed in suitable structures in *remind.rss*. Code in *remind.c* sees that these resources are loaded when needed.

Several aspects of this should be noted:

- The custom project file *remind.pr* contains the instruction "runrs remind" which has the result of creating the binary file *remind.rsc* from the input plain text file *remind.rss* using the batch file *rs.bat* which in turn invokes the resource compiler *rcomp.exe*
- This project file runs the resource compiler UNCONDITIONALLY but a more sophisticated project file, as discussed in the *Object Oriented Programming Guide*, could avoid recompiling the resource file unnecessarily (assuming no changes have been made)
- The binary file *remind.rsc* is listed in the add-file-list file *remind.afl* to ensure that it is automatically linked together with the object code as part of the application file *remind.app*
- The routine `LoadMenus` in *remind.c* loads the menu text out of the resource file into static data structures AND THEN "walks" these data structures, converting them into the form required by the Hwif menu subsystem
- In contrast, string data is only loaded into memory when required using the function `LoadStr`

The code in *remind.c* is copiously commented.

CHAPTER 3

ADVANCED USE OF HWIF

This chapter describes how to build your own version of the Hwif library and hence how to add your own extensions to Hwif. It also explains, with examples, how to combine Hwif programming with the use of Psion's object oriented programming techniques.

Building the Hwif library

Buildable source of the Hwif library is supplied as an HWIFSRC component on the Optional disk of the SIBO C SDK software. If installed, this source is copied into a `\sibosdk\hwifsrc` directory.

This directory should contain all the source files necessary to build your own version of the Hwif library, but you may need to insure that the *ts.red* file is suitable for your environment. Since the Hwif source code contains some object oriented software, you will also need to install the OOP component from the Optional disk before building the library.

Executing the *make.bat* batch file in the `\sibosdk\hwifsrc` directory will create an *hwif.lib* file that should be identical, apart from four bytes of date-stamp information, to the *hwif.lib* that is copied into the `\sibosdk\lib` directory by installing the HWIF component from the Optional disk. Note that making the Hwif library will also create an *hwifo.lib* library, whose use is described later in this chapter.

Extending Hwif

Once you have successfully built an *hwif.lib* that reproduces the one supplied with the SDK software, you may, if you wish, add your own extensions. These will typically be additional utility functions, but could be anything that you wish to add to Hwif.

All you have to do is add further code, either to the existing Hwif source files, or to additional source files and then rebuild the library. You may, if you wish, modify the *make.bat* file to remove the line:

```
tscx /m hwifo /v0
```

so that the *hwifo.lib* library is not rebuilt. If you have written source code in additional files you will, of course, have to modify the *hwif.pr* project file to include them.

Combining Hwif with object oriented code

It is possible for an Hwif program to be written to use parts of the built-in object oriented libraries OLIB, FORM, HWIM and (on the Series 3a and *Workabout*) XADD. One of the most important advantages of doing this is to use one or more object oriented (HWIM) dialogs, via the `hooDialog` utility function. Because of its importance, the rest of this chapter concentrates on the techniques that allow the use of this function.

HWIM dialogs support several features not available to Hwif dialogs. For example:

- Subdialogs can be launched when the user presses Tab.
- The value shown in one field can be made to change dynamically according to changes made by users in other fields in the dialog.
- Features such as locking items or dimming items are also available.

See the *Object Oriented Programming Guide* and the *HWIM Reference* manual for a full discussion of programming HWIM dialogs. The following description assumes some familiarity with the contents of those manuals.

In the Psion programming system, the class definition of an object oriented class must appear in a category file, and each code segment may only be associated with a single category file. Since the Hwif library is associated with its own category file, this means that application-specific classes can not simply be added directly to an Hwif program.

When combining application-specific classes with an Hwif program, the classes can be defined:

- in a separate category file, associated with a separate (DYL) code segment;
- in a modified Hwif category file.

These two alternatives are described in the following two sections.

Using a separate DYL

This technique is illustrated by the *OQuery* example application that may be copied into a `lsibosdk\hwifood` directory by installing the HWIFOOD component of the Optional disk. This example will be recognised as a version of the standard Hwif *Query* example program.

In *OQuery*, only one dialog is converted into HWIM form; this is the dialog to scan forwards or backwards in time to find the next occurrence of a particular date combination. On running the application, the difference between this and the original Hwif dialog (in particular, its "flicker free" updating) should be immediately noticeable.

In addition to the changes needed to run an HWIM dialog, all use of the floating point emulator has been abolished by replacing explicit floating point manipulation with calls to the `p_fxxx` functions.

The resource file, *oquery.rss*, contains the following dialog resource:

```
RESOURCE_DIALOG oqd_date_combins
{
    title="Find date combinations";
    controls=
    {
        CONTROL
        {
            class=C_CHLIST;
            prompt="Day in week";
            info=CHLIST { rid=oqm_daynames; };
        },
        CONTROL
        {
            class=C_NCEDIT;
            prompt="Day in month";
            info=NCEDIT
            {
                low=1;
                high=31;
            };
        },
        CONTROL
        {
            class=C_DTEDIT;
            prompt="Found date";
            info=DTEDIT
            {
                flags=IN_DTEDIT_DDMYYYYY|IN_DTEDIT_INIT;
                low=0;
                high=935011;
            };
        },
    },
}
```

```

CONTROL
{
    class=C_ACLIST;
    info=ACLIST { rid=oqa_date_combins; };
}
};
}

```

The code associated with the HWIM dialog itself is in the two files *oqd.cat* (the class definition of the dialog) and *oqdc.c* (the source code for the dialog's method functions). These are the only two source files that are used to build the application's DYL, *oqd.dyl*, using the techniques explained in the *Object Oriented Programming Guide*.

The category file, *oqd.cat* contains the following definition of the `OQD_COMBINS` class that subclasses the HWIM `DLGBOX` class:

```

LIBRARY oqd

EXTERNAL olib
EXTERNAL hwim

INCLUDE dlgbox.g

CLASS oqd_combins dlgbox
{
    REPLACE dl_dyn_init
    REPLACE dl_key
}

```

which means that the class number of the dialog will be represented by the symbolic constant `C_OQD_COMBINS`.

The name of the DYL is included in the DYL file list in *oquery.dfl*, which means that the DYL will be built into the final image file, *oquery.img*. This is the preferred way of packaging a DYL with an application.

The code associated with running this dialog is in *oquery.c*. In the function `SpecificInit()` the DYL is loaded and its handle written to the static variable `DylHandle` by:

```
DylHandle=hLoadOwnDyl(0);
```

where the zero parameter indicates that *oqd.dyl* is the first (and, in this case, the only) DYL built into the *.img* file.

The dialog is run, from the `ManageCommand()` function, by:

```
hOODialog(DylHandle,C_OQD_COMBINS,OQD_DATE_COMBINS,NULL);
```

Note that static data is not allowed in a DYL; all data transfer between an Hwif program and an HWIM dialog has to be via:

- the `rbuf` result buffer;
- the EPOC magic statics `DatApp1` through `DatApp7`, which are specifically designed for this kind of use.

In this application, there is no transfer of data to or from the dialog, and so the `rbuf` parameter is set to `NULL`.

Also note that, in this example application, the DYL is never unloaded by application code. In general, a DYL should be unloaded (using the function `p_unloadlib`) as soon as the code it contains is no longer required.

Debugging an Hwif DYL

Note that, when using the SIBO Debugger, you can only apply breakpoints in a code segment (or see its source code) when the code segment is loaded.

If you wish to apply a breakpoint in, or otherwise debug, a DYL used by an Hwif program you must first run the program until the DYL is loaded. In the case of the *OQuery* program you could, for example, set a breakpoint on the line:

```
DylHandle=hLoadOwnDyl(0);
```

On stepping through this line, the DYL is loaded and you can then debug it as normal.

You can, for example, view its code by using the Source module option of the Debugger's View menu and selecting the appropriate code segment and source file in the resulting dialog.

You can set a breakpoint in the DYL without first viewing its code, provided you specify the code segment name (which is the same as the name of the DYL). Suppose you wish to set a breakpoint in *oqd.dyl*, on the *dl_dyn_init* method of the *oqd_combins* dialog. While the DYL is loaded, you can, in the Debugger's Set breakpoints dialog, add the breakpoint by typing in:

```
OQD:oqd_combins_dl_dyn_init
```

Modifying the Hwif category

The second way of introducing application-specific object classes into an Hwif application is by incorporating them into the Hwif category file. Effectively, this is the same idea as described earlier to add extensions to Hwif.

In principle, the way to do this is to:

- add the application-specific class definitions to the end of the Hwif category file, *hwif.cat*,
- write the additional code in one or more additional source files,
- compile and link all the source files, including those of Hwif, into the application *.img* file.

In practice, it is more convenient to do this in a different, but totally equivalent, way. Instead of modifying the Hwif category file, you create an application-specific category file. The initial lines of this file must be an exact copy of the Hwif category file, *hwif.cat*, whose content is shown below.

```
IMAGE hwif

EXTERNAL olib
EXTERNAL hwim

INCLUDE lprinter.g
INCLUDE help.g

CLASS hprinter lprinter
{
    REPLACE lpr_read
    REPLACE lpr_sense_text
    PROPERTY
    {
        INT (*sense)(WDR_PRINT *);
    }
}

CLASS hhelpdlg helpdlg
{
    REPLACE destroy
}
```

You may change the file name and the name in the *IMAGE* statement to match the particular application. Note that the two names must be the same, so that, for example, a category file called *myapp.cat* must start with the statement

```
IMAGE myapp
```

You may also, if necessary for the application, add further *EXTERNAL* and/or header file *INCLUDE* statements.

Apart from these possible changes and additions, the first part of the file must match the contents of the *hwif.cat* file exactly. This data should then be followed by the application-specific class definitions.

Further files contain the application source code exactly as for a normal Hwif application, except that they also include the method function code for the application-specific classes.

After compiling these files, you should link them with the *hwifo.lib* library file, rather than the normal *hwif.lib*.

The file *hwifo.lib* is copied into the `\sibosdk\lib` directory when you install the HWIF component from the Optional disk. It can also be built from the Hwif source code, as described earlier in this chapter. It differs from *hwif.lib* only in that it does not contain the Hwif category data.

The technique of modifying the Hwif category file is illustrated by the *gbar.img* example code that is described in the following section.

Access to a growing scroll bar from Hwif

The source code for the *gbar.img* example application is copied into the `\sibosdk\hwifood` directory by installing the HWIFOOD component from the Optional disk.

This example, in addition to illustrating the technique of including object oriented code by modifying the Hwif category, also provides an example of how to use a growing scroll bar, or percentage done indicator, in Hwif.

Note that including a growing scroll bar in a dialog is only possible when the dialog is fully object oriented, that is, either in an HWIM application, or in a dialog called from Hwif via `hooDialog()`.

To make *gbar.img*, run the *makegbar.bat* batch file in `\sibosdk\hwifood`.

To run it, copy it to a top-level `\IMG\` directory and run from under RunImg. When finished, press Psion-Esc to exit it.

The first dialog in the loop lets you specify the parameters for how the second, growbar, dialog operates.

The application's category file

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

```
IMAGE gbar

EXTERNAL olib
EXTERNAL hwim

INCLUDE lprinter.g
INCLUDE help.g
INCLUDE dlgbox.g

CLASS hprinter lprinter
{
    REPLACE lpr_read
    REPLACE lpr_sense_text
    PROPERTY
    {
        INT (*sense)(WDR_PRINT *);
    }
}

CLASS hhelptlg helptlg
{
    REPLACE destroy
}
```

```
CLASS data_dl dlgbox
{
    REPLACE dl_dyn_init
    REPLACE dl_key
    TYPES
    {
        typedef struct
        {
            UWORD exit_code;
            UWORD totloops;
            UWORD update;
            UWORD esc;
            UWORD loopsleft;
            UWORD toupdate;
        } RB_GBAR;
    }
}
```

```
CLASS gbar_dl dlgbox
{
    REPLACE wn_sense_help
    REPLACE dl_dyn_init
    REPLACE dl_key
    ADD gbar_update
}
```

```
CLASS bar_ao active
{
    REPLACE ao_init
    REPLACE ao_run
    PROPERTY
    {
        RB_GBAR *prb;
    }
}
```

Comparing this with *hwif.cat* shows that:

- the IMAGE name has changed,
- there is the additional inclusion of *dlgbox.g*
- there are application-specific class definitions for the two dialogs DATA_DL and GBAR_DL, and the BAR_AO active object.

This application uses a pointer to a result buffer (in this case, an RB_GBAR struct in the property of the BAR_AO active object) to communicate with the dialogs. This is a design decision that is mentioned later.

C_DONEWN items in dialogs

As shown by the DL_GBAR dialog resource in *gbar.rss*, a growing scroll bar item in a dialog is defined in the resource file simply as:

```
CONTROL
{
    class=C_DONEWN;
}
```

Compare this with, for example, the definition of a numeric editor control:

```
CONTROL
{
    class=C_NCEDIT;
    prompt="Total number of loops";
    info=NCEDIT
    {
        high=100;
        low=1;
    };
}
```

The differences are:

- there is no associated prompt
- there is no info data.

When a dialog contains a `c_DONEWN` element, that element must be sent a `WN_SET` message, defining the range of the element. This must be done during the initialisation of the dialog, before it becomes visible, usually from the `dl_dyn_init` method of the dialog. For example, in the code of *growbar.c*:

```
METHOD VOID gbar_dl_dyn_init(PR_DLGBOX *self)
{
    RB_GBAR *prbuf;
    SE_DONEWN set;

    prbuf=self->dlgbox.rbuf;
    set.flags=SE_DONEWN_RANGE;
    set.range=prbuf->totloops;
    hDlgSet(1,&set);
}
```

Note that the `WN_SET` message to the `c_DONEWN` element (the element with index 1 in the dialog) is hidden in the HWIM utility function `hDlgSet()`.

The `SE_DONEWN` struct and the `WN_SET` method of `DONEWN`

The above code uses an `SE_DONEWN` struct to pass information in the `WN_SET` message. This struct is defined as follows:

```
typedef struct
{
    UWORD flags;
    ULONG val;
    ULONG range;
} SE_DONEWN;
```

The value of `flags` can be any one of the following:

<code>SE_DONEWN_RANGE (0x01)</code>	set the control's range
<code>SE_DONEWN_VALUE (0x02)</code>	set the current value to <code>val</code>
<code>SE_DONEWN_INCREMENT (0x04)</code>	increase value by 1
<code>SE_DONEWN_INC_VAL (0x08)</code>	increase value by <code>val</code>

For example, inside the `gbar_update` method of the `GBAR_DL` dialog in *growbar.c* it is used to set a specific current value:

```
METHOD VOID gbar_dl_gbar_update(PR_DLGBOX *self)
{
    RB_GBAR *prbuf;
    SE_DONEWN set;

    prbuf=self->dlgbox.rbuf;
    set.flags=SE_DONEWN_VALUE;
    set.val=prbuf->totloops-prbuf->loopsleft;
    hDlgSet(1,&set);
}
```

Note that the above `gbar_update` method is not a `REPLACED` method but one that has been `ADDED` by the definition of `GBAR_DL` to the set defined by the superclass `DLGBOX`.

The `BAR_AO` active object

A typical use of a grow bar dialog is to report on the progress of an extended activity. The "update" method of the dialog has to be called every so often, during the course of the activity being described in the dialog.

However, this activity has to take place in between the dialog starting and the dialog exiting and therefore has to take place inside the `ao_run` method of an active object.

The example uses the `BAR_AO` active object which is created and initialised early in the `MainLoop()` function in *growbar.c* by the code:

```
InitGbarData(&gb_data);
...
ao=f_newsend(CAT_GBAR_GBAR,C_BAR_AO,O_AO_INIT,&gb_data);
```

which calls the object's `ao_init` method:

```
METHOD VOID bar_ao_ao_init(PR_BAR_AO *self,RB_GBAR *prb)
{
    p_send3(w_am,O_AM_ADD_TASK,self);
    self->bar_ao.prb=prb;
}
```

In the example, the activity is simulated in the `ao_run` method by simply calling `p_sleep(5)` to pause the application for half a second. The code for the method is:

```
METHOD INT bar_ao_ao_run(PR_BAR_AO *self)
{
    RB_GBAR *prb;

    p_sleep(5); /* the next stage of computation or drawing etc */
    p_send2(self,O_AO_QUEUE);
    prb=self->bar_ao.prb;
    if (!--(prb->loopsleft))
    {
        p_send2(DatDialogPtr,O_DESTROY);
        prb->exit_code=0;
    }
    else if (!--(prb->toupdate))
    {
        p_send2(DatDialogPtr,O_GBAR_UPDATE);
        prb->toupdate=prb->update;
    }
    return(RUN_ACTIVE_USED);
}
```

Note the following points about the code of this method:

- The next stage of the "computation" is, for convenience, always queued in the `ao_run` method (by sending an `AO_QUEUE` message, which will cause the `ao_run` method to be called again at some time in the future).
- In consequence, the grow bar dialog itself can be called by the code:

```
p_send2(ao,O_AO_QUEUE);
doDial(C_GBAR_DL,DL_GBAR,&gb_data);
p_send2(ao,O_AO_CANCEL);
```

which always cancels the activity of the active object when the dialog completes.

- The handle of the current dialog is always accessible from the "reserved static" `DatDialogPtr`.
- If the computation is finished, the active object sends a `DESTROY` message to the dialog. Otherwise, every so often, the active object sends an "update" message to the dialog.
- Don't forget to


```
return(RUN_ACTIVE_USED)
```

 from your `ao_run` method (on pain of being panicked 143, most likely).
- In this example, there are no parameters to the "update" message, but in another example there might be.

Termination of the grow bar dialog

In general, the grow bar dialog can terminate in either of two ways:

- the user presses `Esc` - in which case the call to `hooDialog` returns without the cooperation of any of the code in the `ao_run` method
- the computation finishes - in which case the `ao_run` method sends the dialog a `DESTROY` message and this precipitates the completion of the `hooDialog` call.

General comments

Help has to be disallowed while the grow bar dialog is running, to avoid accidents if the dialog is terminated before the help system is shut down. (These complications only arise for `h00bdialog`, and not for pure HWIM programs.) Hence the "magic" in the dialog's `wn_sense_help` method:

```
METHOD VOID gbar_dl_wn_sense_help(VOID *self)
{
    p_leave(RUN_ACTIVE_USED);
}
```

For a similar reason, access to the freeform dialler has to be disallowed - hence the "magic" in the routine `DisallowDialling`:

```
LOCAL_C VOID DisallowDialling(VOID)
{
    w_ws->wserv.flags|=PR_WSERV_FREEFORM_DIALLING;
}
```

For simplicity, the code deliberately ignores various run-time errors that might arise - for example, running out of memory when launching either dialog.

In *growbar.c*, the active object is created early in the application, and is used repeatedly each time the grow bar dialog is invoked. Another design approach would be to have the active object exist only throughout the lifetime of an individual grow bar dialog.

Another design decision in this example is not to access any specific static data from inside either the dialog code or the active object code. Each of these object interacts with the rest of the world only via the result buffer pointers, as noted earlier. An alternative design option would be to access more static data from inside these objects.

CHAPTER 4

HWIF REFERENCE DOCUMENTATION

Overview of the Hwif library

Routines in the Hwif library fall into two categories:

- *utility functions*, which have names starting with lower-case *u*
- *primitive functions*, which have names starting with lower-case *h*.

The former category are routines which an experienced Hwif programmer could dispense with or re-write. They layer over Window Server function calls, Console I/O requests, and some of the more primitive Hwif routines. They turn out to be very useful in practice, but if the need arises, they can in principle be replaced by alternative code.

On the other hand, the *h*-routines can be replaced only by someone familiar with Psion's proprietary object-oriented system.

Two levels within the h-layer calls

In turn, functions in the h-layer of Hwif can be further classified:

- *Low level* functions, which would normally be accessed directly only by programmers providing their own versions of the u-level functions
- *High level* functions, which are more widely useful - such as `hPrint`, `hEBSenseText`, and `hDTMFString`.

Two layers within the u-layer calls

The u-layer calls in the Hwif library can also be classified into two types:

- *Central* functions, which are likely to be called in every non-trivial Hwif application, and which encapsulate detailed knowledge of the operation of the low level h-layer Hwif functions
- *Auxiliary* functions, whose contents are more straightforward, and which can more easily be duplicated by applications programmers.

Many of the auxiliary functions in fact exist in the library only because they are called from within other Hwif functions. It would be wasteful for applications to create their own versions of these functions, since this would lead to *two* duplicate functions in the same application.

Groups of functions and naming conventions

The following groups of high-layer h-routines each have their own naming convention, to clarify their roles:

<code>hEBxxx</code>	<i>edit box</i> functions
<code>hTTxxx</code>	<i>time text</i> functions
<code>hDTxxx</code>	<i>date/time text editor</i> functions
<code>hPrintxxx</code>	<i>printing and print support</i> functions.

Return values and error notification

In most cases, Hwif functions that can fail return negative values to indicate the nature of the failure, and zero or a positive value to indicate success. (The exceptions include a few *constructive* functions, which return the handle of some allocated block if successful, and NULL if not.)

For a function which can only have two outcomes, success or failure, the former is usually indicated by the return value 0, and the latter by the return value -1. In practice, the outcome can be determined by the caller making a simple test, such as

```
if (TryOpenDbf(...))
    ...      /* the call to TryOpenDbf failed */
else
    ...      /* the call succeeded */
```

or

```
if (uOpenDialog(NULL))
    return;  /* no memory to open the dialog */
...
```

Most Hwif library routines automatically notify the user of any error that arises, before returning the error value to the caller. It is only the low level h-layer routines that leave the notification to the caller.

Binary counted strings

Although most of the Series 3 ROM functions work with ZTSs (zero terminated strings) as opposed to BCSs (leading byte-counted strings), some of the functions within Hwif instead work with BCSs.

Where conversion between the two types of representation of string is required, this is obviously straightforward.

In all cases below, strings are assumed to be given in ZTS form, unless otherwise stated.

Dialog and menu interactions as a special mode

An Hwif application enters a special mode when it makes the calls `uPresentMenus`, `uRunDialog`, `hPrint` or `hPrintSetupDialog`:

- messages from the System Screen are blocked during this time, with the user being told that the application is "busy"
- if, during a menu or dialog interaction, the application is tasked into background or foreground, or is switched off then on again, the application is not notified of this fact
- if a timer expires, or another non-keypress event occurs, the application only finds out about this when the user in due course concludes the menu or dialog interaction.

In the above cases, the processing of events passes temporarily out of the hands of application, to a central get-event loop in ROM code. This ROM get-event loop can only process events from event sources it explicitly knows about - and this excludes any timers, alarms, or other non-keypress event sources installed by the application. If the ROM get-event loop detects a signal without the status words of any of the event sources it knows about being written to (as will occur if, for example, an application-installed timer expires), it simply increments an internal counter. When program execution is about to pass back out of the ROM get-event loop to the application, one signal is re-emitted for every time the internal counter was incremented.

The central functions in the u-layer of Hwif

uCommonInit Initialisation common to Hwif applications

```
VOID uCommonInit(VOID);
```

Opens and sizes a console window appropriate for the Series 3, Series 3a or *Workabout* screen. Initialises control blocks for subsequent menu and dialog interactions.

If unsuccessful (for example, because the Window Server has insufficient memory to open the console window), the application is terminated with an in-line call to `p_exit`.

If successful, the address of the control block of the console is written to the globally referenced static

```
VOID *winHandle;
```

The routine copes with the case when `winHandle` is non-zero when the routine is called - usually because a CLIB start-up module has been used. Rather than attempt to open the console again, the existing console channel is used, with any flashing block cursor being turned off.

A basic example of the use of `uCommonInit` is:

```
GLDEF_C VOID main(VOID)
{
    uCommonInit();
    SpecificInit();
    MainLoop();
}
```

The general behaviour and appearance of the console window that is created in a call to `uCommonInit` depends on the value of the global variables `_UseFullScreen` and `_S3UseFullScreen` on entry to the call.

On all machines, if the values of `_UseFullScreen` and `_S3UseFullScreen` are zero (the default values) `Hwif` is initialised in Series 3 compatibility mode. This means that the behaviour and appearance of the console on the Series 3a or *Workabout* emulates the behaviour of the console on the Series 3.

If, however, the value of `_UseFullScreen` is non-zero, `Hwif` is initialised appropriately for the machine on which the application is running. On the Series 3a and *Workabout*, the application can take full advantage of the larger screen size, grey and other additional features.

The value of `_S3UseFullScreen` is ignored by Series 3 and Series 3a machines. On the *Workabout*, if `_S3UseFullScreen` is non-zero (and regardless of the value of `_UseFullScreen`) `Hwif` is initialised to use the Window Server's `W_CTBY_S3_SCR` compatibility mode. This is a Series 3 (i.e. no grey) compatibility mode, but the full 240x100 area of the *Workabout* screen is used. For example:

```
GLREF_D UWORD _S3UseFullScreen;

GLDEF_C VOID main(VOID)
{
    _S3UseFullScreen=TRUE; /* Set Workabout to S3 full-screen compatibility */
    uCommonInit();
    ...
}
```

Note that on all machines (except the *Workabout* in true Series 3 compatibility mode, where a 240x80 window is used) the console window is created to be occupy the full screen. Regardless of its size, it will always display an exact number of lines of text.

In addition to its use to specify whether or not to use compatibility mode, the value of `_UseFullScreen` can be tested on *return* from `uCommonInit` to determine the type of the machine on which the code is running; a non-zero value means that the application is running on a Series 3a or *Workabout*, while a zero value means that the machine is running on the Series 3. This is demonstrated in the following code fragment:

```
GLREF_D UWORD _UseFullScreen;

GLDEF_C VOID main(VOID)
{
    _UseFullScreen=TRUE;
    uCommonInit();
    if (_UseFullScreen)
    {
        /* Running on a Series 3a or Workabout */
        ...
    }
    else
    {
        /* Running on a Series 3 */
        ...
    }
}
```

A call to `uCommonInit` does not write anything to `_S3UseFullScreen` so there is no equivalent test that can be made on its value.

uEnableGrey**Enable use of grey**

```
VOID uEnableGrey(VOID);
```

This function enables the use of grey in the main console window. It *must* be called before any attempt is made to draw grey on the Series 3a or *Workabout*.

Because of the overhead involved, grey should *not* be enabled as a matter of routine. If the application never intends to draw grey then it should not be enabled!

Ideally, it should be called as soon as possible after the call to `uCommonInit`. In general, a call to this function should be imbedded in the initialisation code specific to the application. Referring to the general structure of an Hwif program as mentioned in the description of `uCommonInit`, the function `SpecificInit()` is usually a good place to imbed a call to `uEnableGrey`.

The following program is a very simple example that draws a shadowed grey effect border using the function `gBorder2`, if running on the Series 3a or the *Workabout*; if running on the Series 3, or on the Series 3a or *Workabout* in compatibility mode, it draws a simple shadowed border without grey.

On the Series 3a and *Workabout*, grey *must* be enabled before the grey shadowed effect border can be drawn.

```
#include <p_std.h>
#include <wlib.h>
#include <hwif.h>

GLREF_D UWORD _UseFullScreen;

LOCAL_D WORD    keystat;
LOCAL_D INT     gc;

LOCAL_C VOID SpecificInit(VOID)
{
    gc = gCreateGCO(uFindMainWid());
    if (_UseFullScreen)
    {
        wFree(gc);
        uEnableGrey();
        gc = gCreateGCO(uFindMainWid());
        gBorder2(W_BORDER_TYPE_1, W_BORD_CORNER_4 | W_BORD_SHADOW_ON | W_BORD_SHADOW_D);
    }
    else
        gBorder(W_BORD_CORNER_4 | W_BORD_SHADOW_ON | W_BORD_SHADOW_D);
}

LOCAL_C VOID MainLoop(VOID)
{
    WMSG_KEY key;

    FOREVER
    {
        uGetKeyA(&keystat, &key);
        p_iowait();
        if (key.keycode & W_SPECIAL_KEY)
            if ((key.keycode & (~W_SPECIAL_KEY)) == 'x')
                p_exit(0);
    }
}

GLDEF_C VOID main(VOID)
{
    _UseFullScreen = TRUE;
    uCommonInit();
    SpecificInit();
    MainLoop();
}
```

It is important to note that `uEnableGrey` causes the ID of the main window to change. Thus, after calling `uEnableGrey` and before calling any functions that need the main window ID as a parameter, call `uFindMainWid`.

uGetKey **Read a key, waiting for its delivery**

VOID uGetKey(WMSG_KEY *pkey);

Reads a keypress into *pkey, waiting indefinitely if no keypress is received.

The routine also returns if any special event is received; in this case, the keycode has the bit W_EVENT_KEY set. The various possible values in this case are given in the header file *p_cons.h*:

CONS_EVENT_FOREGROUND	The application has passed into foreground
CONS_EVENT_BACKGROUND	The application has passed into background
CONS_EVENT_ON_OFF	The machine has been switched off and then on again
CONS_EVENT_COMMAND	The application has received a message (probably from the System Screen) and should call wGetCommand to obtain a buffer containing more details.
CONS_EVENT_DATE_CHANGED	The system date has changed. Typically, the application will get this message when the system date passes midnight.

Note that this is only available on Epoc V3.18 or later and Window Server V4.32 or later

For example:

```
LOCAL_C VOID MainLoop(VOID)
{
    WMSG_KEY key;

    FOREVER
    {
        uGetKey(&key);
        if (key.keycode & W_EVENT_KEY)
        {
            if (key.keycode == CONS_EVENT_COMMAND)
                ProcessSystemCommand();
        }
        else ...
    }
}
```

uGetKeyA **Read a keypress asynchronously**

VOID uGetKeyA(WORD *pstat, WMSG_KEY *pkey);

Reads a keypress into *pkey whenever the next keypress is received, without however waiting for this to occur. The value of *pstat is changed to E_FILE_PENDING when the call is made. When a keypress is received, the value of *pstat is changed from E_FILE_PENDING to 0.

An application that calls uGetKeyA when the previous such call is still outstanding is liable to be panicked in due course with panic 73.

For example:

```
LOCAL_D WORD timstat;
LOCAL_D WORD keystat;
LOCAL_D WORD keyactive=FALSE;
...

LOCAL_C VOID MainLoop(VOID)
{
    WMSG_KEY key;

    FOREVER
    {
        if (keyactive)
            wFlush(); /* flush any outstanding graphics calls */
        else
        {
            uGetKeyA(&keystat,&key);
            keyactive=TRUE;
        }
        p_iowait(); /* wait for something to happen */
        if (keystat==E_FILE_PENDING)
        {
            ... /* the timer must have expired */
        }
        else
        {
            keyactive=FALSE;
            ... /* proceed as above */
        }
    }
}
```

uCancelGetKeyA Cancel outstanding asynch' keypress request

VOID uCancelGetKeyA(VOID);

This function cancels any outstanding asynchronous request for a keypress or any of the other event types described in uGetKey.

Consider the code fragment given as an example in the description of uGetKeyA. This could be modified to include a call to uCancelGetKeyA as soon as the timer has expired as shown below:

```

LOCAL_D WORD timstat;
LOCAL_D WORD keystat;
LOCAL_D WORD keyactive=FALSE;
...

LOCAL_C VOID MainLoop(VOID)
{
    WMSG_KEY key;

    FOREVER
    {
        if (keyactive)
            wFlush(); /* flush any outstanding graphics calls */
        else
        {
            uGetKeyA(&keystat,&key);
            keyactive=TRUE;
        }
        p_iowait(); /* wait for something to happen */
        if (keystat==E_FILE_PENDING)
        {
            ... /* the timer must have expired */
            uCancelGetKeyA(); /* cancel outstanding keypress requests */
            keyactive=FALSE; /* no key press requests outstanding */
            ...
        }
        else
        {
            keyactive=FALSE;
            ... /* proceed as above */
        }
    }
}

```

uKeyPressOutstanding**Test if a keypress is outstanding**

INT uKeyPressOutstanding(VOID);

Returns TRUE if a keypress is outstanding, else FALSE.

For example:

```

UpdatePending=FALSE;
FOREVER
{
    uGetKey(&key);
    switch (key.keycode)
    {
        ... /* may set UpdatePending */
    }
    if (UpdatePending && !uKeyPressOutstanding())
    {
        DrawIcon(); /* time consuming */
        UpdatePending=FALSE;
    }
}

```

Note that for the purposes of this routine, "keypress" does *not* include a general console event (such as coming into foreground).

uLocateCommand**Convert accelerator into command index**

INT uLocateCommand(INT accel);

Looks through the table of commands implicitly identified by the static `_cmds`, searching for a command with the accelerator `accel`.

Returns -1 if no match is found, or else the index of the matching command, starting with 0 for the first command.

For the Series 3, the sets of valid accelerators are:

- the lower case letters 'a' to 'z' inclusive.
- four characters that vary from language to language - in English they are '+', '-', '*', and '/'.

For the Series 3a and *Workabout*, the sets of valid accelerators are those which are valid for the Series 3 plus:

- the upper case letters 'A' to 'Z' inclusive.

The Series 3a and the *Workabout* distinguish between shifted and unshifted alphabetic accelerator keys. For example, PSION+A and PSION+SHIFT+A may be used to invoke two different commands.

Shifted accelerators are not available on the Series 3 and should not be used in software intended to run on any range of machine types that includes the Series 3.

A shifted accelerator is defined by an upper case accelerator in the command array, as for the *Search backwards* command in the following example:

```
LOCAL_D TEXT *cmds[] =
{
    "nNew File",
    "aSaveas",
    ...
    "sSearch forwards",
    "SSearch backwards",
    ...
    "xExit",
    NULL
};
```

Before calling `uLocateCommand`, a small change must be made to the accelerator key handling code; if the Shift Modifier is set, the accelerator key must be converted to uppercase. This is illustrated in the following code fragment:

```
LOCAL_C VOID TryExecuteCommand(INT keycode)
{
    INT comid;

    comid = uLocateCommand(keycode);
    if (keycode >= 0)
    {
        /* execute command with command ID comid */
        ...
    }
}
```

```
LOCAL_C VOID MainLoop(VOID)
{
    INT code;
    INT ret;
    WMSG_KEY key;
```

```

FOREVER
{
    uGetKey(&key);
    if (key.keycode & W_SPECIAL_KEY)
    {
        code = key.keycode & (~W_SPECIAL_KEY);
        if (key.modifiers & W_SHIFT_MODIFIER)
            code = p_toupper(code); /* code change needed to use shifted accelerators */
        TryExecuteCommand(code);
    }
    else
    {
        switch (key.keycode)
        {
            case W_KEY_MENU :
                if (key.modifier & W_CTRL_MODIFIER)
                {
                    /* toggle status window */
                    ...
                }
                else
                {
                    ret = uPresentMenus(); /* no code change here */
                    if (ret > 0)
                        TryExecuteCommand(ret);
                }
                break;
            case W_KEY_TAB :
                ...
                break;
            case W_KEY_RETURN :
                ...
                break;
        }
    }
}
}

```

Note that no change is required to the code concerned with selecting a command by highlighting its menu item and pressing Enter (implemented by a call to `uPresentMenus`).

uPresentMenus

Present a set of menus

INT uPresentMenus(VOID);

Commence a menu bar interaction, presenting the menus implicitly defined via the statics `_cmds` and `_mdata`. Waits until the menu interaction has terminated before returning.

This function returns:

- 0 if the user cancels or if an error such as out of memory (OOM) occurs - in which case the user will already have been notified of this
- the accelerator of the command chosen.

The Epoc static `DatLocked` is set TRUE for the duration of the call to `uPresentMenus`.

The call requires Window Server resources, and hence its success can never be guaranteed.

For example:

```

LOCAL_C VOID TryExecuteCommand(INT keycode)
{
    keycode=uLocateCommand(keycode);
    if (keycode>=0)
        ManageCommand(keycode);
}

```

```

LOCAL_C VOID MainLoop(VOID)
{
    INT ret;

    FOREVER
    {
        uGetKey(&key);
        if (key.keycode&W_SPECIAL_KEY)
            TryExecuteCommand(key.keycode&~W_SPECIAL_KEY);
        else if (key.keycode==W_KEY_MENU)
        {
            ret=uPresentMenus();
            if (ret>0)
                TryExecuteCommand(ret);
        }
        else ...
    }
}

```

A common situation in moderately complex applications is the need to display different menu bars as the context of the application changes.

For example, the built in spreadsheet application on the Series 3a has a different menu bar when running in graph mode compared to that when running in normal mode.

In switching between different menu bars in this way, the "position" of the menu item highlighted in one menu bar is often lost after switching to a different menu bar. In other words, after switching back to the original menu bar, the menu item highlighted is different to the one that was highlighted when this menu bar was last displayed. This problem is often a cause of irritation to users.

This difficulty can be avoided by using the global variable `_MenuPositions` available on the Series 3a only. The following code fragment illustrates its use:

```

GLREF_D UWORD *_MenuPositions;
...
LOCAL_D UWORD m1;
LOCAL_D UWORD m2;
...
... /* build main menu bar */
...
_MenuPositions = &m1;
uPresentmenus();
...
... /* build alternative menu bar */
...
_MenuPositions = &m2;
uPresentmenus();
.....
... /* re-build main menu bar */
...
_MenuPositions = &m1;
uPresentmenus();

```

In essence, Hwif uses `m1` and `m2` to store the "position" of the menu item highlighted. Before presenting a particular menu bar, the address of the corresponding `UWORD` variable (i.e. `m1` or `m2`) should be loaded into `_MenuPositions`.

uOpenDialog

Open a dialog

```
INT uOpenDialog(TEXT *title);
```

Prepares to display a dialog. The dialog has a title line given by the zero terminated string pointed to by `title`, unless `title` is `NULL`, in which case the title line is omitted.

Returns 0 for success, or else a negative error - in which case the user will already have been notified of the error.

On the *Workabout*, the dialog will be displayed in a small font if the global variable `_SmallFontDialog` is set to a non-zero value before calling `uOpenDialog`. If this feature is used, it is recommended that the value

of `_SmallFontDialog` should be set immediately prior to the call to `uOpenDialog`, as in the following example:

```
GLREF_D UWORD _SmallFontDialog;

LOCAL_C VOID RunSmallDialog(VOID)
{
    _SmallFontDialog=TRUE;
    uOpenDialog(NULL);
    ...
}
```

A call to `uOpenDialog` automatically clears `_SmallFontDialog`.

Note that `_SmallFontDialog` must not be set to a non-zero value for any dialog that is run on a Series 3 or Series 3a. Attempting to run such a dialog on either of these machines will result in the application being terminated with a panic 55.

See below for additional examples of the use of `uOpenDialog`.

uRunDialog

Run a dialog

```
INT uRunDialog(VOID);
```

Runs the current dialog, waiting until it is complete.

Returns 0 if the user cancelled, a negative value if an error such as OOM occurred (in which case the user will already have been notified of the error), or else (as for `Opl/W`):

- in the case of a dialog with action buttons, the return value is the (lower-case) keycode of the button pressed (unless that button was the Escape key, in which case the return value is zero)
- otherwise, the index of the item highlighted when the dialog is terminated, counting the first line (which is the title line if that is present) as 1.

If the dialog is completed successfully, all live variables specified by the items included in the dialog are written to, according to the values selected by the user.

The Epoc static `DatLocked` is set `TRUE` for the duration of the call to `uRunDialog`.

The call requires Window Server resources, and hence its success can never be guaranteed.

See below for examples.

Note that following a call to `uRunDialog`, any flashing cursor in the main display of the application may stop flashing. This will happen if any field in the dialog displayed a flashing cursor.

Applications which display single- or multi-line edit boxes, or which otherwise incorporate a flashing cursor, will need in general to provide a layer of the following sort around calls to `uRunDialog`:

```
LOCAL_C INT RunDialog(VOID)
{
    INT ret;

    ret=uRunDialog();
    ReassertCursor();
    return(ret);
}
```

(see also the later discussion on `HEBEmphasise`).

uAddButtonList

Add an action list to a dialog

```
INT CDECL uAddButtonList(TEXT *but,INT code,...);
```

Adds an action list of buttons to the current dialog, with each button being specified by a pair of passed parameters `but`, `code`. The text `*but` (ZTS) appears above the button and a representation of the keycode code appears inside the button.

Buttons are added from the parameters passed until a `NULL` is encountered for the `but` of a pair.

For example,

```
uAddButtonList("Cancel",W_KEY_ESCAPE,"Grey",'g',"Invert",'i',NULL);
```

Returns 0 for success, or else a negative error - in which case the user will already have been notified of the error.

Allowed values for code are any printable key (such as 'a' through 'z', or '+' or '*') as well as W_KEY_RETURN, W_KEY_ESCAPE, W_KEY_DELETE_LEFT, W_KEY_SPACE, W_KEY_UP, W_KEY_DOWN, W_KEY_RIGHT, W_KEY_LEFT, W_KEY_TAB, and W_KEY_MENU. Alphabetic values of code are always displayed in upper-case form but are returned (when the corresponding key is pressed) in lower-case form.

If a keycode for a button is specified as negative, then if the user presses ESCAPE, that button will visibly depress and the dialog will be terminated (with return value 0 and without the contents of any live variables being overwritten).

uAddChoiceList **Add a choice list to a dialog**

```
INT CDECL uAddChoiceList(TEXT *prompt,UWORD *nset,TEXT *choice,...);
```

Adds a choice list to the current dialog, with prompt *prompt and live variable nset, and choices given in text form by additional parameters until a NULL is encountered.

For example,

```
uAddChoiceList("Font",&font,"Standard","Bold","Small digits",NULL);
```

The value of nset should initially be 1 to select the first choice in the list ("Standard" in the above example), 2 to select the second choice, and so forth. This is also the form in which the choice of the user is written back to nset on successful completion of the dialog.

Returns 0 for success, or else a negative error - in which case the user will already have been notified of the error.

If prompt is passed as NULL, the choice list is displayed centred horizontally in the dialog, without any prompt.

uAddDialogItem **Add a general item to a dialog**

```
INT uAddDialogItem(INT type,TEXT *prompt,VOID *data);
```

Adds an item of specified type to the current dialog.

Returns 0 for success, or else a negative error - in which case the user will already have been notified of the error.

The item has prompt as specified, unless prompt is passed as NULL, in which case the item has no prompt, and is displayed centred horizontally in the dialog.

Possible values of type, and the corresponding structs for data, are:

H_DIALOG_TEXT	for a text item, with struct H_DI_TEXT
H_DIALOG_NUMBER	for a numeric editor, with struct H_DI_NUMBER
H_DIALOG_FLOAT	for a floating point editor, with struct H_DI_FLOAT
H_DIALOG_TIME	for a time or duration editor, with struct H_DI_TIME
H_DIALOG_DATE	for a date editor, with struct H_DI_DATE
H_DIALOG_EDIT	for a non-scrolling text editor, with struct H_DI_EDIT
H_DIALOG_SEDIT	for a scrolling text editor, with struct H_DI_SEDIT
H_DIALOG_XINPUT	for a secret data input item, with struct H_DI_XINPUT
H_DIALOG_FSEL	for a filename editor or filename selector, with struct H_DI_FSEL.

(It is also possible to use this routine to add a choice list or an action list to a dialog but in practice, the customised routines uAddChoiceList and uAddButtonList given earlier are much to be preferred.)

More details of each of the above item types are given in the following sections.

H_DIALOG_TEXT**Text items in dialogs**

```
typedef struct
{
    TEXT *str;
    UWORD type;
} H_DI_TEXT;
```

Possible bit values of type are:

H_DTEXT_ALIGN_LEFT	left align the text in its field (the default)
H_DTEXT_ALIGN_RIGHT	right align the text in its field
H_DTEXT_ALIGN_CENTRE	centre the text in its field
H_DTEXT_BOLD	display in bold
H_DTEXT_UNDERLINE	underline the item
H_DTEXT_SELECTABLE	give the item a bullet and allow it to be highlighted.

The field `str` is a BCS giving the string to display. This is displayed in the right hand column of the dialog, unless `prompt` is passed as `NULL` in the corresponding call to `uAddDialogItem`.

For example:

```
H_DI_TEXT txt;
TEXT buf[10];

if (uOpenDialog(NULL))
    return;
txt.type=H_DTEXT_ALIGN_CENTRE|H_DTEXT_BOLD|H_DTEXT_UNDERLINE;
txt.str=&buf[0];
uZTStoBCS(txt.str,"Warning");
if (uAddDialogItem(H_DIALOG_TEXT,NULL,&txt))
    return;
...
uRunDialog();
...
```

H_DIALOG_NUMBER**Numeric editors in dialogs**

```
typedef struct
{
    LONG *value;
    LONG low;
    LONG high;
} H_DI_NUMBER;
```

The passed value of the live variable `*value` is what is initially displayed in the dialog. The user is constrained from changing the value beyond the limits `low` and `high`.

Setting `value` equal to the address of a 2-byte integer, instead of a 4-byte long integer, would be a severe error.

H_DIALOG_FLOAT**Floating point editors in dialogs**

```
typedef struct
{
    DOUBLE *value;
    DOUBLE low;
    DOUBLE high;
} H_DI_FLOAT;
```

The meanings of the fields are as for `H_DI_NUMBER`.

H_DIALOG_TIME**Time editors in dialogs**

```
typedef struct
{
    ULONG *value;
    ULONG low;
    ULONG high;
    UWORD type;
} H_DI_TIME;
```

Possible bit values for type are:

H_DTIME_SHOW_SECONDS	the time display is to include seconds (which are suppressed by default)
H_DTIME_DURATION	the time being edited is a duration, not an absolute time, and as such it never makes sense to display (eg) an <i>am</i> or <i>pm</i> alongside it.

The meanings of the other fields in the H_DI_TIME struct are as for H_DI_NUMBER. Note that all times are expressed in seconds since midnight, regardless of the setting of the bit H_DTIME_SHOW_SECONDS.

H_DIALOG_DATE**Date editors in dialogs**

```
typedef struct
{
    ULONG *value;
    ULONG low;
    ULONG high;
} H_DI_DATE;
```

The meanings of the fields are as for H_DI_NUMBER.

The dates are all expressed in days since 1900.

Useful date constants

The following constants, defined in *hwif.h*, may prove useful:

H_LAST_DAY	the largest legal value for any of the three fields (when the P_DATE representation of date expires)
H_FIRST_SYS_DAY	the smallest value of day number that can be converted into the system-time representation of date (ie 1st January 1970)
H_LAST_SYS_DAY	the largest value of day number that can be converted into the system-time representation of date.

(No value is defined for what would have been H_FIRST_DAY, since this is just zero.)

H_DIALOG_EDIT**Non-scrolling text editors in dialogs**

```
typedef struct
{
    TEXT *str;
    UWORD len;
} H_DI_EDIT;
```

The field *len* gives the maximum allowed length of the string. This also determines the width set aside for the display of the item in the dialog.

The live variable *str* gives the initial contents of the string, in BCS form. The string, once edited, is also written back in BCS form.

H_DIALOG_SEDIT**Scrolling text editors in dialogs**

```
typedef struct
{
    TEXT *str;
    UWORD len;
    UWORD width;
} H_DI_SEDIT;
```

The meaning of the fields is as for H_DI_EDIT, except that the width set aside for display purposes is given by width full character widths.

H_DIALOG_XINPUT**Secret data input items in dialogs**

```
typedef struct
{
    TEXT *str;
} H_DI_XINPUT;
```

The field str must point to a buffer long enough to hold a BCS with eight characters.

H_DIALOG_FSEL**Filename selectors and filename editors**

```
typedef struct
{
    TEXT *fname;
    UWORD flags;
} H_DI_FSEL;
```

The live variable fname must point to a buffer of at least 128 characters. This is used to seed the file selector and also to receive the filename chosen.

If the bit H_FILE_NEW_EDITOR is set in flags, a filename editor is produced, with behaviour governed by the following remaining bits in flags:

H_FILE_ALLOW_DIRS	allow the user to choose a directory name
H_FILE_JUST_DIRS	<i>force</i> the user to choose a directory name
H_FILE_FORCE_NXIST	force the user to choose the name of a file that doesn't already exist
H_FILE_NO_AUTOQUERY	disable the "Confirm overwrite?" dialog that appears, by default, when the user types the name of a file that already exists
H_FILE_ACCEPT_NULL	allow the user to leave the field blank
H_FILE_SET_DEFEEXT	set the default extension from the seed name passed (so that if the seed is <i>first.pic</i> and the user types <i>second</i> , the filename returned to the program is <i>second.pic</i>)
H_FILE_CAN_WILDCARD	allow the user to type in wildcards (such as *.pic)

If the bit H_FILE_NEW_EDITOR is *not* set (there is a #define H_FILE_PICK_SELECTOR equal to zero), a filename selector is produced, with behaviour governed by the following remaining bits in flags:

H_FILE_ALLOW_DIRS	allow the user to choose a directory name
H_FILE_JUST_DIRS	<i>force</i> the user to choose a directory name
H_FILE_RESTRICT_LIST	restrict the set of files initially selectable (ie until the user presses TAB) to those with extension matching that of the seed filename passed
H_FILE_ACCEPT_NULL	allow the user to leave the field in a state in which no filename is selected - displaying, for example, (<i>no files</i>).
H_FILE_SET_DEFEEXT	set the default extension from the seed name passed (so that if the seed is <i>first.pic</i> and the user selects <i>second</i> , the filename returned to the program is <i>second.pic</i>)
H_FILE_CAN_WILDCARD	allow the user to specify wildcards (such as *.pic) - by means of the CONTROL+TAB, CONTROL+ENTER mechanism

uBeginDCL**Begin a dynamic choice list**

```
INT uBeginDCL(H_DI_CHOICE *pch);
```

Prepares to add a dynamically-defined choice list to a dialog.

The routine writes into the passed `H_DI_CHOICE` struct, which is declared and provided by the caller. The address of this same struct should be passed to subsequent calls to `uGrowDCL` and `uAddDCL`.

Returns 0 for success, or else a negative error - in which case the user will already have been notified of the error.

See below for an example.

uGrowDCL**Add a choice to a dynamic choice list**

```
INT uGrowDCL(H_DI_CHOICE *pch, TEXT *choice);
```

Adds an entry whose text is given by `*pchoice` to the end of the dynamically-defined choice list identified by `*pch`.

Returns 0 for success, or else a negative error - in which case the user will already have been notified of the error.

See below for an example.

uAddDCL**Add a dynamic choice list to a dialog**

```
INT uAddDCL(TEXT *prompt, UWORD *nset, H_DI_CHOICE *pch);
```

Adds a dynamically defined choice list to the current dialog, with prompt `*prompt` and live variable `nset`, and choices defined by `*pch`.

Returns 0 for success, or else a negative error - in which case the user will already have been notified of the error.

For example, the following routine builds up a choice list whose contents are the twelve month names:

```
LOCAL_C INT AddMonthChoiceList(UWORD *pmonno)
{
    H_DI_CHOICE ch;
    TEXT mon[32];
    INT i;

    if (uBeginDCL(&ch))
        return(-1);          /* report failure to caller */
    for (i=0; i<12; i++)
    {
        p_nmmon(&mon[0], i);
        if (uGrowDCL(&ch, &mon[0]))
            return(-1);      /* report failure to caller */
    }
    return(uAddDCL("Month", pmonno, &ch));
}
```

No access should be made to the contents of `*pch` after adding it into a dialog.

The calls `uBeginDCL` and `uGrowDCL` each allocate memory which is added into the central control block of the current dialog only when a subsequent call `uAddDCL` is made. If no such call is made, the memory will remain permanently tied up. (However, if any call `uGrowDCL` fails, this memory is automatically freed before reporting back the error.)

uAddGreyUline**Underline menu items with a grey line**

```
VOID uAddGreyUline(UBYTE accel, UWORD *plines);
```

Built in applications have the ability to add grey lines underneath menu items. This function allows Hwif programs to do the same.

Grey underlining serves to group related menu items and can be a useful visual aid if used sparingly.

To use this function, the parameter `accel` must contain the accelerator character corresponding to the menu item under which a grey underline is to be placed. The accelerator character is the first character in each entry of the table pointed to by `_cmds` (see the section on Menu bar interactions in the *Introduction to Hwif* chapter in this manual).

The parameter `plines` must point to a memory location containing 16 words (256 bits) provided by the application; this area should be initialised to zero.

Further, the global variable `_GreyLines` should be declared in the application and should contain the address of this area before the menu bar is displayed

The function is implemented as follows:

```
GLDEF_C VOID uAddGreyUline(UBYTE accel,UWORD *plines)
{
    *(plines+(accel/16)) |= (1<<(accel%16));
}
```

It merely sets a bit corresponding to the value of the accelerator character in the 16 word area provided by the application.

Some points should be noted:

- This function cannot be used if running on the Series 3 or in compatibility mode on the Series 3a.
- Menu items can be safely underlined in grey even if grey is not enabled for the main console window.

The following code is an example of the use this function.

```
#include <p_std.h>
#include <wlib.h>
#include <hwif.h>

GLREF_D UWORD _UseFullScreen;
GLREF_D UWORD *_GreyLines;

LOCAL_D WORD   keystat;
LOCAL_D INT    gc;
LOCAL_D UWORD  lines[16];

LOCAL_D TEXT *cmds[] =
{
    "nNew file",
    "oOpen file",
    "aSave as",
    "sSave",
    "iInsert",
    "cCopy",
    "dDelete",
    "gChange group",
    "tChange type",
    "bChange subentry",
    "pSet preferences",
    "xExit",
    NULL
};

GLDEF_D TEXT **_cmds = &cmds[0];

LOCAL_D H_MENU_DATA mdata[] =
{
    "File",4,
    "Edit",3,
    "Changes",3,
    "Special",2,
    NULL
};
```

```
GLDEF_D H_MENU_DATA *_mdata = &mdata[0];

LOCAL_C VOID ManageCommand(INT index)
{
    switch(index)
    {
        case 0 :
            break;
        case 11 :
            p_exit(0);
        default:
            break;
    }
}

LOCAL_C VOID SpecificInit(VOID)
{
    _GreyLines = &lines[0];
    p_bfil(&lines[0], sizeof(lines), 0);
    uAddGreyUline('o', &lines[0]);
    uAddGreyUline('g', &lines[0]);
    uAddGreyUline('t', &lines[0]);

    if (_UseFullScreen)
    {
        uEnableGrey();
        gc = gCreateGCO(uFindMainWid());
        gBorder2(W_BORDER_TYPE_1, W_BORD_CORNER_4 |
                W_BORD_SHADOW_ON | W_BORD_SHADOW_D);
    }
    else
    {
        gc = gCreateGCO(uFindMainWid());
        gBorder(W_BORD_CORNER_4 | W_BORD_SHADOW_ON |
                W_BORD_SHADOW_D);
    }
}

LOCAL_C VOID TryExecuteCommand(INT keycode)
{
    keycode = uLocateCommand(keycode);
    if (keycode >= 0)
        ManageCommand(keycode);
}

LOCAL_C VOID MainLoop(VOID)
{
    WMSG_KEY key;
    INT      ret;

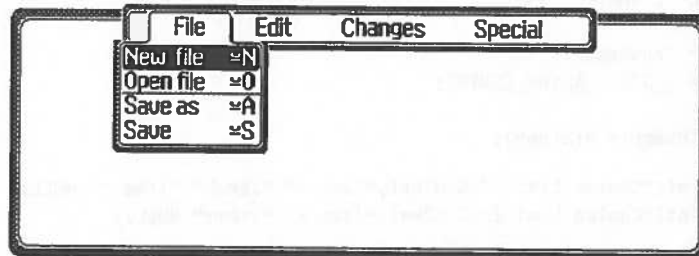
    FOREVER
    {
        uGetKeyA(&keystat, &key);
        p_iowait();
        if (key.keycode & W_SPECIAL_KEY)
        {
            key.keycode &= (~W_SPECIAL_KEY);
            if (key.keycode == 'x')
                p_exit(0);
            TryExecuteCommand(key.keycode);
        }
        else if (key.keycode == W_KEY_MENU)
        {
            ret = uPresentMenus();
            if (ret > 0)
                TryExecuteCommand(ret);
        }
    }
}
```

```

GLDEF_C VOID main(VOID)
{
    _UseFullScreen = TRUE;
    uCommonInit();
    SpecificInit();
    MainLoop();
}

```

Note the use of `_GreyLines` and the implementation of grey underlining inserted at the beginning of `SpecificInit()`. The effect is as illustrated below:



uSetDialogUline

Underline dialog components

```
VOID uSetDialogUline(INT pos,INT on);
```

Built in applications have the ability to add or remove a solid underline to components in a dialog. This function allows Hwif programs to do the same.

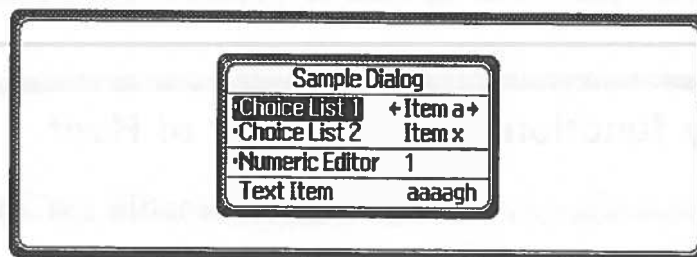
Underlining serves to group related dialog components and can be a useful visual aid if used sparingly.

The parameter `pos` specifies the number of the dialog component under which a line is to be inserted or removed. A zero value refers to the title line while a value of one refers to the immediately following dialog component, and so on. The value of `pos` must lie in the inclusive range from zero to one less than the number of lines in the dialog (including the title line), otherwise the results are unpredictable.

The parameter `on` specifies either a zero or a non-zero value; a non-zero value means that an underline is to be inserted while a zero value means that any underline is to be removed.

The function is usually called before calling `uRunDialog`.

By taking the example given in the description of `uAddGreyUline` earlier and by adding the `RunSampleDialog` function and modifying the `ManageCommand` function as shown below, the following dialog display results.



```

#define ON 1
#define OFF 0

LOCAL_C VOID ManageCommand(INT index)
{
    switch(index)
    {
        case 7 :
            RunSampleDialog();
            break;
        case 11 :
            p_exit(0);
        default:
            break;
    }
}

```

```

LOCAL_C VOID RunSampleDialog(VOID)
{
    UWORD ch1sel = 1;
    UWORD ch2sel = 1;
    LONG nEditorValue = 1L;
    H_DI_NUMBER nEditor;
    H_DI_TEXT tItem;

    nEditor.value = &nEditorValue;
    nEditor.low = 0;
    nEditor.high = 100;

    tItem.str = "aaaaagh";
    tItem.type = H_DTEXT_ALIGN_CENTRE;

    uOpenDialog("Sample Dialog");

    uAddChoiceList("Choice List 1",&ch1sel,"Item a","Item b","Item c",NULL);
    uAddChoiceList("Choice List 2",&ch2sel,"Item x","Item y",NULL);

    uAddDialogItem(H_DIALOG_NUMBER,"NumericEditor",&nEditor);
    uAddDialogItem(H_DIALOG_TEXT,"Text Item",&tItem);

    uSetDialogUline(2,ON);
    uSetDialogUline(3,ON);

    uRunDialog();
}

```

The above illustration was produced on a Series 3a in non-compatibility mode.

Underlining in dialogs

By default, the title of a dialog is always underlined on all machines.

There are, however, the following differences in behaviour between the different machine types:

- on the Series 3a and the *Workabout*, in both compatibility and non-compatibility mode, any number of dialog components can be underlined.
- on the Series 3a and the *Workabout*, in both compatibility and non-compatibility mode, if the title underline is to be removed then it must be done explicitly calling `uSetDialogUline(0,0)`.
- on the Series 3, only one underline is permitted in a dialog; therefore, inserting an underline under a component other than the title causes the title underline itself to be removed.

The auxiliary functions in the u-layer of Hwif

uEscape

Disable/enable Escape processing

```
VOID uEscape(UWORD flag);
```

If `flag` is `FALSE`, prevents the application from being automatically terminated if the user presses `PSION+ESCAPE`. Otherwise, enables this behaviour (which is the default).

For example,

```

LOCAL_C VOID SpecificInit(VOID)
{
    uEscape(FALSE);
    MainWid=uFindMainWid();
    CreateGC();
    hCrackCommandLine();
    SmallScreen();
    OpenEditors();
    DrawBorders();
    wsEnable();
}

```

uFindMainWid**Find ID of main window**

```
UINT uFindMainWid(VOID);
```

Returns the ID of the graphics window opened by the Console. This is required as a parameter to many Wlib calls, such as `gCreateGC` and `wSetWindow`.

See above for an example.

uForceToFront**Force the application into foreground**

```
VOID uForceToFront(VOID);
```

Of use mainly when an error has arisen that must be notified to the user, when the application might be in background (eg processing a *Shutdown* message from the System Screen).

uErrorString**Fail-safe presentation of an error string**

```
VOID uErrorString(TEXT *message);
```

Presents an alert consisting of the message passed, in a manner guaranteed not to fail with OOM.

While the alert is on screen, the application appears as "Busy" in the System Screen.

The application is forced into foreground (if not already there).

uErrorValue**Fail-safe presentation of an error message**

```
INT uErrorValue(INT ret);
```

Presents a failsafe alert, similar to that presented by `uErrorString`, except that:

- nothing happens if `ret` is zero
- otherwise, the message displayed is the system error text for the error number passed as `ret`
- in all cases, the value `ret` is returned from the call itself.

This routine is more useful than might at first be thought.

For example,

```
LOCAL_C INT DoMerge(TEXT *pb,INT flags,INT dir)
{
    UINT state;

    state=DbfStateDisabled;
    return(uErrorValue(DbfcopyFile(&state,dH,pb,flags,1,dir)));
}
```

or

```
LOCAL_C VOID ReduceScreenSize(VOID)
{
    W_WINDATA wd;

    wd.extent.tl.x=wd.extent.tl.y=0;
    wd.extent.width=189;
    wd.extent.height=80;
    wSetWindow(MainWid,W_WIN_EXTENT,&wd);
    if (uErrorValue(wCheckPoint()))
        p_exit(0);
}
```

uCheckHandle**Check that a handle is non-zero**

```
VOID *uCheckHandle(VOID *handle);
```

If `handle` is zero, presents an appropriate error message. Otherwise does nothing.

In either case, returns `handle`.

For example (to give the source code for `uBeginDCL`):

```
GLDEF_C INT uBeginDCL(H_DI_CHOICE *pch)
{
    pch->u.count=1;
    pch->menu=uCheckHandle(hChoiceOpen());
    return(!(INT)pch->menu);
}
```

uZTStoBCS

Convert a ZTS to a BCS

```
VOID uZTStoBCS(TEXT *bcs, TEXT *zts);
```

Writes the length of `*zts` (excluding the terminating zero) to `*bcs` and copies `*zts` (again excluding the zero) to `*(bcs+1)`.

For example, to produce some text centred in the top line of a dialog, but without the usual underline:

```
H_DI_TEXT txt;
TEXT buf[20];

if (uOpenDialog(NULL))
    return;
uZTStoBCS(&buf[0], "Time is now");
txt.type=H_DTEXT_ALIGN_CENTRE;
if (uAddDialogItem(H_DIALOG_TEXT, NULL, &txt))
    return;
...
```

uDialogMenu

Present a menu-like dialog

```
INT CDECL uDialogMenu(TEXT *title, TEXT *pb, ...);
```

Presents a dialog of text items that functions as a sort of menu, for example as the first level in a Help dialog suite.

The dialog has `*title` in its title line, and then subsequent choices, determined by following parameters `pb, ...`, until a `NULL` is encountered. The choices are each selectable.

Following the standard rules for a dialog, returns 0 if the user cancelled or a negative value if an error occurred (such as OOM) - in which case the user will already have been notified of this fact - or else the index of the item chosen.

For example:

```
choice=uDialogMenu("Help on which topic",
    "Cursor position",
    "Select regions",
    "Drawing lines",
    "Brushes",
    "Using clipboards",
    "Borders and frames", NULL);
switch (choice)
    ...
```

uDisplayText

Present a dialog containing textual information

```
VOID CDECL uDisplayText(TEXT *pb, ...);
```

Presents a dialog with no title, every line of which is left-aligned text - for example, as a subsidiary dialog in a Help dialog suite.

Lines are added to the dialog until a `NULL` is encountered in the argument list.

For example:

```
uDisplayText("To move the cursor:",
            "use the usual cursor keys",
            "including Home, End, PgUp and PgDn",
            "or use 'Goto' in the 'Edit' menu.",
            "Use Enter or Space to toggle the pixel",
            "at the cursor position.",NULL);
```

Time-text utility functions

The Hwif time-text functions provide a convenient way of generating textual representations of times and/or dates that reflect the user's preferences as given in the Formats dialog in the *Time* application.

hTTOpen Open a time-text channel

```
VOID *hTTOpen(VOID);
```

Opens a channel for use by subsequent time-text utility functions. Returns NULL on error (OOM) - in which case the user will already have been notified - or otherwise a handle to be used in subsequent hTTxxx calls.

See below for an example.

It is possible to have more than one time-text channel open at any one time. Two channels might differ as regards the formats specified for them, or times set into them.

hTTSetAbbreviations Set time-text abbreviation lengths

```
VOID hTTSetAbbreviations(VOID *handle,INT dayabb,INT monabb);
```

Configures the specified time-text channel so that any day names rendered by it are abbreviated to dayabb characters, and any month names rendered are abbreviated to monabb characters.

To abbreviate (eg) day names but not month names, give monabb a value larger than any expected month name - 40, for example.

See below for an example.

hTTSetFormat Set time-text format

```
VOID hTTSetFormat(VOID *handle,INT values);
```

Configures the specified time-text channel so that any text rendered by it subsequently conforms to the bits present in values as follows (all the bits affect any date strings produced, except the last, which affects any time strings produced):

H_TIME_FORMAT_NO_DAY	Omit the number of the day in the month
H_TIME_FORMAT_NO_MONTH	Omit the number of the month in the year
H_TIME_FORMAT_NO_YEAR	Omit the year number
H_TIME_FORMAT_MONTH_NAME	Include the name of the month
H_TIME_FORMAT_SUFFIX	Include a suffix after any day number
H_TIME_FORMAT_DAY_NAME	Include the name of the day
H_TIME_FORMAT_NO_CENTURY	Include the century in any year
H_TIME_FORMAT_NO_SECONDS	Include seconds

In all cases, the sense of the bit indicates what the default is. For example, the century is given unless the bit H_TIME_FORMAT_NO_CENTURY is set, whereas no month name is given unless the bit H_TIME_FORMAT_MONTH_NAME is set.

At the same time as a call to `HTTSetFormat` is made, the time-text channel records the user's preferences, as indicated via the Formats dialog in the *Time* application, for the following components of any textual representation of date and time:

- whether to use 12 or 24 hour clock
- whether the day name comes before or after the month name
- what time and date separators to use.

A second call to `HTTSetFormat` completely wipes out the effect of any previous such call on the same channel. (However, it has no effect on the result of a previous call to `HTTSetTime` or `HTTSetAbbreviations`.)

See below for an example.

HTTSetTime Set time-text time and date

```
INT HTTSetTime(VOID *handle, INT type, VOID *data);
```

Sets the time and date to be represented in the next string produced from the specified time-text channel.

The time and date can be given in any of four ways, depending on the value of `type` passed:

<code>H_TIME_SET_SDATE</code>	data points to a <code>ULONG</code> giving the time and date in system format
<code>H_TIME_SET_DATE</code>	data points to a <code>P_DATE</code> representation of time and date
<code>H_TIME_SET_DAYSEC</code>	data points to a <code>P_DAYSEC</code> representation of time and date
<code>H_TIME_SET_NOW</code>	the value of data is ignored and the time and date are set from the current system time.

Returns zero unless an illegal value was passed (which is a programming error), causing underflow or overflow.

See below for an example.

HTTSenseString Sense string from time-text channel

```
INT HTTSenseString(VOID *handle, INT type, TEXT *buf);
```

Writes a string of the requested type from the specified time-text channel into the passed buffer.

The types of string that can be requested are:

<code>H_TIME_SENSE_TIME</code>	just write the time
<code>H_TIME_SENSE_DATE</code>	just write the date
<code>H_TIME_SENSE_BOTH</code>	write both the time and the date.

The application must ensure that `buf` points to a sufficiently long buffer to receive the text written.

No terminating zero is written; instead, the length of the string produced is returned from the `HTTSenseString` call.

For example, the following code

```
VOID *tth;
TEXT buf[48];

tth=HTTOpen();
HTTSetFormat(tth, H_TIME_FORMAT_DAY_NAME|H_TIME_FORMAT_MONTH_NAME);
HTTSetAbbreviations(tth, 3, 40);
HTTSetTime(tth, H_TIME_SET_NOW, NULL);
buf[HTTSenseString(tth, H_TIME_SENSE_DATE, &buf[0])]=0;
HTTClose(tth);
```

would result in today's date being written as a ZTS into `buf[]`, in the form "Wed, 15 January 1992" on an English language Series 3 with default settings.

Note that the above routine omits to check the value of `tth` after the call to `HTTOpen`. This would be permissible for a call to `HTTOpen` during the initialisation of an application (provided its start-up heap was properly calibrated), but (possibly) not during its steady-state phase.

hTTClose**Close a time-text channel**

```
VOID hTTClose(VOID *handle);
```

Frees the resources allocated for the specified time-text channel.

It is unnecessary to make this call if the time-text channel is used only for some of the lifetime of an application. Any application which uses a time-text channel throughout its lifetime has no need to close the channel down prior to exiting.

See above for an example.

Stand alone date/time text editor functions

These are a set of functions which construct and manipulate stand-alone date/time editors. As such, they need not exist within a dialog. In many respects they are similar to the time-text functions but have fewer date/time formats.

hDTOpen**Create a date/time text editor**

```
VOID *hDTOpen(H_DTEDIT *dte);
```

Creates a stand alone date/time text editor for use by subsequent date/time editor functions. Returns NULL on error (OOM) - in which case the user will already have been notified. On successful completion, it returns a handle to be used in subsequent hDTxxx calls.

A pointer to a struct of type H_DTEDIT must be supplied. This enables initialisation information to be supplied to the function. For example, the position of the date/time text editor within a window can be specified.

The format of H_DTEDIT is shown below.

It is possible to have more than one date/time text editor open at any one time. Two editors might differ in their formats or the date/time(s) set into them.

hDTClose**Close date/time text editor**

```
VOID hDTClose(VOID *dte);
```

Closes the stand alone date/time text editor and frees any resources used. The handle of the editor must be passed to this function.

hDTSet**Set value in date/time text editor**

```
VOID hDTSet(VOID *dte, H_SE_DTEDIT *sdte);
```

Sets a value in the date/time text editor. dte contains the handle of the text editor and the source for the value to be set is found in a struct of type H_SE_DTEDIT pointed to by sdte.

The format of struct H_SE_DTEDIT is as shown below.

hDTSense**Retrieve value from date/time text editor**

```
VOID hDTSense(VOID *dte, H_SE_DTEDIT *sdte)
```

Retrieves the current information held by the date/time text editor. dte contains the handle of the text editor and the retrieved information is placed into a struct of type H_SE_DTEDIT pointed to by sdte.

The format of struct H_SE_DTEDIT is as shown below.

hDTSelfCheck**Date/time text editor self-check**

```
INT hDTSelfCheck(VOID *dte);
```

The date/time text editor performs a validation of the value it currently holds and will return a non-zero value if the check fails. The handle of the text editor is contained in dte.

hDTHandleKey Pass key to date/time text editor

```
INT hDTHandleKey(VOID *dte, INT keycode, INT modifier)
```

Delivers the specified keypress to the date/time text editor whose handle is contained in *dte*.

The meaning of the return value has little significance for Hwif programming (date/time editors, once initialised, can never subsequently fail with OOM).

hDTEmphasise Emphasise date/time text editor

```
VOID hDTEmphasis(VOID *dte, INT flag);
```

Changes the emphasis for the date/time text editor. If *flag* is TRUE, the text editor is highlighted (and should gain the keyboard focus). If *flag* is FALSE, any highlighting is removed from the text editor.

The handle of the text editor is contained in *dte*.

H_DTEDIT hDTxxxstructs

```
typedef struct
```

```
{
    UWORD flags;
    LONG value;
    LONG low;
    LONG high;
    P_POINT pos;
    UWORD win;
    UWORD width;
    UWORD emph;
} H_DTEDIT
```

Possible values for flags are:

H_DTEDIT_DDMYYYYY

date/time text editor has Date format in the shape: DD/MM/YYYY; for example: 01/01/1993

H_DTEDIT_HHMMSS

date/time text editor has Time format in the shape: HH:MM:SS. Depending on the system settings, HH can be in 24 hour or 12 hour format. If in 12 hour format then the time is followed by am or pm as appropriate; for example: 02:20:30 pm or 14:20:30

H_DTEDIT_HHMM

date/time text editor has Time format in the shape: HH:MM. Depending on the system settings, HH can be in 24 hour or 12 hour format. If in 12 hour format then the time is followed by am or pm as appropriate; for example: 02:20 pm or 14:20

H_DTEDIT_HHMMSS_D

date/time text editor represents a time duration in the shape HH:MM:SS

H_DTEDIT_HHMM_D

date/time text editor represents a time duration in the shape HH:MM

H_DTEDIT_SET_VALUE

If this flag is set, then the current date/time in the editor is set to value, otherwise it is set to the default (the current date or time)

H_DTEDIT_SET_LOW

If this flag is set, then the minimum date/time in the editor is set to low, otherwise it is set to the default

H_DTEDIT_SET_HIGH

If this flag is set, then the maximum date/time in the editor is set to high, otherwise it is set to the default

H_SE_DTEDIT**hDTxxxstructs**

```
typedef struct
{
    UWORD flags;
    LONG value;
    LONG low;
    LONG high;
} H_SE_DTEDIT
```

Possible values for flags are the H_DTEDIT_SET_ flags explained above.

Edit box functions**hEBOpen****Open an edit box**

```
VOID *hEBOpen(INT flags, H_EDIT_BOX *heb);
```

Opens an edit box, as specified by flags and by *heb.

Returns the handle of the edit box, if successful, or else NULL (in which case the user will already have been informed of the failure). The handle should be used to identify this edit box, as opposed to others an application may have, in subsequent hEBxxx calls.

The H_EDIT_BOX struct is defined as follows:

```
typedef struct
{
    UWORD maxchars;
    UWORD vulen;
    UWORD vislines;
    P_POINT pos;
    UWORD win;
    UWORD font;
    UWORD style;
    UWORD lead_tot;
    UWORD lead_top;
} H_EDIT_BOX;
```

It is not necessary for all the fields in this struct to be filled in before a call to hEBOpen is made. Default values are supplied for some of the fields. These defaults are overridden by the supplied values only if corresponding bits are set in flags.

Possible bits set in flags are as follows:

H_EDIT_BOX_CLIPBOARD

The edit box is to allocate extra resources so that it can perform the clipboard functions Paste and Copy (with deleted text of size more than one character automatically being cut into the clipboard, as standard for Series 3 editors)

H_EDIT_BOX_LEFT_CURSOR

A triangular pointing cursor is to be displayed down the left hand side of the edit box, opposite the line with the flashing cursor

H_EDIT_BOX_FONT

The font and font style for the edit box are to be as specified in the fields font and style in *heb (the defaults are the system font with normal style); possible values might be W_FONT_BASE+1 for the bold font, and G_STY_ITALIC for an italic style

H_EDIT_BOX_LEADING

The leading for the edit box font is to be as specified in the fields lead_tot and lead_top in *heb (the defaults being 2 and 1 respectively): the former being the total extra vertical spacing between lines, in addition to the font height, and the latter being how much of the total leading applies at the top of each line.

H_EDIT_BOX_VISLINES

The edit box is to be heb->vislines lines high (the default is one line) - although in all cases, it will scroll vertically if enough text is added to form more lines than can be seen at once.

The meanings of the remaining fields in the `H_EDIT_BOX` struct, which always have to be filled in by the caller, are as follows:

<code>maxchars</code>	The maximum number of characters in total that the edit box can contain (before beeping at the user and displaying a <i>Maximum number of characters reached</i> information message); paragraph ends count as one character each
<code>vulen</code>	The total visible width of the edit box, in pixels, including any margin required for a left cursor; this width also implicitly defines the wrapping margin for multi-line edit boxes
<code>win</code>	The ID of the enclosing window - which is usually the value <code>MainWid</code> returned by a call to <code>uFindMainWid</code>
<code>pos</code>	The offset of the top left of the edit box, relative to the enclosing window.

The edit box may be further customised, before any keys are passed on to it, by means of many of the calls discussed below.

hEBHandleKey

Pass a key to an edit box

```
INT hEBHandleKey(VOID *ebH, INT keycode, INT modifier);
```

Delivers the specified keypress to the edit box. This should be either a printable character or an editing key. In practice, ENTER keys should only be allowed through to multi-line editors.

Returns zero for success, or a negative value for an error (in which case the user will already have been notified).

Formatting in background

Keys which cause a change in the location of line breaks due to word-wrap are treated slightly differently in Hwif editors than in editors used by the built-in applications:

- in the built-in applications, only the line containing the cursor is reformatted and redrawn at once, before the edit box checks to see if another keypress is ready to be processed; lines further from the cursor are reformatted and redrawn, if needed, *as a background activity* in pauses between the receipt of incoming keys
- in Hwif editors, any subsequent key is processed only when the edit box has been completely reformatted and redrawn.

The difference in performance only becomes apparent for larger editors with longer paragraphs. For programmers wishing to increase the responsiveness of their editors to incoming keys, the following lines of code may be tried:

```
GLREF_D UWORD _ebControlFormat;

_ebControlFormat=TRUE;
```

In this case, Hwif editors will, for any one call to `hEBHandleKey`, only reformat and redraw one line (except on receipt of a cursor key). This means it is the responsibility of the programmer to make a later call to `hEBCompleteFormat` (discussed below) at a suitable moment - for example, when a call to `uKeyPressOutstanding` next returns FALSE.

hEBCompleteFormat

Complete edit box formatting

```
INT hEBCompleteFormat(VOID *ebH);
```

Ensures that the specified edit box is completely formatted and completely drawn. Does nothing if the formatting is already up to date.

Only needs to be called explicitly by an application if the static variable `_ebControlFormat` has been set TRUE (see above).

Returns zero for success, or a negative value if there was insufficient memory to complete formatting.

hEBSenseText**Sense the text in an edit box**

```
TEXT *hEBSenseText(VOID *ebH);
```

Returns a pointer to the buffer where the edit box is currently keeping its own copy of its text.

This copy is always zero terminated, so its length can be obtained by a call to `p_slen`.

The buffer may contain embedded `\n`'s, representing paragraph ends.

Note that the location of the buffer may change as more text is added into the edit box, so there is no point in an application trying to keep a permanent copy of this address.

hEBSetText**Set the text in an edit box**

```
INT hEBSetText(VOID *ebH, TEXT *pb, INT blen);
```

Sets the `blen` characters at `*pb` as the text for the specified edit box.

Returns zero for success, or a negative value if an error occurred (in which case the user will already have been notified).

hEBEmphasise**Emphasise an edit box**

```
VOID hEBEmphasise(VOID *ebH, INT flag);
```

Either emphasises or de-emphasises the specified edit box, depending on the value of `flag` (TRUE to emphasise it, FALSE to de-emphasise it).

An emphasised edit box displays:

- a flashing text cursor (unless the width of the text cursor has been set to zero)
- a triangular cursor in the left margin (if the flag `H_EDIT_BOX_LEFT_CURSOR` was set on initialisation)
- a highlight between the cursor and anchor points of any select region.

A de-emphasised edit box displays none of these features.

By default, an edit box is de-emphasised.

Applications may wish to surround edit boxes with curved borders, with varying degrees of shadowing to help indicate whether or not each edit box is currently emphasised. In this case, the application has the responsibility for drawing the borders (using the Wlib call `gBorderRect`).

When edit boxes lose their cursor

Note that edit boxes maintain their own record of whether they are emphasised, *and do nothing* if they receive an `hEBEmphasise` call instructing them to change their emphasis state to what it already is. Thus in the following sequence of code

```
hEBEmphasise(ebH, TRUE);
...
hEBEmphasise(ebH, TRUE);
```

the later `hEBEmphasise` call is ignored, unless a call `hEBEmphasise(ebH, FALSE)` has been made in the meantime.

The significance of this is as follows: suppose that, in the meantime, a flashing text cursor is drawn elsewhere on the screen by the application. This may be as a result of an explicit call to `wDrawTextCursor`; more likely, it will be as a side effect of a call to `uRunDialog` or `hPrintSetupDialog`. In this case, the text cursor will be removed from the edit box - since there can only be one text cursor per application at any one time. When the intervening text cursor is cancelled - say as the result of the termination of the dialog - it is the responsibility of the application to re-activate the flashing cursor where it used to be (if that is still appropriate). However, in the light of what has just been explained, a simple call to `hEBEmphasise` will be insufficient to effect this.

Accordingly, applications which contain an edit box as part of their main display, and which invoke dialogs with items which can also display a flashing cursor, need to use a layer of the following sort around calls to `uRunDialog`:

```

LOCAL_C INT RunDialog(VOID)
{
    INT ret;

    hEBEmphasise(edit[emph], FALSE);
    ret=uRunDialog();
    hEBEmphasise(edit[emph], TRUE);
    return(ret);
}

```

A similar protective layer is needed around any calls to `hPrintSetupDialog`.

hEBSetSelect

Set edit box select and cursor

```
INT hEBSetSelect(VOID *ebH, INT aoff, INT coff);
```

Sets the anchor point and cursor point of the specified edit box. Both positions are given as character offsets into the content of the edit box, starting at 0 for the position in front of the first character.

Note that neither the anchor point nor the cursor point is permitted to come after the last character in the edit box.

To set the cursor position without setting any highlighted select region, pass the value of `aoff` to be equal to that of `coff`.

Returns zero for success or a negative error value (in which case the user will already have been notified).

hEBSenseSelect

Sense edit box select and cursor

```
INT hEBSenseSelect(VOID *ebH, UWORD *top);
```

Returns the length of the select region of the specified edit box, and writes the character offset of the top of the select region to `*top`.

If there is no select region, the value 0 is returned, and the character offset of the cursor point is written to `*top`.

hEBSenseClipText

Sense edit box clipboard contents

```
TEXT *hEBSenseClipText(VOID *ebH);
```

Returns a pointer to the buffer where the clipboard of the edit box is currently keeping its own copy of its text.

The form of this buffer is exactly the same as the buffer used for the main text of the edit box (see the discussion on `hEBSenseText` above).

hEBSetClipText

Set edit box clipboard contents

```
INT hEBSetClipText(VOID *ebH, TEXT *pb, INT blen);
```

Sets the `blen` characters at `*pb` as the text for the clipboard of the specified edit box.

Returns zero for success, or a negative value if an error occurred (in which case the user will already have been notified).

hEBChangeWidth

Change width of edit box

```
INT hEBChangeWidth(VOID *ebH, INT width);
```

Changes the width of the specified edit box to `width`. The new width is specified in pixels and includes an allowance for any left margin required to display a left triangular cursor - exactly as for the `vulen` field of the `H_EDIT_BOX` struct used to initialise the edit box.

In the case of a multi-line editor, the text is automatically re-formatted, wrapping to the new width. In all cases, the display is scrolled (vertically and/or horizontally) to expose the cursor position.

Returns zero for success, or a negative value if an error occurred (in which case the user will already have been notified).

Even if the return value is negative (indicating a failure to re-format the edit box to its new width), the record within the edit box of its width will be updated as requested.

hEBSetCWidth

Set width of edit box cursor

```
VOID hEBSetCWidth(VOID *ebH, INT cwidth);
```

Changes the width of any flashing cursor displayed. The value of `cwidth` is in pixels.

By default, the flashing cursor is two pixels wide.

Passing `cwidth` as zero makes the flashing cursor invisible. This may be appropriate when, for example, highlighting some found text by means of the call `hEBSetSelect`.

hEBInsert

Insert text buffer into edit box

```
INT hEBInsert(VOID *ebH, TEXT *pb, INT blen);
```

Inserts the `blen` characters at `*pb` into the edit box at the cursor position. Any select region is cancelled first. The cursor is positioned afterwards to the end of the inserted text.

Returns zero for success, or a negative value if an error occurred (in which case the user will already have been notified).

hEBReplace

Replace selection in edit box

```
INT hEBReplace(VOID *ebH, TEXT *replace, INT backwards);
```

Deletes any highlighted selection, into the clipboard if the edit box has one, and then inserts the contents of the ZTS `*replace` at the cursor position. The cursor is finally positioned at the end of the text inserted, unless `backwards` is `TRUE`, in which case it is placed at the beginning of the selected text.

Returns zero for success, or a negative value if an error occurred (in which case the user will already have been notified).

The routine was originally designed with the functionality of a Replace command in mind. See also the section below on `hEBFind`.

hEBEvaluate

Evaluate edit box expression

```
INT hEBEvaluate(VOID *ebH);
```

Attempts to perform the standard Evaluate function on the specified edit box.

Returns zero for success, or a negative value if an error occurred (in which case the user will already have been notified).

Typical contents of the Evaluate routine called from the `ManageCommand` routine of an application would simply be as follows:

```
if (!CheckEditing())      /* check focus is positioned suitably */
    hEBEvaluate(ebH);     /* ignore any error */
```

hEBCopy

Copy function for edit box

```
INT hEBCopy(VOID *ebH);
```

Attempts to perform the standard Copy function on the specified edit box.

Returns zero for success, or a negative value if an error occurred (in which case the user will already have been notified).

But if there is no select region (and hence nothing to copy into the clipboard), the special value 1 is returned.

Typical contents of the Copy Text routine called from the `ManageCommand` routine of an application would accordingly be as follows:

```

if (!CheckEditing())
{
    if (hEBCopy(ebH)>0)
        wInfoMsg("No text to copy");
    else
        wInfoMsg("Text copied");
}

```

hEBPaste Paste function for edit box

```
INT hEBPaste(VOID *ebH);
```

Attempts to perform the standard Paste function on the specified edit box.

Returns zero for success, or a negative value if an error occurred (in which case the user will already have been notified).

But if the clipboard is empty (and hence there is nothing to paste), the special value 1 is returned.

Typical contents of the Paste routine called from the ManageCommand routine of an application would accordingly be as follows:

```

if (!CheckEditing())
{
    if (hEBPaste(ebH)>0)
        wInfoMsg("No text to insert");
}

```

hEBFind Find text within edit box

```
INT hEBFind(VOID *ebH, TEXT *str, INT flags);
```

Attempts to find a copy of the ZTS *str in the contents of the specified edit box. If successful, returns TRUE and automatically creates a highlighted selection over the copy found. Otherwise returns FALSE (except if an error occurred, in which case the return value is negative, and the user will already have been notified).

The following bits in flags determine how the search is done:

HEB_FIND_BACKWARDS	Search backwards from the cursor position (the default is to search forwards from the cursor position)
HEB_FIND_CASESENS	The search is case sensitive (the default is for a case insensitive search).

hEBClearChanged Clear edit box changed flag

```
VOID hEBClearChanged(VOID *ebH);
```

Clears the internal "changed" flag of the specified edit box.

This flag is clear when the edit box is first initialised.

This flag is set whenever the contents of the edit box change as a result of any of the calls hEBHandleKey, hEBInsert, hEBEvaluate, hEBPaste, or hEBReplace.

Typically, an application might call hEBClearChanged following calls to hEBSetText or hEBSenseText.

hEBSenseChanged Sense if edit box contents have changed

```
INT hEBSenseChanged(VOID *ebH);
```

Returns the "changed" flag of the specified edit box. See above for some further discussion.

hEBShowSymbols Hide or show edit box symbols

```
INT hEBShowSymbols(VOID *ebH, INT flag);
```

Shows or hides end of paragraph symbols and visible space markers, depending on the value of flag (TRUE to show them, FALSE to hide them).

These symbols are hidden by default.

Returns zero for success, or a negative value if an error occurred (in which case the user will already have been notified).

Even if the return value is negative (indicating a failure to re-format the edit box), the record within the edit box of whether to show these symbols will be updated as requested.

hEBClose

Close an edit box

```
VOID hEBClose(VOID *ebh);
```

Frees the resources allocated for the specified edit box.

Any application which uses an edit box throughout its lifetime has no need to close the edit box prior to exiting as any resources used will automatically be freed when the application terminates. However, it may be sensible to close an edit box explicitly if it is only used for a short period during the lifetime of the application. This avoids resources being tied up unnecessarily.

hEBSenseDoc

Return handle of edit box document component

```
VOID *hEBSenseDoc(VOID *ebh);
```

This function returns the handle of the document component of an edit box allowing it to be manipulated by other Hwif functions.

In some circumstances, the manipulation of the document component rather than the whole edit box can significantly reduce overhead (see hEDCapacity, hEDInsert, hEBDocChanged).

The parameter ebh must contain the handle of the opened edit box whose document component is required.

hEDCapacity

Set capacity of document component

```
INT hEDCapacity(VOID *doc,UINT maxlen);
```

This function sets the capacity of the document component of an edit box. In other words, it specifies the maximum size of the text that the edit box can hold.

The parameter doc must contain the handle of the document component of the edit box as returned from a call to hEBSenseDoc. The maximum size of the text is specified in parameter maxlen.

The function returns zero if successful or a negative value otherwise. If an error occurs, the user will already have been notified.

hEDInsert

Insert text into document component

```
INT hEDInsert(VOID *doc,UINT pos,VOID *buf,UINT len);
```

This function inserts text into the document component of an edit box.

The parameter doc must contain the handle of the document component of the edit box as returned from a call to hEBSenseDoc.

The text to be inserted is located at buf and is len bytes long. The text is inserted at offset pos within the document component.

The function returns zero if successful or a negative value otherwise. If an error occurs, the user will already have been notified.

hEBDocChanged

Notify edit box that document has changed

```
INT hEBDocChanged(VOID *ebh);
```

Notifies the edit box that the content of its document component has changed. This causes the text to be re-formatted.

The parameter ebh must contain the handle of the edit box to be notified.

The function returns zero if successful or a negative value otherwise. If an error occurs, the user will already have been notified.

Note that no attempt is made to validate the cursor position. It is the programmer's responsibility to set this correctly.

hEBSetMargin

Set word-wrap margin

```
INT hEBSetMargin(VOID *ebh,UINT right);
```

This function allows the right hand word-wrap margin of an edit box to be set. This value, in effect, sets a limit on the amount of text that can be displayed on one line before being wrapped around onto the next line.

The parameter *ebh* must contain the handle of the edit box whose margin is to be set. The margin itself is defined by the value in the parameter *right*; this value is measured in pixels.

Setting *right* to a large value such as 4096, effectively turns off word-wrap. In fact this is the only sensible use of this function. If there were a need to set a margin smaller than the width of the edit box then it would be easier to use a smaller edit box.

The following code fragment suggests how it might be used to turn off word-wrap:

```
H_EDIT_BOX heb;
VOID *      ebh;

...
ebh = hEBOpen(H_EDIT_BOX_VISLINES|H_EDIT_BOX_LEFT_CURSOR,&heb);
hEBSetMargin(ebh,4096);
...
```

The function returns zero if successful or a negative value otherwise. If an error occurs, the user will already have been notified.

hEBSenseMargin

Sense word-wrap margin

```
UINT hEBSenseMargin(VOID *ebh);
```

This function senses and returns the current value of the right hand word-wrap margin of an edit box whose handle is passed in *ebh*.

hEBPosToXL

Convert position to line number and pixel offset

```
UINT hEBPosToXL(VOID *ebh,H_SCLAY_PLX *plx);
```

This function converts a character position, within the text of the edit box whose handle is passed in *ebh*, to a position on the screen.

The character position is passed in the *pos* member of the *H_SCLAY_PLX* struct pointed to by *plx*. This struct is defined in *hwif.h* as:

```
typedef struct
{
    UWORD pos;
    WORD line;
    WORD x;
} H_SCLAY_PLX;
```

If the character position is visible on the screen, the line number and horizontal pixel offset within the line are written to *plx->line* and *plx->x* respectively and the function returns zero.

If the character position is above the screen, a value of -30000 is written to *plx->line* and the function returns -1.

If the character position is below the screen or beyond the end of the text, a value of +30000 is written to *plx->line* and the function returns +1.

Printing and print support functions

hPrintSetupDialog

Invoke Print Setup dialog

VOID hPrintSetupDialog(VOID);

This is *NOT* to be confused with the function hPrinterSetupDialog described later.

It presents the standard Print Setup dialog, waiting until it is complete.

The Epoc static DatLocked is set to TRUE for the duration of the call to hPrintSetupDialog.

As for the Print Setup dialogs in the built-in applications, the dialog sometimes allows the user to make choices that the printer eventually chosen cannot deliver. For example, some style combinations (Bold *and* italic) may not be supported in some fonts (in such a case, the text will probably come out *either* as bold *or* as italic). Again, not all printers support landscape orientation.

hPrinterSetupDialog

Invoke Printer Configuration Setup dialog

VOID hPrinterSetupDialog(VOID);

This is *NOT* to be confused with the function hPrintSetupDialog described earlier.

It presents the standard Printer Configuration Setup dialog, waiting until it is complete.

The Epoc static DatLocked is set to TRUE for the duration of the call to hPrinterSetupDialog.

The Printer Configuration Setup dialog behaves in exactly the same way as for the built-in applications in that it requests printer device information and provides an entry into the print preview settings dialog.

hPrint

Print data

VOID hPrint(INT PrintLine(H_PRINT *));

Prints from the application, according to parameters specified by the user in any Print Setup dialog.

The callback function PrintLine is repeatedly called by system code, as printing progresses, until printing terminates. Each time the function is called, the application has to return either TRUE or FALSE:

- a return value of TRUE indicates that more data is being passed to print; this data is written into the fields of the H_PRINT struct passed
- a return value of FALSE indicates that the application has no more data to print.

The H_PRINT struct is defined as follows:

```
typedef struct
{
    WORD flags;
    WORD typf;
    WORD fheight;
    WORD style;
    WORD down;
    WORD indent;
    WORD height;
    WORD right;
    TEXT *buf;
    UWORD blen;
} H_PRINT;
```

Most of the fields of this struct will have already been set to suitable values before PrintLine is called. All but the most ambitious of Hwif application writers should be content to write to, at most, the following fields:

blen

This must be set if the bit H_PRINT_TEXT is set in flags (as it is by default). It gives the number of characters to be printed.

<code>buf</code>	This must be set if <code>blen</code> is non-zero (and <code>H_PRINT_TEXT</code> is set in <code>flags</code>). It gives the address of a contiguous buffer holding the <code>blen</code> characters to be printed. Note that the buffer must be permanent (as opposed to being declared on the stack of the routine <code>PrintLine</code>).
<code>indent</code>	This (zero by default) specifies the additional amount to indent the print position by, horizontally, prior to the specified text being printed. Any value given must be in printer units (see below).
<code>flags</code>	The bit <code>H_PRINT_KEEP</code> may be <i>ORed</i> in, to request that, if possible, this line of text be kept together on the same page with the following line; the bit <code>H_PRINT_PAGE</code> may be <i>ORed</i> in to force the emission of a form feed prior to the line being printed.
<code>style</code>	Any of the bits <code>H_PRINT_STY_UNDERLINE</code> , <code>H_PRINT_STY_BOLD</code> , <code>H_PRINT_STY_ITALIC</code> , <code>H_PRINT_STY_SUPER</code> , and <code>H_PRINT_STY_SUB</code> may be <i>ORed</i> in, to further embellish the font chosen by the user, in the Print Setup dialog, as the default font.

Note that in both the last two cases, it is crucial to *or* in the bits required, rather than simply setting them with an assignment statement. Note also that in the case of `style`, there is no guarantee that just because a particular font style is set, the current printer will be able to fulfil the request made on it.

Word wrapping during printing

Before any text is printed, ROM resident printing code checks that it will fit in the width available to it. If not, word wrap is performed, with any excess being printed on the subsequent line instead. This subsequent line is printed without any additional call to the `PrintLine` function. That is, more than one line may be printed as a consequence of any one call to `PrintLine`.

The word-wrap calculation performed makes the following assumptions:

- the text is to fit into the full width of the page, minus its left and right margins, and minus any indent specified
- the text is to be printed in the font and style specified by the user in the Print Setup dialog.

Some fonts may change their width if (for example) they are bolded or italicised; if the application *ors* any such bit into `style`, there is, accordingly, a risk that the word-wrap calculation will be incorrect. (Similarly - but even more so - if the application writes into the `right` field of the `H_PRINT` struct, or if the `H_PRINT_LINE` bit is removed from `flags`. See below for a discussion of these possibilities.)

If word-wrap is required, the `H_PRINT_KEEP` bit is set into `flags` provided the user has not set the *Allow widows/orphans* choice list in the Print Setup dialog suite to "yes".

Page breaks during printing

Page breaks are automatically calculated by ROM resident layout software, and do not have to be inserted by application code.

However, as noted above, it is possible for the application to force a given piece of text to appear at the top of a new page.

Limitations during the `PrintLine` callback

The application is actually in a somewhat vulnerable state during a `PrintLine` callback:

- Only a limited amount of the Hwif system services are available to it in this state; in particular, no call can be made to `uRunDialog`, `uPresentMenus`, `hPrintSetupDialog`, `hPrinterSetupDialog` or (recursively) `hPrint` - on pain of indeterminate damage resulting
- Only a limited amount of time should be spent inside each call to `PrintLine`; if an application remains in `PrintLine` indefinitely, and the application is tasked into background and then into foreground again in the meantime, the Printing dialog in the foreground will not be redrawn properly, and its Escape action button will not be effective.

The Epoc static `DatLocked` is set `TRUE` for the duration of the call to `hPrint`.

hPrintSetSI **Set subsequent indent for printing**

VOID hPrintSetSI(UINT subseqind);

Sets the indent to be used for the second and following lines of any wrapped block of text. The value given must be in printer units (see below).

The subsequent indent defaults to zero. But once it is set, by one PrintLine call, it retains its value during subsequent PrintLine calls (unless it is changed again).

hPrintSensePageWidth **Sense page width for printing**

INT hPrintSensePageWidth(VOID);

Returns the width of the page, minus its left and right margins, in current printer units.

hPrintSenseBufWidth **Sense printed width of buffer**

INT hPrintSenseBufWidth(TEXT *buf, INT blen);

Returns the width of the passed buffer of text, in current printer units. The width is worked out in the default font and style specified by the user in the Print Setup dialog.

The result may be of use in calculating indents for lines. For example, it allows the printing of centred text.

Advanced possibilities when printing

By default, the H_PRINT_LINE bit is always set in flags. This has the following effect:

- before printing any text, the print position is moved down by an amount equal to the sum of height (equal by default to the height of the default font) and down (zero by default), except that down is ignored for the first line on a page
- the print position is moved back to the left margin, and then in by indent.

None of these things happen if the H_PRINT_LINE bit is missing. Printing just continues from where it left off the previous time.

In order to print in columns, it is possible to proceed as follows:

- print the first column as per normal
- the next time PrintLine is called, clear the H_PRINT_LINE and H_PRINT_TEXT flags that are set by default; instead, set the H_PRINT_RIGHT flag and supply a suitable value of right
- following that, keep the H_PRINT_LINE bit clear, but pass the text corresponding to the second column
- repeat for any additional columns.

The value specified for right has, again, to be in printer units. It can be calculated from the result of a call to hPrintSenseBufWidth.

Inevitably, there are limitations with this approach. For greater control over printing, it is necessary to interact more directly with the object classes in the Series 3 ROM.

Miscellaneous functions

hCrackCommandLine **Crack the command line**

INT hCrackCommandLine(VOID);

Reads the command line and sets up appropriate initial values of various reserved statics, including DatUsedPathNamePtr (the full path name of any file to open or create).

The return value has significance for file-based applications:

- 'O' the specified file already exists, and is to be opened

- 'C' a file of the specified name is to be created and opened anew, with any existing file of that name to be overwritten
- 0 the command line is not present (for example, the application may have been run from a source other than the System Screen).

For example:

```
LOCAL_C VOID SpecificInit(VOID)
{
    INT command;
    INT bid;
    VOID *fcb;

    command=hCrackCommandLine();
    bid=ObeySystemCommand(command,DatUsedPathNamePtr,&fcb);
    ...
}
```

hSetUpStatusNames Notify a change in the name of the file

VOID hSetUpStatusNames(TEXT *pb);

Changes the value of DatUsedPathNamePtr to pb and makes other required associated changes in reserved statics.

For example:

```
LOCAL_C VOID ChangeName(TEXT *newname)
{
    p_scpy(&filename[0],newname);
    hSetUpStatusNames(&filename[0]);
    wsUpdate(WS_UPDATE_NAME);
}
```

Note that *pb must be a *fully parsed* filename (such as is returned by a file selector item in a dialog). Further, the buffer *pb must be a permanent one (as opposed to being defined on the stack of a routine such as ChangeNames).

hEnsurePath Ensure that the path exists for a filename

VOID hEnsurePath(TEXT *fname);

Ensures that the path of the specified filename exists.

Should generally be called following a *New file* or *Save as* menu command.

For example:

```
if (RunDialog())
{
    /* Save As dialog successfully completed */
    savebuf[1+savebuf[0]]=0; /* BCS to ZTS conversion */
    hEnsurePath(&savebuf[1]);
    DoSave(&savebuf[1]);
    wInfoMsg("Saved");
}
```

hDlgPosition Position a dialog

VOID hDlgPosition(INT x,INT y);

Affects the position in which the current dialog will appear.

If x is negative, the dialog will appear on the left edge of the screen; if positive, on the right edge; if zero, centred horizontally.

If y is negative, the dialog will appear on the top edge of the screen; if positive, on the bottom edge; if zero, centred vertically.

Thus in all there are 9 possible locations for the dialog.

In the absence of this call being made for a dialog, it is positioned in the centre of the screen, just as if the call

```
hDlgPosition(0,0);
```

had been made.

hDTMFString

Emit DTMF tones

```
INT hDTMFString(TEXT *zts);
```

Emits DTMF tones for the passed string.

Uses the tone lengths and pauses as specified by the user in the *World* application (or otherwise), reverting to system defaults in the absence of any such setting.

Returns zero for success or a negative error if the sound system was unavailable (on account of being hogged by another application). The call embodies a double retry before failing.

The call waits for the tones to be emitted before returning.

For example:

```
if (hDTMFString("123"))
    wInfoMsg("Sound system busy");
```

hIsDbfCompressible

Check if database file is compressible

```
INT hIsDbfCompressible(VOID *dH);
```

Returns TRUE if the database file open on database channel dH is compressible (ie if it is on any medium other than Flash), and FALSE otherwise.

For example,

```
LOCAL_C VOID CompressFile(VOID)
{
    UINT state;

    if (!hIsDbfCompressible(dH))
        wInfoMsg("Cannot compress on Flash");
    else
    {
        state=DbfStateDisabled;
        Check(DbfCompress(&state,dH));
        wInfoMsg("File compressed");
    }
}
```

hHelpSubSystem

Access help engine

```
VOID hHelpSubSystem(INT startid, INT indexid);
```

This function allows the developer to access the same Help engine as used by the inbuilt applications on the Series 3.

Like uPresentMenus and uRunDialog, hHelpSubSystem only returns when the entire operation (in this case, the Help operation) has been completed by the user. It does its own error handling internally, automatically presenting suitable error messages where necessary.

It is worth noting that other events cannot be notified when inside this call; messages from the system screen are ignored and the expiry of timers is effectively delayed until the call completes.

The two parameters are the IDs of suitable resources in an application resource file in which the hierarchy of Help text is defined:

- The first is the resource ID of the "top-level" Help resource in the resource file.
- The second is the resource ID of the "index" set of Help resources.

For further general information on resource files, see the *Additional System Information* manual.

The following "structures" are used in building Help resources. For further information and examples on the use of these structures in building a hierarchy of Help text, see the *Introduction To HWIF* chapter in this manual.

HELP_ARRAY**Help resource structure**

```
STRUCT HELP_ARRAY
{
    LINK topic_id;
    TEXT topic;
    LEN BYTE STRUCT strlst[];
}
```

where any instances of the strlst fields must always be as STRING

STRING**Help resource structure**

```
STRUCT STRING
{
    TEXT str;
}
```

TOPIC_ARRAY**Help resource structure**

```
STRUCT TOPIC_ARRAY
{
    LEN BYTE LINK id_lst[]
}
```

hDeclareAppRcb**Pass handle of Appl.Res.Channel**

```
VOID hDeclareAppRcb(VOID *rcb);
```

Before an application resource file can be used (e.g. to invoke Help), a channel to the file must be opened and its handle passed to the system.

This function performs the action of passing the handle of an (already) opened application resource file channel to the system via the rcb parameter.

It is called from the function hInitAppRcb, described later, which opens an application's built in resource file.

hInitAppRcb**Open channel to resource file**

```
VOID *hInitAppRcb(VOID);
```

This function opens a channel to an application's built in resource file and returns its handle. The handle itself is passed to the system using the function hDeclareAppRcb, described earlier.

In general, this function allows the application's built in resource file to be accessed; the handle itself is used explicitly by such functions as hRequestReplacePack described later.

In actual fact the function creates and initialises a resource file object. However, for those unfamiliar with Object Oriented Programming, it is easier to think in terms of opening a channel.

The function is implemented as shown below. Note again that it uses hDeclareAppRcb described earlier.

```
GLDEF_C VOID *hInitAppRcb(VOID)
{
    /* create and initialise resource file object */
    INT ret;
    VOID *rcb; /* resource file object handle */

    rcb = p_new(1,C_RSCFILE);
    ret = p_entsend3(rcb,O_RS_INIT,DatCommandPtr); /* pass file name */
    if (ret < 0) /* no resource file found */
        p_exit(ret); /* fatal (programmer) error*/
    hDeclareAppRcb(rcb);
}
```

The implementation assumes that the application resource file is built into the *.app* file following standard conventions.

If a resource file is to be used which is separate from the application, then the above code could be used but the reference to *DatCommandPtr* would need to be changed. In general the third parameter to the *p_entersend3* function in the above code should contain a pointer to the resource file name.

hRequestReplacePack Request insertion of resource file pack

```
VOID hRequestReplacePack(VOID *rcb, TEXT *fname)
```

This function is normally used when a channel to an application resource file (with handle *rcb*) has previously been opened but is now found to be unavailable. This is most commonly caused by the removal of the pack on which the resource file is located.

The function issues a warning message to the user which PERSISTS until the appropriate resource file, as specified by *fname*, has been loaded and found.

fname must point to a buffer of at least *P_FNAME_SIZE* characters.

The following code fragment illustrates the use of this function. It makes use of the *rscfile* class; this is documented in the *resource files* chapter in the *additional system information* manual.

Also note that the function *InitRcb* is, in essence, the same code that implements the utility function *hInitAppRcb*, described earlier.

It simply displays an information message where the text is taken from the resource file whose name is contained in *fname*.

```
GLREF_D VOID *rcb;
GLREF_D TEXT fname[P_FNAME_SIZE];

LOCAL_C VOID InitRcb(VOID)
{
    INT ret;

    rcb = p_new(1,C_RSCFILE);
    ret = p_entersend3(rcb,O_RS_INIT,&fname[0]);
    if (ret < 0)
        p_exit(ret);
    hDeclareAppRcb(rcb);
}

LOCAL_C VOID DoInfoMessage(INT index)
{
    TEXT buf[60];

    while(p_send4(rcb,O_RS_READ_BUF,&buf[0],index) < 0)
        hRequestReplacePack(rcb,&fname[0]);
    wInfoMsg(&buf[0]);
}

...
InitRcb();
...
index = INFO_USEFUL_MESSAGE;
DoInfoMessage(index);
....
```

hSetSystemResourceLang Set system resource file language

```
VOID hSetSystemResourceLang(UINT langnum);
```

Changes the language used by system resources to the that specified by *langnum*. Language numbers are as described for the *PLIB* function *p_getlanguage*.

This function is useful only for programs running on multi-lingual machines. It changes the system resource for your single application and NOT for other applications running at the same time.

Note that *hSetSystemResourceLang* does *not* alter messages from the *.cfo* config file.

An example of its use might be:

```
LOCAL_C VOID SpecificInit(VOID)
{
    uEscape(FALSE);
    hSetSystemResourceLang(6);
    MainWid = uFindMainWid();
    ...
}
```

to try to set the system resource file for language number 6 (Swedish).

hSetVarrayInChlist **Set choice list to use a variable array**

```
VOID hSetVarrayInChlist(INT index, INT nsel, VOID *varray);
```

This function permits a choice list to be added to a dialog item with index number `index`. The list of items is found in the variable array object with handle `varray`. The default selected item is set to array item `nsel`.

The assumption is made that the dialog item is already a choice list.

This function provides a means of setting a choice list containing more than 255 items. The implementation is as shown below.

```
GLDEF_C VOID hSetVarrayInChlist(INT Index, INT nsel, VOID *varray)
{
    SE_CHLIST set;

    set.nsel = nsel;
    set.data = varray;
    set.set_flags = SE_CHLIST_DATA | SE_CHLIST_NSEL | SE_CHLIST_RETAIN;
    p_send4(DatDialogPtr, O_WN_SET, index, &set);
}
```

An example of the use of `hSetVarrayInChlist` is as shown below. Note that `CreateVariableArray()` is a small user written function which would create the variable array.

```
LOCAL_C VOID LongChoiceList(VOID)
{
    UWORD junk;
    UWORD used;
    UWORD longnsel;
    PR_ROOT *vastr;

    junk    = 1;
    used    = 0;
    longnsel = 1;
    vastr    = CreateVariableArray();

    if (uOpenDialog("Example long list"))
        return;
    if (uAddChoiceList("Ignored",&junk,"Date","Time",NULL))
    if (uAddChoiceList("Long choice",&used,NULL))
        return;
    hSetVarrayInChlist(2,longnsel,vastr);
    if (uRunDialog() > 0)
    {
        ...
    }
}
```

hLoadOwnDyl **Load applications's DYL**

```
INT hLoadOwnDyl(INT index);
```

This is a utility function that loads a DYL from the multiple DYL file built into the application's `.app` file and makes it ready for use.

If successful, the function returns the handle of the loaded DYL. It returns 0 if the file can not be opened.

The parameter `index` specifies which DYL is to be loaded from the multiple DYL file. Zero refers to the first DYL, one refers to the second and so on.

The function is implemented as shown below:

```
GLDEF_C INT hLoadOwnDyl(INT index)
{
    INT DylHandle;
    VOID *fcb;
    INT ret;

    DylHandle = 0;
    ret = p_openlib(&fcb,DatCommandPtr);
    if (!ret)
    {
        p_loadfilelib(fcb,index,&DylHandle,TRUE);
        p_close(fcb);
    }
    return(DylHandle);
}
```

hLastSystemKey

Get last key press

```
UINT hLastSystemKey(UINT *pmodifiers);
```

This function allows the application to find out what keypress was last processed by the system on its behalf.

The function returns the keypress value while the bit values representing any modifier keys pressed (e.g SHIFT, CTRL etc) are placed in `*pmodifiers`.

More information on keys and key modifiers can be found in the *Window Server Reference*.

This function is particularly useful in determining what key press caused a dialog to terminate. For example, a dialog can be terminated by pressing ESC or HELP (as well as other key combinations).

Knowing which key caused the dialog to terminate allows the application to take appropriate action. For example, if a dialog were terminated by pressing the HELP key, the application could continue by displaying help information.

For example, consider the following code fragment:

```
UINT key;
UINT modifiers;

... /* build a dialog */

uRunDialog();
key = hLastSystemKey(&modifiers);
...
```

If the dialog were terminated by pressing SHIFT+CTRL+ENTER, then `key` would contain the value `W_KEY_RETURN` (0x0D) while `modifiers` would contain the value `W_SHIFT_MODIFIER+W_CTRL_MODIFIER` (0x06).

Note that this function is only supported on the Series 3a.

hOODialog

Call an object oriented dialog

```
VOID hOODialog(INT catHandle,INT class,INT resid,VOID *rbuf);
```

This function allows an HWIM (object oriented) dialog to be run from within an Hwif program. Use of this function requires a knowledge of programming HWIM dialogs, as described in the *Dialogs* chapter of the *Object Oriented Programming Guide*. See also the *Combining Hwif with object oriented code* section of the *Advanced Use of Hwif* chapter of this manual.

The dialog object to be created and run is specified by its class number `class` and the category handle `catHandle` of the category containing that class. The dialog contents are specified by the resource with

resource ID `resid`. Data may be transferred to and from the dialog by means of a result buffer pointed to by `rbuf`.

If the dialog's class is defined in an external DYL category, this category must be loaded before a call to `hOODialog`. It is good practice to unload the category as soon as its code is no longer required.

The low level h-layer functions

This section may be omitted by almost all readers. Possible exceptions include readers wishing to construct equivalents to the u-layer calls that, instead of returning errors on failure, call `p_leave`.

Menu bar interactions - overview

Any menu bar interaction consists of the following three stages:

- a call to `hMenuOpen`
- one or more calls to `hMenuAdd`
- a call to `hMenuRun`.

The calls to `hMenuOpen` and `hMenuAdd` progressively build up a data structure in the form required by the subsequent `hMenuRun` call. The menu bar is displayed only when the call to `hMenuRun` is made.

A menu bar itself consists of a series of one or more menu *cards*. Each call to `hMenuAdd` adds another card to the menu bar.

In turn, each card has a *title* and a series of *items*. The title is what appears on the menu bar, and the items are the various choices presented to the user. Each item consists of some *text* and an *accelerator*.

Like the menu bar as a whole, each menu card is built up in stages. Each menu card requires:

- a call to `hCardOpen`
- one or more calls to `hCardAdd`
- a call to `hMenuAdd` (to add it into the current menu bar).

Each call `hMenuOpen` and `hMenuAdd` allocates extra memory specifically for the menu bar. This memory is freed following a successful call to `hMenuRun`. If however the process of building up the menu bar fails before the call to `hMenuRun`, the application should generally call `hMenuClose` to free this memory.

(However, any call to `hMenuOpen` when there has been a previous call to `hMenuOpen` not matched by a following call to `hMenuRun` or `hMenuClose` also has the effect of performing an `hMenuClose` before proceeding.)

Calls to `hCardOpen` and `hCardAdd` also allocate memory, associated with the particular menu card. This memory is freed neither by the subsequent call `hMenuAdd`, nor by the call `hMenuRun`, nor by a call `hMenuClose`. Instead, the memory associated with a menu card remains allocated until specifically freed by a call to `hCardClose`.

Dialog interactions - overview

In contrast to the case with menus - in which one u-layer Hwif call (`uPresentMenus`) encapsulates the functionality of some seven h-layer calls - for dialogs, there is a reasonably close correspondence between u-layer calls and h-layer calls.

The main differences between the two sets of dialog calls are:

- the h-layer calls require item prompts and dialog titles in BCS form, whereas the u-layer calls require them in ZTS form
- the u-layer calls automatically present an appropriate error notification on detection of an error
- the u-layer contains convenience utilities for adding a choice list or an action button list to the current dialog, in each case encapsulating the functionality of some four h-layer functions.

hIFInit**Low level initialisation**

```
INT hIFInit(VOID *concb);
```

Initialises an application for subsequent menu bar or dialog interactions.

Returns zero for success or a negative error. However, an application that sets its start-up heap appropriately can legitimately assume the function always succeeds.

The control block of the console (*concb*) must be passed. This is used internally by the ROM code just before commencing any dialog or menu interaction (in response to an *hDlgRun*, *hMenuRun*, *hPrinterSetupDialog*, or *hPrint* call), and again just after such an interaction. In both cases, an I/O message

```
UINT func;
```

```
func=P_SCR_DISABLE_READS;
p_iow4(concb,P_FSET,&func,&state);
```

is sent to the console, with *state* set TRUE on commencing the interaction, and set FALSE on concluding it.

hCardOpen**Open a menu card**

```
VOID *hCardOpen(VOID);
```

Prepares to build up a menu card.

Returns a handle to use in subsequent calls to *hCardAdd*, *hMenuAdd*, and *hCardClose*, or else 0 for OOM (no Window Server resources are required by the call).

hCardAdd**Add an item to a menu card**

```
INT hCardAdd(VOID *card,INT index,INT accel,TEXT *str);
```

Adds an item to the menu card with handle *card* (as returned by a prior call to *hCardOpen*).

The item is inserted as the item with position *index* in the menu card. (Thus ordinarily *index* would be 1 the first time *hCardAdd* is called for a card, 2 the second time, and so on).

The item has accelerator *accel*, and text defined by the BCS *str*.

Returns 0 for success or a negative error. (No Window Server resources are required by the call).

hCardClose**Close a menu card**

```
VOID hCardClose(VOID *card);
```

Frees all the memory resources associated specifically with the menu card with handle *card* (as returned by a prior call to *hCardOpen*). Harmless if *card* is zero (may be useful in error-recovery code).

hMenuOpen**Open a menu bar**

```
INT hMenuOpen(VOID);
```

Prepares for a menu bar interaction.

Returns 0 for success or a negative error. Applications should *always* test the return value, since this routine involves opening another Window Server window.

hMenuAdd**Add a menu card to a menu bar**

```
INT hMenuAdd(TEXT *title,VOID *card);
```

Adds the menu card identified by *card* and with title given in BCS form by *title* into the current menu bar.

The card is always added at the end of the current menu bar.

Returns 0 for success or a negative error. (No Window Server resources are required by the call).

hMenuRun**Run a menu bar**

```
INT hMenuRun(VOID);
```

Presents the menu bar prepared by earlier calls to `hMenuOpen` and `hMenuAdd`, and returns only when the user has made a choice (or cancelled).

Returns 0 if the user cancelled, or a negative error value, or else the accelerator of the item selected by the user. Applications must not assume that the function always succeeds, since additional Window Server resources are involved in its execution.

hMenuClose**Close a menu bar**

```
VOID hMenuClose(VOID);
```

Frees all the memory resources associated specifically with the current menu bar. (Harmless if there is no current menu bar.)

hDlgOpen**Open a dialog**

```
INT hDlgOpen(TEXT *title);
```

The low-layer version of `uOpenDialog`.

hDlgRun**Run a dialog**

```
INT hDlgRun(VOID);
```

The low-layer version of `uRunDialog`.

hDlgClose**Close down an incomplete dialog**

```
VOID hDlgClose(VOID);
```

Frees all resources known to the current dialog (if any).

Not called from within any of the u-layer functions under the rationale that a call to `hDlgClose` is implicitly made every time a menu bar or dialog interaction is initiated.

hDlgAdd**Add an item to a dialog**

```
INT hDlgAdd(UWORD type, TEXT *prompt, VOID *data);
```

The low-layer version of `uAddDialogItem`.

In addition to the values of `type` discussed in the documentation for `uAddDialogItem`, the following are also available:

- | | |
|-------------------------------|---|
| <code>H_DIALOG_CHOICE</code> | for a choice list, with corresponding data struct <code>H_DI_CHOICE</code> |
| <code>H_DIALOG_BUTTONS</code> | for an action list of buttons, with corresponding data struct <code>H_DI_BUTTONS</code> . |

hChoiceOpen**Open a choice list**

```
VOID *hChoiceOpen(VOID);
```

Prepares to build up a choice list.

Returns a handle to use in subsequent calls to `hChoiceAdd`, `hDlgAdd`, and `hChoiceClose`, or else 0 for OOM (no Window Server resources are required by the call).

hChoiceAdd**Add an item to a choice list**

```
INT hChoiceAdd(VOID *hand, INT index, TEXT *str);
```

Adds an item to the choice list with handle `hand` (as returned by a prior call to `hChoiceOpen`).

The item is inserted as the item with position `index` in the choice list. (Thus ordinarily `index` would be 1 the first time `hChoiceAdd` is called for a choice list, 2 the second time, and so on).

The item has text defined by the BCS str.

Returns 0 for success or a negative error. (No Window Server resources are required by the call).

hChoiceClose **Close an incomplete choice list**

```
INT hChoiceClose(VOID *hand);
```

Frees all the memory resources associated specifically with the choice list with handle *hand* (as returned by a prior call to *hChoiceOpen*). Harmless if *hand* is zero (may be useful in error-recovery code).

This call only needs to be made if a failure occurs before the completion of the associated *hDlgAdd* call, since from that time on, the resources of the choice list fall under the responsibility of the dialog as a whole.

hButtonOpen **Open a button action list**

```
VOID *hButtonOpen(VOID);
```

Prepares to build up an action list of buttons.

Returns a handle to use in subsequent calls to *hButtonAdd*, *hDlgAdd*, and *hButtonClose*, or else 0 for OOM (no Window Server resources are required by the call).

hButtonAdd **Add a button to an action list**

```
INT hButtonAdd(VOID *hand, INT index, INT key, TEXT *str);
```

Adds a button to the action list with handle *hand* (as returned by a prior call to *hButtonOpen*).

The item is inserted as the button with position *index* in the action list. (Thus ordinarily *index* would be 1 the first time *hButtonAdd* is called for an action list, 2 the second time, and so on).

The button has keycode defined by *key* and text defined by the BCS str.

Returns 0 for success or a negative error. (No Window Server resources are required by the call).

hButtonClose **Close an incomplete action list**

```
VOID hButtonClose(VOID *hand);
```

Frees all the memory resources associated specifically with the action list with handle *hand* (as returned by a prior call to *hButtonOpen*). Harmless if *hand* is zero (may be useful in error-recovery code).

This call only needs to be made if a failure occurs before the completion of the associated *hDlgAdd* call, since from that time on, the resources of the action list fall under the responsibility of the dialog as a whole.

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

INDEX

H_DIALOG_DATE 86
 H_DIALOG_EDIT 86
 H_DIALOG_FLOAT 85
 H_DIALOG_FSEL 87
 H_DIALOG_NUMBER 85
 H_DIALOG_SEDIT 87
 H_DIALOG_TEXT 85
 H_DIALOG_TIME 86
 H_DIALOG_XINPUT 87
 H_DTEDIT 98
 H_SE_DTEDIT 99
 hButtonAdd 119
 hButtonClose 119
 hButtonOpen 119
 hCardAdd 117
 hCardClose 117
 hCardOpen 117
 hChoiceAdd 118
 hChoiceClose 119
 hChoiceOpen 118
 hCrackCommandLine 109
 hDeclareAppRcb 112
 hDlgAdd 118
 hDlgClose 118
 hDlgOpen 118
 hDlgPosition 110
 hDlgRun 118
 hDTClose 97
 hDTEmphasise 98
 hDTHandleKey 98
 hDTMFString 111
 hDTPOpen 97
 hDTSelfCheck 97
 hDTSense 97
 hDTSet 97
 hEBChangeWidth 102
 hEBClearChanged 104
 hEBClose 105
 hEBCompleteFormat 100
 hEBCopy 103
 hEBDocChanged 105
 hEBEmphasise 101
 hEBEvaluate 103
 hEBFind 104
 hEBHandleKey 100
 hEBInsert 103
 hEBOpen 99
 hEBPaste 104
 hEBPosToXL 106
 hEBReplace 103
 hEBSenseChanged 104
 hEBSenseClipText 102
 hEBSenseDoc 105
 hEBSenseMargin 106
 hEBSenseSelect 102
 hEBSenseText 101

hEBSetClipText 102
 hEBSetCWidth 103
 hEBSetMargin 106
 hEBSetSelect 102
 hEBSetText 101
 hEBShowSymbols 104
 hEDCapacity 105
 hEDInsert 105
 HELP_ARRAY 112
 hEnsurePath 110
 hHelpSubSystem 111
 hIFInit 117
 hInitAppRcb 112
 hlsDbfCompressible 111
 hLastSystemKey 115
 hLoadOwnDyl 114
 hMenuAdd 117
 hMenuClose 118
 hMenuOpen 117
 hMenuRun 118
 hOODialog 115
 hPrint 107
 hPrinterSetupDialog 107
 hPrintSenseBufWidth 109
 hPrintSensePageWidth 109
 hPrintSetSI 109
 hPrintSetupDialog 107
 hRequestReplacePack 113
 hSetSystemResourceLang 113
 hSetUpStatusNames 110
 hSetVarrayInChlist 114
 hTTClose 97
 hTTOpen 95
 hTTSenseString 96
 hTTSetAbbreviations 95
 hTTSetFormat 95
 hTTSetTime 96
 STRING 112
 TOPIC_ARRAY 112
 uAddButtonList 83
 uAddChoiceList 84
 uAddDCL 88
 uAddDialogItem 84
 uAddGreyUline 88
 uBeginDCL 88
 uCancelGetKeyA 78
 uCheckHandle 93
 uCommonInit 74
 uDialogMenu 94
 uDisplayText 94
 uEnableGrey 76
 uErrorString 93
 uErrorValue 93
 uEscape 92
 uFindMainWid 93
 uForceToFront 93
 uGetKey 77
 uGetKeyA 77
 uGrowDCL 88
 uKeyPressOutstanding 79
 uLocateCommand 79
 uOpenDialog 82
 uPresentMenus 81
 uRunDialog 83
 uSetDialogUline 91
 uZTStoBCS 94

INDEX

101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325	326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425	426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450	451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525	526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541	542	543	544	545	546	547	548	549	550	551	552	553	554	555	556	557	558	559	560	561	562	563	564	565	566	567	568	569	570	571	572	573	574	575	576	577	578	579	580	581	582	583	584	585	586	587	588	589	590	591	592	593	594	595	596	597	598	599	600	601	602	603	604	605	606	607	608	609	610	611	612	613	614	615	616	617	618	619	620	621	622	623	624	625	626	627	628	629	630	631	632	633	634	635	636	637	638	639	640	641	642	643	644	645	646	647	648	649	650	651	652	653	654	655	656	657	658	659	660	661	662	663	664	665	666	667	668	669	670	671	672	673	674	675	676	677	678	679	680	681	682	683	684	685	686	687	688	689	690	691	692	693	694	695	696	697	698	699	700	701	702	703	704	705	706	707	708	709	710	711	712	713	714	715	716	717	718	719	720	721	722	723	724	725	726	727	728	729	730	731	732	733	734	735	736	737	738	739	740	741	742	743	744	745	746	747	748	749	750	751	752	753	754	755	756	757	758	759	760	761	762	763	764	765	766	767	768	769	770	771	772	773	774	775	776	777	778	779	780	781	782	783	784	785	786	787	788	789	790	791	792	793	794	795	796	797	798	799	800	801	802	803	804	805	806	807	808	809	810	811	812	813	814	815	816	817	818	819	820	821	822	823	824	825	826	827	828	829	830	831	832	833	834	835	836	837	838	839	840	841	842	843	844	845	846	847	848	849	850	851	852	853	854	855	856	857	858	859	860	861	862	863	864	865	866	867	868	869	870	871	872	873	874	875	876	877	878	879	880	881	882	883	884	885	886	887	888	889	890	891	892	893	894	895	896	897	898	899	900	901	902	903	904	905	906	907	908	909	910	911	912	913	914	915	916	917	918	919	920	921	922	923	924	925	926	927	928	929	930	931	932	933	934	935	936	937	938	939	940	941	942	943	944	945	946	947	948	949	950	951	952	953	954	955	956	957	958	959	960	961	962	963	964	965	966	967	968	969	970	971	972	973	974	975	976	977	978	979	980	981	982	983	984	985	986	987	988	989	990	991	992	993	994	995	996	997	998	999	1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1292	1293	1294	1295	1296	1297	1298	1299	1300	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311	1312	1313	1314	1315	1316	1317	1318	1319	1320	1321	1322	1323	1324	1325	1326	1327	1328	1329	1330	1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343	1344	1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1357	1358	1359	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372	1373	1374	1375	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386	1387	1388	1389	1390	1391	1392	1393	1394	1395	1396	1397	1398	1399	1400	1401	1402	1403	1404	1405	1406	1407	1408	1409	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1420	1421	1422	1423	1424	1425	1426	1427	1428	1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439	1440	1441	1442	1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455	1456	1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470	1471	1472	1473	1474	1475	1476	1477	1478	1479	1480	1481	1482	1483	1484	1485	1486	1487	1488	1489	1490	1491	1492	1493	1494	1495	1496	1497	1498	1499	1500	1501	1502	1503	1504	1505	1506	1507	1508	1509	1510	1511	1512	1513	1514	1515	1516	1517	1518	1519	1520	1521	1522	1523	1524	1525	1526	1527	1528	1529	1530	1531	1532	1533	1534	1535	1536	1537	1538	1539	1540	1541	1542	1543	1544	1545	1546	1547	1548	1549	1550	1551	1552	1553	1554	1555	1556	1557	1558	1559	1560	1561	1562	1563	1564	1565	1566	1567	1568
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------