SIBO 'C' Software Development Kit

ADDITIONAL SYSTEM INFORMATION

Version 2.00

December 21, 1993

(C) Copyright Psion PLC 1990-93

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 and Psion Series 3a are trademarks of Psion PLC.

TopSpeed is a registered trademark of Clarion Software 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 Mclink, Mcprint, and Slink 1

Mclink.exe 1
Commands provided in MCLINK 1
MCLINK and single floppy disk drive PCs 2
Exiting the MCLINK program 2
Display the version of MCLINK 2

MCLINK file-handling commands 2
Rules on filenames 2
DIR 3
COPY 3
RENAME 4
DELETE 4
MKDIR 4

Changing MCLINK communications settings 4
Options for the SET command 4
Serial port and Baud rate options 5
Modem options 5
Examples of the SET command

Advanced use of MCLINK 5
Running programs remotely on the MC, HC or Series 3 5
MCLINK batch files 6
MCLINK command line processing 6
Invoking MCLINK inside an MS-DOS batch file 6

MCLINK and modems 6
MCLINK as a PC file server via the phone system 7
MCLINK and modem Baud rates 7
Link on the MC/HC as a requestor via modem 7
Link and modem Baud rates 8
Link/MCLINK with MNP 8

Examples with modems 8
Using a Dacom QuadPlus MNP 5 compressing modem 8
Using an Amstrad SM2400 modem 8
Using a Dowty Quattro SB2422 9
Using a WorldPort 1200 pocket modem 9
An HC with a Psion Quad modem 9
An HC with the Amstrad SM2400 modem 9

Mcprint.exe 9
Using MCPRINT 9
Exiting MCPRINT 9
Printer configuration on the MC 10
Printer configuration on the Series 3 10
Parameters 10
The <prdev> parameter 10
The -c<port> parameter 10
The -t<timeout> parameter 11
The -q parameter 11

Slink.exe 11

2 Resource Files 13

Introduction 13
Some uses of resource files on Sibo computers 14

Format of Sibo resource files 15
The format of .rsc files 15
Some strategies for reading .rsc files 15
Example of reading resource files directly 16

Using the rscfile class in Olib 16
Basic services of the rscfile class 16
Reading compressed resource files with the rscfile class 17
Initialising an rscfile object 17
Which header files are needed 18
Run-time errors with the rscfile class 18
Possible errors during initialisation 18
Errors during rs_read or rs_read_buf 18
How rscfile errors are reported 19
Dealing with errors in rs_read or rs_read_buf 19

Advice on where to locate resource files 21
Mono-lingual applications 21
Multi-lingual applications 21
Copying of applications 22

General comments on multi-lingual applications 22
The basic principle of independence of code from resource file 22
Careful design of screen layout 23
Codesize problems 23
Varying keyboards 23
Conclusion 23

Creating .rsc files using rcomp.exe 23
Generated .rsg files 24
The syntax of the rcomp command 24
Include files within a resource script 25
Conditional compilation in resource files 25

Contents of .rss files 25
Declaring STRUCTs 26
Possible member types in STRUCTs 26
Declaring RESOURCEs 27
Declaring the values of sub-STRUCTs 27
Leading byte and word length values 28
Arrays within resource files 29
Creating SYSTEM resource files 30

3 WDR Printing 31

Introduction 31
Creating .wdr files 31
The WDR printing environment variables 31
A note on reading environment variables 32

Overview of the contents of a .wdr file 32
Deciphering general.wdr 33
The header resource 33
The commands resource 34
Model resources 34
Typeface resources 35
Translates resources 35
Summary of resource types in a .wdr file 35

More details on the contents of .wdr files 36
Possible wdr-flags 36
The notion of "printer models" 36
Overview of the different command strings 36
Special characters in command strings 37
The MOVE_RIGHT commands 37
Printer units 37
Possible model-flags 38
Typefaces and fonts 38
Typeface numbers 38

Possible typeface-flags 39
Heights of fonts 39
Widths of characters in fonts 39

Creating .wdr files using wdtran.exe 40
Contents of .wd files 40
Acceptable syntax within a COMMANDS resource definition 41
Acceptable commands within a TRANSLATES resource definition 41
Acceptable commands within a WIDTHS resource definition 42
Acceptable commands within a TYPEFACE resource definition 42
Acceptable commands within a MODEL resource definition 42
Allowed typeface numbers 43

4 DBF files 45

Introduction 45

Basic structure of DBF files 45
The standard header 46
The extended header 46
The field information record 46
The format of all records 46
Deleted records 47
Descriptive records 48
More on type 1 records 48

The Series 3 Database 48
Field information record 48
Extended header 48
Descriptive record 48
Flags options for the Series 3 Database 50
Type 1 records 50
Continuation sub-fields 50

The Series 3 Agenda 50
Field information record 50
Extended header 50
Descriptive record 51
Type 1 records 51
Calculating with AlarmTime 51
The text of an appointment 52
ToDo items 52
Repeat items 52

5 Series 3a Agenda File Format 55

Introduction 55

Basic structure of Agenda files 55
The standard header 55
The extended header 56
The data records 56

Record Types 56

Type 0 - deleted record 56

Types 1 to 4 - entry records 57
Entry details field 57
The title field 59
The alarm field 60
The memo field 60

Type 5 - repeats 60
Type 6 - anonymous data 62
Types 7 and 8 - reserved 62
Type 9 - to-do list information 62
Types 10 to 14 - descriptive records 62
Type 10 - styles descriptive record 63
Type 11 - to-do manager descriptive record 63
Type 12 - frequently changing data descriptive record 63
Type 13 - general descriptive record 63
Diamond list setup field 64
Day entry defaults field 64
Anniversary entry defaults field 65
General defaults field 65
Day view settings field 66
Week view settings field 66
Year view settings field 66
To-do view settings field 66
Anniversary view settings field 67
List view settings field 67
Type 14 - print setup descriptive record 67
Type 15 - illegal 67

6 Word Processor File Format 69

The document header 69
Record types 70
The options data record (type 1, length 10) 70
Printer-related data record (type 2, length 58) 70
Printer model data record (type 3, variable length) 72
Page header record (type 4, variable length) 72
Page footer record (type 5, variable length) 72
Style data record (type 6, length 80) 73
Emphasis data record (type 7, length 28) 74
Document text record (type 8, variable length) 74
Document index record (type 9, variable length) 75

7 Writing Device Drivers 77

Introduction 77
The location of device drivers 78
Device Driver Names 78
Device Driver Channels 78
Searching for PDDs 79
Device Driver Hierarchies And Attached Drivers 79
Interrupts and Interrupt Service Routines 80
Device Driver I/O Semaphore Waithandlers 80
Loadable Logical Device Driver Structure 81
Single Code Segment 81
The LibEnt Structure 81
Mandatory LDD Functions 82
DevFuncInstall 82
DevFuncRemove 83
DevFuncHold 83
DevFuncResume 85
DevFuncReset 85
DevFuncUnits 86
DevFuncOpen 87
DevFuncStrategy 89

Loadable Physical Device Driver Structure 90
Single Code Segment 90
The LibEnt Structure 90
Mandatory PDD functions 91
DevFuncInstallPDD 91
DevFuncRemovePDD 92
DevFuncOpenPDD 93
DevFuncStrategyPDD 93

8 Example Device Drivers 95

An Attached Device Driver Example 95
The device table 95
The Install Function 95
The Remove Function 95
The Hold Function 95
The Resume Function 95
The Reset Function 95
The Units Function 96
The Open Function 96
The Strategy Function 96
The Wait Handler Function 96

Non Interrupt Based Sound Driver 97
The device table 97
The Install Function 97
The Remove Function 97
The Hold Function 97
The Resume Function 98
The Reset Function 98
The Units Function 98
The Open Function 98
The Strategy Function 98
The Wait Handler Function 98
Exercising the vectors 99

Interrupt Driven Sound Driver 99
The device table 99
The Install Function 99
The Remove Function 99
The Hold Function 100
The Resume Function 100
The Reset Function 100
The Units Function 100
The Open Function 100
The Strategy Function 100
The Wait Handler Function 101

9 PCMCIA cards and SSDs 103
Mobility and robustness 103
Size considerations 103
Different standards for different purposes 103
Hot insertion 104
Flash filing systems 104
Cost considerations 104
Architectural openness 104

CHAPTER 1

MCLINK, MCPRINT, AND SLINK

The directory *\sibosdk\sys* contain the following programs (amongst others), all of which can be run on a PC connected to a SIBO computer:

mclink.exe	a program allowing file transfer and remote file access between the PC and the SIBO computer
slink.exe	a "no frills" server-only version of *mclink.exe*, which may run on PCs or PC-lookalikes that cannot run *mclink*
mcprint.exe	a program for printing "through" a PC to an attached printer.

Basic information on connecting a PC with either an MC computer, a Series 3 computer (see note below), or an HC computer, is given in, respectively, the MC Operating Manual, the Series 3 User Guide (see note below), and the *Introduction* chapter of the *HC Programming Guide* (part of this SDK). The information in this chapter gives some more advanced details on the above three programs.

Note: throughout this chapter a reference to the Series 3 machine is taken to include both the Series 3 and Series 3a machines unless explicitly stated otherwise.

Mclink.exe

MCLINK requires MS-DOS version 3.2 or above. MCLINK is unlikely to run inside "DOS emulations" provided by other operating systems (though it happily runs inside MS-DOS tasks inside MicroSoft Windows).

If you experience any problems running MCLINK on your PC, you should experiment with reduced contents of *autoexec.bat* and *config.sys* files. Serial mouse cards may be particularly prone to interfere with the operation of MCLINK.

If all else fails, you may wish to use the alternative SLINK program, also supplied on the PC MCLINK disks.

Note that, by default the HC and Series 3 run at 9600 Baud, the MC and Series 3a at 19200 Baud. When a PC running MCLINK is connected to an MC, either the PC or MC end will in general have to be changed to enable a link to be established.

Commands provided in MCLINK

When you start up the MCLINK program, the lower window contains a $ prompt. At this prompt you can enter various commands. These commands cover:

- file-handling
- changing the communications settings
- exiting the MCLINK program
- displaying the version of MCLINK
- running programs on the remote computer.

For all these commands:

- the command can be abbreviated, to a minimum of the first two letters, eg DE for DELETE, RE for RENAME, CO for COPY, and SE for SET
- CTRL-C stops the command (though in multiple file operations, some files may already have been copied, deleted etc, before the command is stopped).

MCLINK and single floppy disk drive PCs

If your PC has only one floppy disk drive (currently referenced as "A:"), and you mistakenly enter DIR B: while in the MCLINK program, the program will be halted by MS-DOS, requesting you to insert a disk into B:.

To avoid this, use the MS-DOS ASSIGN command before running the MCLINK program, like this:

```
ASSIGN B=A
```

Then DIR B: will be read as DIR A: and the program will not be halted. See your MS-DOS manual for further details of the ASSIGN command.

Exiting the MCLINK program

Type EXIT to return to MS-DOS.

Display the version of MCLINK

Type VER to display the version number.

MCLINK file-handling commands

For file transfer operations between a PC and a SIBO computer, you would normally use the File Manager on the MC or various file options on the Series 3. For certain purposes, however, you might choose to use the file-handling commands within MCLINK on the PC instead.

Rules on filenames

In order to make full use of the file-handling commands of MCLINK, various rules about filenames need to be appreciated.

In MCLINK the syntax of full filenames on the PC, MC, HC or Series 3 is:

```
filing system::device:\directory\sub-directory\file.extension
```

This is very similar to MS-DOS, except for the <filing system> prefix.

If you do not specify a filing system in a file specification, LOC:: is presupposed.

In MCLINK on the PC, <filing system> is

- LOC:: for files on the PC (local)
- REM:: for files on the MC, HC, Series 3 or Series 3a (remote).

On the MC, HC or Series 3 the situation is reversed, with LOC:: for files on the MC, HC, Series 3 or Series 3a, and REM:: for files on the PC.

Note that for all MCLINK file management commands:

- if no directory is specified, the current directory is assumed
- if no device is specified, the current device is assumed
- if no filing system is specified, the PC is assumed.

When specifying a directory or sub-directory in the file-handling commands, make sure to add a \ onto the end of the directory name. Otherwise the directory name will be taken as a filename. So:

`COPY A:\*.* REM::\LETTERS`	is wrong - it would try to copy the files in the root directory of A: to the file LETTERS

`COPY A:\*.* REM::\LETTERS\` is right - it would copy the files on `A:` to the directory `LETTERS`.

DIR — list files in specified directory

`Syntax: DIR filespec`

The directory specified may be on the remote or local filing system - eg `DIR A:\LETTERS\` looks in directory `LETTERS` on the disk in drive `A:` of the PC, `DIR REM::A:\NOTES\` looks in the `NOTES` directory on the SSD in drive `A:` of the remote machine.

Use wildcards to list only certain files - eg `DIR *.TXT` to list just the `.TXT` files in the current directory.

When you get a directory listing with the `DIR` command, the following file information is given:

- file name and extension
- date and time when the file was last modified
- size of file in bytes

and a combination of these indicators as appropriate:

`Mod` file has been modified since last backed up

`RdO` read-only file

`Sys` system file

`Hid` hidden file

Example: `CLIENTS.DBF 14/01/90 09:54:23 544 Mod RdO`

COPY — copy files to specified device and directory

`Syntax: COPY filespec1 filespec2`

Optional flags:

`-i` include sub-directories: If there are any files copied from subdirectories they are placed in directories below the current directory, to reflect the source directory structure

`-m` modified files only, eg `COPY REM::A:\*.* -i -m` copies all modified files in all directories of the SSD in drive A: on the remote machine to the PC.

Examples:

`COPY REM::M:\*.TXT \BACKUP\` would copy all the `.TXT` files from the internal disk of the remote machine to the `BACKUP` directory on the PC

`COPY \SMITH\*.DOC REM::A:\LS\` would copy all the `.DOC` files from the `SMITH` directory on the PC to directory `LS` on the SSD in drive `A:` of the remote machine.

If you do not supply a complete destination name, the root directory and/or default disk on the remote machine is assumed, and the source filename is used as the destination filename. Eg

```
COPY \HOME\SECURE.TXT REM::M:
```

would copy `SECURE.TXT` to the `M:\` directory on the remote machine, giving the file the name `SECURE.TXT`.

If you do not supply a complete source name, the current device/directory is assumed. Eg

```
COPY *.TXT REM::M:\
```

copies files from the current directory on the PC to the root directory of the remote machine's internal disk.

Note: if you are transferring a lot of small files to the Series 3, it is a good idea not to stay in the Series 3 System Screen. This could take longer than usual because the System Screen would continually update its file lists as files arrived. In extreme cases, with lots of very small files, the file transfer could even fail. So press an application button, such as the TIME button.

RENAME — rename specified files

Syntax: RENAME filespec1 filespec2

You can rename a file in any directory on any drive on the PC, MC, HC or Series 3. For example

```
RENAME DETAILS.DOC DETAILS2.DOC
```

```
RENAME REM::M:\CLIENTS.DBF REM::M:\BUSINESS.DBF
```

You can rename more than one file at a time. For example

```
RENAME REM::M:\*.TXT REM::M:\*.DOC
```

You cannot rename a file across directories or devices. Thus

```
RENAME REM::M:\LETTER1.TXT REM::B:\LETTER2.TXT
```

would give an error. Instead, copy the file to the new name and destination then delete the old file.

DELETE — delete specified files

Syntax: DELETE filespec

Optional flag:

`-i` delete files of the same name in sub-directories, eg `DEL *.TXT -i` would delete all `.TXT` files in the current directory and in any sub-directories of the current directory.

You can delete files from any directory on the PC, MC, HC or Series 3.

You can delete more than one file at a time by using wildcards.

You cannot delete directories with this command.

MKDIR — make a directory

Syntax: MKDIR directory

Makes a subdirectory of the current directory.

You can make directories on any drive of the PC, MC, HC or Series 3.

You can make more than one subdirectory at a time - eg `MKDIR \HOME\LETTERS` makes the subdirectory `\HOME\LETTERS` and also the intermediate subdirectory `\HOME` (if it does not exist).

Changing MCLINK communications settings

The `SET` command in MCLINK allows you to:

- use either of the PC's serial ports
- change the Baud rate which the PC uses
- use a modem (see later in this chapter for more details of using MCLINK over a modem)

The `SET` command creates a new *MCLINK.TRM* in the current directory to hold the new settings. The next time you run MCLINK from this directory, these settings will be loaded again.

Options for the SET command

Following the `SET` command you can specify a variety of options. For example:

`SET -p1 -b9600` selects `COM1`, 9600 Baud

`SET -p2 -b9600` selects `COM2`, 9600 Baud

The full range of options for the `SET` command includes `-p`, `-b`, `-m`, `-n`, and `-c`.

These options are also available as parameters to the command line for MCLINK (see later), but in this case, no permanent record of the options are made in any *.TRM* file.

Serial port and Baud rate options

The SET options -p1 or -p2 select COM1 or COM2.

The option -b followed by a number sets the Baud rate. You will need to set the Baud rate on the MC, HC or Series 3 to match that on the PC.

IMPORTANT: In general you should specify *both* -p and -b, *or neither*. If you specify just one, the other is reset to MCLINK's internal default value. The internal default for the Baud rate is 19200. (By default MCLINK runs at 9600 Baud since when starting up it looks for a file called *MCLINK.TRM* which is built in to *MCLINK.EXE*. This file sets the PC to COM1 at 9600 Baud.)

Modem options

The SET options -m or -n specify that you are using a modem (as described in more detail later in this chapter).

The option -m means wait for a call; MCLINK will establish the speed to use to the modem.

The option -n followed by a number causes MCLINK to dial the number. The modem is assumed to conform to the Hayes command set. Here you can if you wish specify the speed for MCLINK to use, with -b. For example: -b2400 -n314159 dials 314159 at 2400 Baud.

The option -c<string> will cause <string> to be transmitted to the modem to configure it before waiting/dialling:

SET -cATM0	turns off the modem speaker
SET -cAT\N3	sets a Dacom modem to MNP fallback mode.

Note that the AT is optional in these commands. Multiple strings can be transmitted. For example,

```
SET -cM0 -c\N3
```

Examples of the SET command:

SET -b1200	1200 Baud
SET -p2 -m	waits for a call using the modem in port 2 (COM2)
SET -p1 -b2400 -n314159	modem connected to port 1 dials the phone number 314159
SET -cm0	turns off the modem speaker.

Advanced use of MCLINK

Running programs remotely on the MC, HC or Series 3

The syntax

```
RUN <program name>, <program command line>
```

causes the named program to be run on the remote computer, with the specified command line.

The program is assumed to exist in the default directory of the remote machine or the root directory of any drive on the remote machine. Otherwise the ROM of the remote machine will be searched for the program.

The program command line is as required by the program being invoked.

For example,

```
RUN CLOCK.IMG
```

will run a copy of CLOCK.IMG on the remote machine.

As an accelerator for running a copy of the remote shell program on HC machines the '!' command is equivalent to typing RUN SYS$SHLL.

For example:

```
! DIR
```

will run a copy of the SYS$SHLL program passing an initial command line of DIR. See the *HC Programming Guide* for further information on commands available to the SYS$SHLL program.

MCLINK batch files

MCLINK can read a text file containing multiple commands:

- Any line in the file containing a '!' is assumed to be a comment line and is ignored.
- Any blank line is ignored.
- You may specify almost any MCLINK command, although some may be meaningless in this environment - DIR, for example.
- The SET command should not be used.

To invoke, a batch file, type the '@' character immediately followed by the name of the text file containing the commands. For example, if the file *SEND_ALL.TXT* contains the following lines:

```
! Sends all text files to the remote machine after
! creating the correct directory, then exits
MKDIR REM::M:\NOTES\
COPY *.TXT REM::M:\NOTES\
COPY *.DOC REM::M:\NOTES\
EXIT
```

then entering @SEND_ALL.TXT at the '$' prompt will cause the \NOTES\ directory to be created in the internal memory of the remote machine, all the .TXT and .DOC files in the current directory to be copied to this directory, then MCLINK to exit.

MCLINK command line processing

MCLINK understands a command line entered when running MCLINK from the MS-DOS prompt. The command line may take the form of the parameters to the SET command, a filename assumed to be a configuration file, or the @ command to run a sequence of commands.

Examples:

`MCLINK -p2 -b9600`	will run MCLINK using port 2 at 9600 Baud. No *.TRM* file will be created, unlike using the SET command, and any future running of the MCLINK program will not use these parameters.
`MCLINK S3SETUP`	will run MCLINK forcing it to use the file *S3SETUP.TRM* as the initial configuration file. This file would typically have been created by a previous use of the SET command within MCLINK.
`MCLINK @SEND_ALL.TXT`	will run MCLINK and cause the initial commands to be read from the file *SEND_ALL.TXT*. MCLINK will wait until a connection to the MC, HC or Series 3 has been established before running any of the commands.

Invoking MCLINK inside an MS-DOS batch file

Perhaps the most convenient way to automate a regular MCLINK task is via an MS-DOS batch file.

For example, a batch file *SEND_ALL.BAT* could contain the single line

```
MCLINK @SEND_ALL.TXT
```

in which case the contents of *SEND_ALL.TXT* would be performed simply by typing

```
SEND_ALL
```

from the MS-DOS command line.

MCLINK and modems

A PC running MCLINK can use one modem at one end of a telephone line to connect to an HC or MC computer attached to another modem at the other end of the telephone line.

Modem communication of this sort is possible only for MC and HC computers. The 3 Link software for the Series 3 does not have any built-in modem support for communicating over a telephone line to a PC running MCLINK. (The HMD: device driver is not present in the ROM of the Series 3.) To communicate over a modem using a Series 3, use the Script language instead (which is included with the 3 Link software for the Series 3).

MCLINK as a PC file server via the phone system

When used as a PC file server via the phone system, MCLINK assumes that your modem follows CCITT rules and regulations concerning the RS232 signals - ie

- The modem drives DSR when powered up, never drops DSR and does not use DSR for any sort of handshaking. If the modem is physically removed, MCLINK detects this as the DSR signal disappears. If DSR is not driven MCLINK will not talk to the modem as it does not think it is there.
- The modem responds to the DTR signal in the following manner - when MCLINK drives DTR low, (off, inactive) the modem should reset itself - ie disconnect if online etc and eventually enter its command mode. When MCLINK is exited, it drives DTR low to disconnect any calls currently connected. When MCLINK is started up it drives DTR low for 2 seconds to try to force the modem into its command mode, at its default settings.
- The modem only drives DCD when it is on line to a remote modem, and drops DCD when the connection with the remote modem is lost.

When MCLINK is asked to communicate with a modem it sends the 'AT' command string to the modem at the following Baud rates: 300, 600, 1200, 2400, 4800 and 9600. It monitors the response to sending this command, and sets itself to the highest speed at which an "OK" reply was received.

MCLINK then sends the following command stream to the modem to configure it:

"ATX4E0S0=1"	if the modem's maximum Baud rate was 2400 Baud or above
"ATX1E0S0=1"	otherwise.

MCLINK then reads the user command configuration strings passed to it and sends them to the modem. All commands sent to the modem are checked for validity by waiting for the modem to respond to the command sent. If the 'OK' response is received the command worked, otherwise an error is reported. This will result in MCLINK re-starting.

NOTE: If the user command stream contains any form of reset, then the auto answering of calls should be re-enabled explicitly.

MCLINK and modem Baud rates

When MCLINK displays the status message *Waiting for an Incoming Call* the Baud rate displayed will be the fastest Baud rate that MCLINK found the modem supported. Setting the Baud rate is really a meaningless exercise since the Baud rate at which the modem connection is made is determined by the Baud rate of the dialling modem (originator) and not the modem accepting the call.

If MCLINK detects the fastest Baud rate its modem can handle is 2400 Baud or above, it assumes that the modem has the ability to provide a constant speed interface - ie the modem does not change to the originator's Baud rate as soon as a connection is established. All modems that can handle Baud rates of 2400 and above *must* have the constant speed interface since this is the way in which MNP throughput is normally achieved.

If the fastest Baud rate is below 2400 Baud then MCLINK will set the connection Baud rate to that reported by the modem when it connects - ie the modem is assumed to change to the originators Baud rate and MCLINK will follow it. Modems in this class will not support any form of data compression/correction since a higher Baud rate than the connection Baud rate is required to achieve data compression.

Link on the MC/HC as a requestor via modem

(This section closely matches the corresponding section above for MCLINK.)

When used on the HC or MC as a requestor via the phone system, Link software on the HC/MC assumes the modem follows CCITT rules and regulations concerning the RS232 signals - ie

- The modem drives DSR when powered up, never drops DSR and does not use DSR for any sort of handshaking. If the modem is physically removed, Link detects this as the DSR signal disappears. If DSR is not driven Link will not talk to the modem as it does not think it is there.

- The modem responds to the DTR signal in the following manner - when Link drives DTR low, (off, inactive) the modem should reset itself - ie disconnect if online etc and eventually enter its command mode. When Link is exited, it drives DTR low to disconnect any calls currently connected. When Link is started up it drives DTR low for 2 seconds to try to force the modem into its command mode, at its default settings.
- The modem only drives DCD when it is on line to a remote modem, and drops DCD when the connection with the remote modem is lost.

When Link is run it sends the `'AT'` command string to the modem at the following Baud rates: 300, 600, 1200, 2400, 4800, and 9600. It monitors the response to sending this command, if it sees an `'OK'` it assumes the modem can be driven at that Baud rate.

Link then sends the following command stream to the modem to configure it:

`"ATX4E0S0=1"` if the modem's maximum Baud rate was 2400 Baud or above

`"ATX1E0S0=1"` otherwise.

Link then reads the user command configuration strings and sends them to the modem. All commands sent to the modem are checked for validity by waiting for the modem to respond to the command sent. If the `'OK'` response is received the command worked, otherwise an error is reported. This will result in Link re-starting.

Link and modem Baud rates

Link will send the dial string to the modem at the Baud rate specified from the Link dialog, or in the case of the HC at the Baud rate specified in the command line. If no Baud rate is specified, the fastest Baud rate that the modem responded to (see above) is used. By sending the dial string at the specified Baud rate the particular type of Vxx connection will be established - eg at 2400 a V22bis, at 1200 a V22, and at 300 a V21 connection. Note that a V23 (1200/75) connection cannot be used.

Note: if you want to use your HC as the file server and the PC as the requestor, swap the above instructions for 'MCLINK' and 'Link'.

Link/MCLINK with MNP

If you have MNP modems do not be surprised if the data transfer rate between your PC and MC/HC is lower. This is because of the way MNP works.

Typically it is not worth having MNP enabled. The protocol used by Link and MCLINK is based heavily on the MNP protocol, ie it provides an error free connection between the PC and HC.

Examples with modems

The first four examples below focus on a PC with MCLINK as a file server. The last two examples focus on an HC with Link software.

For more information on any of the configuration strings see the appropriate modem manuals.

Using a Dacom QuadPlus MNP 5 compressing modem

Run MCLINK with the following command line:

```
MCLINK -C&F&C1&D3\N3\J1S0=1
```

Alternatively, you could set up a *.TRM* file, eg *QUADPLUS.TRM* and type

```
MCLINK @QUADPLUS.TRM
```

or call the *.TRM* file *MCLINK.TRM* and just type

```
MCLINK
```

Using an Amstrad SM2400 modem

Run MCLINK with the following command line:

```
MCLINK -m
```

Using a Dowty Quattro SB2422

Run MCLINK with the following command line:

```
MCLINK -m
```

Using a WorldPort 1200 pocket modem

Run MCLINK with the following command line:

```
MCLINK -m
```

An HC with a Psion Quad modem

Run Link with the following command line:

```
LINK -n<the phone number> -c\n0
```

This talks to *all* of the above PC file server configurations.

An HC with the Amstrad SM2400 modem

Run Link with the following command line:

```
LINK -n<the phone number>
```

Mcprint.exe

MCPRINT allows an MC, HC, or Series 3 (or any other serial-printing device) to print to a printer which is connected to an IBM PC/XT/AT or compatible. The PC must have a free serial port which is used to connect to the MC/HC/Series 3. The printer may be connected to a parallel or serial port on the PC.

Even if you can link the MC/HC/Series 3 directly to the printer, there may be reasons why it is more convenient to use MCPRINT:

- The printer is shared by other PCs - either using a multi-port printer buffer or a local area network - and it would be unreasonable to connect it directly to the MC/HC/Series 3.
- You do not wish to disturb the connection between the PC and the printer.
- You have already set up the serial connection to use MCLINK for file transfer to the PC.
- When your PC is connected to more than one printer, you can select which one the MC/HC/Series 3 will use.

Note: it is also possible to print via MCLINK to a printer attached to your PC. To do this, set your MC/HC/Series 3 to print to a file, and give the printer device on `REM::` (such as `REM::LPT1`) as the "file" to use. This method may be slower than using MCPRINT - especially when printing "justified" text from the Word Processor on either computer - and will only work reliably on version 3.0 or above of MCLINK. However, MCLINK can correct transmission errors, whereas MCPRINT can only report them. Such errors may occasionally be caused by PC add-ons, such as some network card drivers.

Using MCPRINT

Physically connect the MC/HC/Series 3 to the PC exactly as for MCLINK. If the printer is connected to `LPT1`, and MC/HC/Series 3 is connected to `COM1`, you can now run MCPRINT on the PC by typing:

```
MCPRINT
```

If `LPT1` and `COM1` are not the ports used, parameters are required, as described below.

Exiting MCPRINT

To exit MCPRINT, press CONTROL-C.

Printer configuration on the MC

On the MC, select *Print Setup* from the *Options* menu in the System application to display the Printer dialog. Set one of the configurations to output to Serial. You don't normally have to click on the SET SERIAL... button to change any of the Serial options because MCPRINT uses the default settings.

Once the current configuration has been set to Serial, you print as if the printer was directly connected to the MC - by selecting the *Print* menu item in any application which can print. As far as the MC software is concerned, it is printing to a Serial printer.

Printer configuration on the Series 3

On the Series 3, select the *Printer setup* option from the *Special* menu in the System screen. In this dialog, set the *Printer device* line to *Serial*. You don't normally have to change the *Serial characteristics* (it displays a subdialog when you press TAB) because MCPRINT uses the default settings.

You can now print as if the printer was directly connected to the Series 3 - by selecting the *Print* option in any application which can print - including the built-in Word Processor, Agenda, Database, and Program editor. (Remember first to use the *Print setup* options in these applications, to tell the Series 3 about the type of printer and the page layout desired.) As far as the Series 3 software is concerned, it is printing to a serial printer.

Parameters

MCPRINT takes the following parameters:

```
<prdev> -c<port> -t<timeout> -q
```

all of which are optional. To be reminded of these parameters, type:

```
MCPRINT ?
```

The <prdev> parameter

<prdev> is the print device name, as for the MS-DOS PRINT command. If omitted, the default is LPT1 (the MS-DOS name for the first parallel port).

For example, to print to LPT2, type:

```
MCPRINT LPT2
```

If the PC is a station on a local area network, LPT2, LPT3 etc may be used to redirect output to remote printers attached to the network server.

The <prdev> parameter may specify any suitable output device. If you have a printer connected to a second serial port on the PC, you can use:

```
MCPRINT COM2
```

In this case, you should have previously used the MS-DOS MODE command to set the serial parameters to be used between the PC and the printer.

You can also print to a file on the PC using, for example:

```
MCPRINT PRINT.LIS
```

Note that *PRINT.LIS* will be overwritten each time you print.

To test the connection without wasting paper, type:

```
MCPRINT CON
```

CON is the MS-DOS name for the console (screen). When you then print from the MC/HC/Series 3, you should see the output appear on the screen of the PC.

The -c<port> parameter

<-c> is the serial port on the PC to which the MC/HC/Series 3 is connected. If omitted, the default is port 1, which corresponds to COM1. If the MC/HC/Series 3 is connected to the PC's second serial port, type:

```
MCPRINT -C2
```

The -t<timeout> parameter

<timeout> is a number of seconds. This parameter is provided for use on local area networks where it is necessary to close and open the print device between each print job. It is used to specify an inactivity time-out in seconds. If there is no printing for this period, the print device is automatically closed. For example:

```
MCPRINT LPT2 -T5
```

will close the print device after 5 seconds of inactivity.

If the parameter is omitted, the print device is not automatically closed. You don't have to specify a time-out, as you can close the print device manually by pressing any key on the PC keyboard. Exiting MCPRINT will also close the print device.

The MC/HC/Series 3 are multi-tasking - while one application is printing, you can carry on with something else. However, the background printing can be held up at times, depending on the processing requirements of the work you are doing. This can fool MCPRINT's inactivity time-out into thinking that the printing has finished, causing it to prematurely close the print device. If you experience this problem, consider increasing the time-out, or convert to closing the printer device manually by pressing a key on the PC keyboard.

The -q parameter

This parameter suppresses status messages (the 'q' stands for "quiet").

Slink.exe

SLINK is a "no-frills" server-only version of MCLINK.EXE. However, it may run on "PC"s which are less than 100% PC-compatible and cannot run MCLINK, and it may run in combination with other software which conflicts with MCLINK.

By default, SLINK uses the COM1 port, at 9600 Baud. You can specify on the command line the Baud rate and the serial port to use. These are in the same format as in the SET command in MCLINK. For example:

```
SLINK -p2 -b9600
```

This sets SLINK to use COM2. Note that, as with the SET command in MCLINK, -b9600 is used in this example to keep the Baud rate at 9600.

If you just type SLINK -p2 this will reset the Baud rate to the internal default of 19200.

Press Q to quit SLINK.

SLINK has no support for modems.

CHAPTER 2

RESOURCE FILES

This chapter covers a range of related topics:

- reasons programmers might consider using resource files
- the format of Sibo *.rsc* resource files
- the Olib `rscfile` class that can be used to read resource files
- the Sibo resource compiler tool, *rcomp.exe*, that can be used to create resource files
- general considerations about multi-lingual applications.

Although the topics are all related, it is by no means necessary to read and understand all the sections in this chapter, just in order to understand one of these sections.

Introduction

There are two main reasons why a programmer may wish to use resource files:

- Having data in a resource file, rather than as part of the program itself, cuts down on the size of the data segment required by the program, and thus makes more efficient use of RAM.
- Resource files make it easier to write applications that can run in more than one language (eg English, French, German...).

Text strings are a simple but important example of data that can be stored in a resource file. Suppose a program contains lines of code such as

```
wInfoMsg("Starting calculation");
```

and

```
wSetBusyMsg("Scanning");
```

or even

```
p_printf("%d items found",num);
```

The dataspace of this program, when compiled and linked, would contain the three strings `"Starting calculation"`, `"Scanning"`, and `"%d items found"` - a grand total of some 45 bytes (note that a terminating zero is stored for each string). A larger program may have many times this number of data strings; up to 2k would not be uncommon. Now this data would be *permanently loaded into RAM* all the time the program is running. As a result, 2k less space would be available to the ordinary data of the program - such as cells in a spreadsheet, or text in a word processor - thus reducing the amount of such data that the program can accept before giving an "out of memory" error. Moreover, the operating system would be more likely to refuse to load and run the program, on account of insufficient memory being available to start it.

Next consider how the problem worsens for a program that is to be translated into more than one language. Either the program has to carry the data for all the different target languages, with a choice being made at run time between the various different possibilities:

```
if (language==LANG_ENGLISH)
    wSetBusyMsg("Scanning");
else if (language==LANG_FRENCH)
    wSetBusyMsg("Parcourt");
else if (language==LANG_GERMAN)
    wSetBusyMsg("Suche");
...
```

or else the code will have to be recompiled each time for a new language. But this latter approach makes the problem of *maintaining code* much harder; each different change made to the code, such as a bug fix, will have to be propagated to all the different language versions. At the same time, the job of the translator is not helped by the text to translate being all mixed up with the rest of the code, whilst if the translator works on a separate list of text strings, there is the risk of transcription errors when the separate lists are merged back into the code.

For reasons such as these, serious programming in *any* system (Sibosdk or otherwise) frequently adopts one or other resource file approach for text strings and other data items. The strings `"Starting calculation"`, `"Scanning"`, `"%d items found"`, and so on, are kept in a separate file, not as part of the dataspace of the program, and are loaded into RAM only when they are needed.

Thus the above call

```
wInfoMsg("Scanning");
```

would be replaced by a call such as

```
InfoMsg(RESOURCE_SCANNING);
```

where `RESOURCE_SCANNING` is a symbolic constant (`#define`) giving the *index* of the text string `"Scanning"` in the resource file. (The exact meaning of the index varies between different resource file schemes. See below for the meaning in Sibo resource files.)

The contents of the routine `InfoMsg` would be something like

```
LOCAL_C VOID InfoMsg(INT index)
    {
    TEXT buf[60];

    LoadResourceString(&buf[0],index);
    wInfoMsg(&buf[0]);
    }
```

and in turn `LoadResourceString` would read data from the appropriate resource file.

At the initialisation of the program, the name of the appropriate resource file would be determined, once and for all, by reference to the current language (as obtained by a call to `p_getlanguage`).

Some uses of resource files on Sibo computers

Each of the built-in or bundled applications on the MC and Series3 ranges has its own resource file, in which are kept menu and dialog data, as well as more basic text strings. The dialog data can contain numerical layout information and numerical flags customising individual items within dialogs.

Since the text strings and dialogs used by these different applications often overlap, there is also a so-called *system resource file*, where common items are kept. Thus an application loads data at various times from each of two different resource files - its own *application* resource file, and the system one.

The *.wdr* printer driver files used by the printer subsystem in *form.dyl* (as on the Series3 - see the *WDR Printing* chapter in this manual for more details) are also resource files, with the data items consisting of escape sequences, font width tables, and other printer data.

Finally, low-level error messages are defined in another file in the ROM, *sys$ctry.cfo*. This file also contains the keyboard layout tables, the fold tables, and other standard text such as the names of the days of the week. See the *Config Files* chapter for more details.

Format of Sibo resource files

There are in fact *three* kinds of Sibo resource files:

- *.cfo* files, which are *language configuration* files (sometimes just called *config files*), with *sys$ctry.cfo* being the principle example
- *.rsc* files, which are *standard resource files*
- *.rzc* files, which are *Huffman compressed* versions of *.rsc* files.

Access to the data in *.cfo* files is via Plib functions such as `p_errs`, `p_gettext`, and `p_nmmon`, as described in the *Plib Reference* manual.

Data in *.rsc* and *.rzc* files can be accessed using the functionality of the `rscfile` class in *olib.dyl*, as described later in this chapter.

The Sibo resource compiler, *rcomp.exe*, can be used to create instances of *.rsc* files from plain text input known as *resource scripts*, which typically have extension *.rss*. This process is also described later in this chapter. However, the format of *.rsc* files (described immediately below) is so straightforward that programmers could easily create their own tools for producing customised *.rsc* files.

Unless explicitly stated to the contrary below, the remainder of this chapter focuses exclusively on the *.rsc* type of resource files.

The format of .rsc files

A standard resource file just containing the three strings `"Starting calculation"`, `"Scanning"`, and `"%d items found"`, has the following contents (when dumped):

```
 0:  31 00 08 00 53 74 61 72  74 69 6e 67 20 63 61 6c   1...Star ting cal
10:  63 75 6c 61 74 69 6f 6e  00 53 63 61 6e 6e 69 6e   culation .Scannin
20:  67 00 25 64 20 69 74 65  6d 73 20 66 6f 75 6e 64   g.%d ite ms found
30:  00 04 00 19 00 22 00 31  00                       .....".1 .
```

This conforms to the pattern:

```
<header><resources><index table>
```

where:

`<header>`	is always four bytes long, with the first word giving the file offset of the start of the index table, and the second word giving the length (in bytes) of the index table
`<index table>`	is a sequence of words, the first giving the file offset of the start of the first resource, the second giving the file offset of the start of the second resource, and so on, up to the last word, which gives the file offset of the *end* of the last resource (which is also the *beginning* of the index)
`<resources>`	are a series of variable length data items, whose contents can have any form.

Some strategies for reading .rsc files

Clearly, one way to implement a routine such as `LoadResource` is essentially as follows:

```
GLDEF_C VOID LoadResource(UBYTE *pb,INT index)
    {
    ULONG fpos;      /* file offset */
    UWORD tmp[2];    /* section of index table */

    fpos=ixpos+((index-1)*2);        /* position into the index table */
    p_seek(fcb,P_FABS,&fpos);
    p_read(fcb,&tmp[0],4);           /* read two words from index table */
    p_seek(fcb,P_FABS,tmp[0]);
    p_read(fcb,pb,tmp[1]-tmp[0]);
    }
```

where:

- `fcb` is the file control block of the open resource file

- ixpos is the value of the first word in the resource file (ie the file offset of the beginning of the index table), and has been read into memory during program initialisation, for the sake of efficiency
- the passed value of index in this case would be 1 for the first resource, 2 for the second resource, and so on
- the code would need modifications to cope with possible error values returned by the p_seek or p_read calls (further discussed below).

This strategy relies on a value of ixpos being stored in program memory. Another strategy would be to store the entire index table in memory - and this is the reason why the *length* of the index table is recorded as the second word in the resource file. However, in practice there is no observable speed degradation on account of reading index table data from the file every time a resource has to be loaded, and so the earlier scheme is generally to be preferred - in view of the lesser demand it places on RAM usage.

Example of reading resource files directly

See the file *readrsc.c* in *\sibosdk\demo* for an example of how to read the contents of a resource file directly (i.e. without using the services of the rscfile class).

This example code assumes that the resource file is embedded in the application's program (*.app*) file. Thus, when used in an application, the name of the resource file should be specified on the second line of the application's *.afl* file.

To use the example code, you must create a resource file with its first two resources both being short text strings. Any additional resources are not read by the supplied code.

Using the rscfile class in Olib

Although it is possible, along the lines discussed above, to read *.rsc* resource files using ordinary Plib function calls, there are various reasons for instead using the functionality of the rscfile class in *olib.dyl*:

- Using the rscfile class avoids needing to remember any details of the format of *.rsc* files
- The rscfile class also contains considerable logic, hidden from the casual user, to decode *.rzc* Huffman compressed resource files - so that a decision can be taken at a later stage, to use *.rzc* format files instead of *.rsc* format, without any need to alter or recompile existing code
- The rscfile class automatically takes care of locating resource files suitably *embedded in a .img* file - see below for more details
- The interface to the rscfile class clarifies and documents all the possible error conditions that need to be catered for
- Learning about the rscfile class is a useful step along the route to learning about Psion's proprietary object-oriented programming system - since this system makes heavy use of the rscfile class.

Basic services of the rscfile class

Before the functionality of the rscfile class can be used in a program, an instance of this class needs to be created and initialised. This is dealt with below.

The outcome of the initialisation is a *handle*, rather like a handle to a file control block or to other i/o device control blocks. Subsequent rscfile services are directed via this handle.

The most primitive rscfile service is to load a resource, specified by index number (starting at 1 for the first resource), into a supplied buffer. This is the rs_read_buf service. In this case, it is assumed that the caller has supplied a sufficiently long buffer.

Sometimes, however, it is more appropriate for the rscfile class to allocate a cell of sufficient length, for the resource to be loaded into. This typically applies when a resource can have variable length, and when the resultant alloc cell will have some permanence. The rs_read service fulfils this requirement.

As an example of the rs_read_buf service, consider the following routine InfoMsg:

```
LOCAL_C VOID InfoMsg(INT index)
    {
    TEXT buf[60];

    p_send4(rcb,O_RS_READ_BUF,index,&buf[0]);
    wInfoMsg(&buf[0]);
    }
```

with rcb being the handle of a suitably initialised rscfile object.

As an example of the rs_read service, consider loading some menubar data in from a resource file. For Hwif programs, this data is (in part) in the form of an H_MENU_DATA struct. The address of this struct has to be written to the static _mdata (of type H_MENU_DATA*) whose existence the Hwif library presupposes. In that case, the following call might be made during the initialisation of an Hwif application:

```
p_send4(rcb,O_RS_READ,MENU_DATA_INDEX,&_mdata);
```

where MENU_DATA_INDEX is a symbolic constant giving the index of the appropriate resource.

Note that although the calling interface to rs_read_buf and rs_read may look similar, they require different types for the penultimate parameter:

- rs_read_buf requires a parameter such as a TEXT* or a UBYTE*, ie with *one* level of indirection from the actual loaded data
- rs_read requires a parameter such as a TEXT** or a UBYTE**, ie with *two* levels of indirection from the actual loaded data.

Barring run-time errors (discussed below), the calls rs_read_buf and rs_read both return the length of the resource read. Note that in the case of a (zero-terminated) string, this length includes the length of the terminating zero, since that is part of the resource too.

Reading compressed resource files with the rscfile class

If a Huffman compressed resource file, typically with extension *.rzc*, is substituted for a standard resource file (typically having extension *.rsc*), there is no need to alter or recompile in any way application code making use of the rscfile class. The interface remains exactly the same.

The only point possibly worth mentioning is that the lengths returned by rs_read and rs_read_buf are the length of the resources *once decompressed*, and not the length of the compressed resources on file.

Initialising an rscfile object

The following code can be used to create and initialise an rscfile object providing access to a resource file with name rscname (assumed to be a full path name):

```
VOID *InitRcb(TEXT *rscname)
    {
    HANDLE OlibCat;
    VOID *rcb;
    INT ret;

    p_findlib("OLIB.DYL",&OlibCat);
    rcb=p_newlibh(OlibCat,C_RSCFILE);
    if (rcb)
        {
        ret=p_entersend3(rcb,O_RS_INIT,rscname);
        if (ret<0)
            p_exit(ret);
        }
    return(rcb);
    }
```

For overtly object oriented programs, the lines

```
p_findlib("OLIB.DYL",&OlibCat);
rcb=p_newlibh(OlibCat,C_RSCFILE);
```

can be replaced by a line such as

```
rcb=p_new(CAT_HWIF_OLIB,C_RSCFILE);
```

with the category number CAT_HWIF_OLIB being replaced by the suitable reference to *olib.dyl* from the native category.

For Hwif programs, the line

```
rcb=p_new(1,C_RSCFILE);
```

can be used instead, taking advantage of the fact that the value of the (normally hidden) symbolic constant CAT_HWIF_OLIB is 1.

Which header files are needed

The symbolic constants C_RSCFILE, O_RS_READ, O_RS_READ_BUF, and O_RS_INIT, are defined in the object-oriented include file *appman.g*.

Duplicates of these definitions are given in the special SDK file *rscfile.xg*.

The value of O_DESTROY (defined in *olib.g*) is 0. (See below for use of O_DESTROY.)

Run-time errors with the rscfile class

Broadly speaking, there are five kinds of run-time error that can arise with resource files:

1 there is insufficient memory to create or initialise the rscfile object

2 the resource file cannot be found (when the program starts)

3 the data in the resource file is bad

4 the SSD containing the resource file is removed or cannot be accessed

5 there is insufficient memory to load a specified resource.

Of these possibilities, the third is regarded simply as a programming error. Thus if some data in what should be the index table part of the file effectively says that a certain resource has length 5398 bytes, whereas the file itself is smaller than this size, the rscfile object will panic the application (with panic number 141).

Otherwise, errors 1 and 2 can occur when *initialising* a rscfile object, whereas error 4 and 5 can occur when subsequently *using* the object.

Possible errors during initialisation

The only reason the calls p_newlibh or p_new in the above routine InitRcb will fail is on account of lack of memory. Applications can choose to discount this possibility if their minimum heap is appropriately calibrated - see below.

The call to rs_init can fail with file-based errors on account of the filename in *rscname. The most pertinent possibility (assuming that a well-formed name has been passed) is that the specified file does not exist. Applications could guard against this by checking on the existence of the file prior to calling InitRcb. If the file does not exist, the user can be notified, and the program exited.

Errors during rs_read or rs_read_buf

The only errors that an application should in practice worry about, for the rs_read and rs_read_buf services, are the fourth and fifth in the above list.

Running out of memory can occur only in the case of rs_read - when it is impossible to allocate a call from the heap large enough to load the resource into. The implementation of rs_read_buf is guaranteed never to fail with out of memory.

If calls to rs_read are made during program initialisation only, it may well be legitimate to ignore the possibility of out of memory errors in this case too - provided the declared minimum heap of the application is large enough. The point is that only memory in the dataspace of the application has to be allocated - not any memory in another process such as the File Server (see the chapter *Fundamental Programming Guidelines* in the *General Programming Manual* for related discussion).

However, applications calling either rs_read or rs_read_buf ought always to consider the possibility of the user removing the SSD containing the resource file. What will happen in this case is as follows:

- suppose the user removes the relevant SSD, not realising (or forgetting) that the program may wish to access data on it
- the application makes a call to rs_read or rs_read_buf
- system code, detecting that the file is missing, presents a Notifier requesting the user to replace the SSD; this Notifier has two exit options: Retry and Fail
- the user ought to replace the SSD and select Retry; however, it is possible that the Fail option will be selected
- in this case, an error such as E_FILE_ABORT will be generated.

How rscfile errors are reported

The rscfile services rs_read and rs_read_buf do not *return* any error values; instead, they internally call p_leave.

Applications performing sophisticated error handling, using p_enter, should be sure that p_send calls to rs_read or rs_read_buf are (ultimately) enclosed in some call to p_enter - otherwise any errors will result in their application being panicked, with panic number 47.

Applications not wishing to use p_enter should replace the above calls to p_send with calls to p_entersend, as follows:

```
ret=p_entersend4(rcb,O_RS_READ_BUF,index,&buf[0]);
```

and

```
ret=p_entersend4(rcb,O_RS_READ,MENU_DATA_INDEX,&_mdata);
```

Possible values of ret that can be returned are as follows:

positive value	the length of the resource loaded (no error has occurred)
negative value	an error has occurred: either E_GEN_NOMEMORY for out of memory, or some other value in case the resource file could not be accessed.

Note incidentally that the complications over possible errors while reading resource files are by no means exclusive to the *.rsc* format of resource files. An application could devise its own format of data file, and its own library of routines to extract data from these files, but these routines would have to cope with all the same error possibilities as for the rscfile routines. That is, the error possibilities stem not from the rscfile class, but from the notion of resource files itself.

Dealing with errors in rs_read or rs_read_buf

(This section should be skipped on a first reading.)

There follows a more detailed example of how to deal with possible errors during an rs_read call. The case for rs_read_buf is similar, albeit simpler (since there is no possibility of an out of memory error in this case).

```
LOCAL_C VOID ReadResource(VOID *ppcell,INT index)
    {
    INT ret;

    FOREVER
        {
        ret=p_entersend4(rcb,O_RS_READ,index,ppcell);
        if (ret>=0)
            return;
        while (ret<0)
            {
            if (ret==E_GEN_NOMEMORY)
                {
                *ppcell=NULL;/* signal failure to caller */
                TellNoMemory();
                return;
                }
            wsAlertW(WS_ALERT_CLIENT,0,ReplaceDisk,0);
            p_send2(rcb,O_DESTROY);
            FOREVER
                {
                rcb=p_newlibh(OlibCat,C_RSCFILE);
                if (rcb)
                    break;
                TellNoMemory();
                }
            ret=p_entersend3(rcb,O_RS_INIT,rscname);
            }
        }
    }
```

One way to implement the routine `TellNoMemory` - which must never itself run out of memory - would be as follows:

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

    p_errs(&buf[0],E_GEN_NOMEMORY);
    wsAlertW(WS_ALERT_CLIENT,0,&buf[0],0);
    }
```

The way `ReadResource` works, in cases when the user has removed the SSD and has refused to replace it, is to present another alert, wait for the user to respond (by pressing ESC), and then try to re-make the connection with the resource file. For this purpose, various statics are accessed:

`OlibCat`	The value of the category handle of *olib.dyl*, as returned by the earlier call to `p_findlib`
`rscname`	The full path name of where the resource file should be
`ReplaceDisk`	Text that might read, in English, "Replace the application disk".

Clearly, for multi-lingual applications, the string `ReplaceDisk` must itself be read from a resource file. Since the channel to the application resource file is broken at this stage, it may be necessary to read in this text during program initialisation.

Standard practice for applications on a Series3 is to bracket calls to `wsAlertW` with increments and decrements to the reserved static `DatLocked`:

```
DatLocked++;
wsAlertW(...);
DatLocked--;
```

Incidentally, the error handling mechanism described above is implemented *automatically* for object oriented programmers who use the `appman` class and its methods `am_load_resource` and `am_res_buf` to read data from resource files (except that the error recovery code in `appman` is even better, in that it caters with the case of the SSD being removed from one drive and replaced in another).

Advice on where to locate resource files

Mono-lingual applications

In the case of a mono-lingual application, the safest place to locate a resource file is *within the image file.* The `rscfile` class will find any resource file in the second of the four possible *add-file* slots in an *image* file.

For example, if the application is called *archive.app* and the resource file is called *archive.rsc*, an add-file list *archive.afl* should be created, with the following contents:

```
archive.pic
archive.rsc
...
```

where *archive.pic* will be placed in add-file slot 1, and *archive.rsc* in add-file slot 2.

The named files will be added to the resultant image file, whenever this is made, just by virtue of the existence of an *.afl* file with the same basic name as the image file.

In this case, the appropriate name to pass to `rs_init` is simply `DatCommandPtr` (recall that a zero-terminated string giving the full path name of the image file is placed at this reserved static, by the operating system, when the process is started):

```
p_entersend3(rcb,O_RS_INIT,DatCommandPtr);
```

Not only does this scheme have the advantage of simplicity, it also prevents accidents if users copy the main image file to an SSD, but neglect to copy the associated resource file.

For a multi-lingual application, the above continues to apply in any case when a different *.img* file is re-made (using the tool *eremake*) for each new language version. (For more details about *eremake*, see the chapter *Building an Application* in the *General Programming Manual.*)

However, for multi-lingual applications in which the resource data for more than one language is shipped together, the resource files must in general be separate from the main *.img* file. (There is no scope for an indefinite number of add-files.) The documentation for the application should emphasise to users that if the *.app* file (or the *.img* file) is copied from one SSD to another, for consolidation purposes, then appropriate *.rsc* (or *.rzc*) files should also be copied.

Multi-lingual applications

One scheme that has much to recommend it is to rename the resource files as follows:

archiv02.rsc	for a French language resource file
archiv03.rsc	for a German language resource file
archiv18.rsc	for a Dutch language resource file

and so on (for an application *archive.app*), where the numbers at the end of the filename are the *language codes* of the target languages, listed in the documentation of `p_getlanguage` in the *Plib Reference* manual.

These files should be located in a sub-directory *underneath* the directory containing the application program file. The name of this subdirectory should be the same as the basic name of the program file. For example, if the full pathname of the program file is *\app\archive.app*, the full pathnames of the resource files should be *\app\archive\archiv??.rsc*. Again, resource files used by a program with full pathname *\img\backup.img* should be located as *\img\backup\backup??.rsc*.

Then code to determine the name of the resource file, suitable to the language of the computer at run time, could be as follows:

```
TEXT *FindRscName(VOID)
    {
    LOCAL_D TEXT RscNameBuf[P_FNAMESIZE];
    TEXT *RscName;
    P_INFO f;

    RscName=(&RscNameBuf[0]);
    p_atos(RscName,"\\app\\archive\\archiv%02d.rsc",p_getlanguage());
    p_fparse(RscName,DatCommandPtr,RscName,NULL);
    if (p_finfo(RscName,&f)<0)
        p_scpy(RscName,DatCommandPtr);
    return(RscName);
    }
```

Note the check on the existence of the first filename generated by this routine; in case the language as returned by `p_getlanguage` is not supported by the application, the routine defaults back to whatever resource file *is* built into the application.

Copying of applications

The above recommendation for where resource files should be located conforms to the important general rule that if users wish to copy an application *\path\name.ext* from one SSD to another (say from drive *a:* to drive *b:*), all they need to do is type

```
copy a:\path\name.ext b:\path\name.ext
copy a:\path\name\*.* b:\path\name\*.*
```

in which case (assuming the application is not copy-protected!) *all* the files required or presupposed by the application will be transferred.

General comments on multi-lingual applications

It is a common programming error to design an application too closely around the text of one language (eg English), and to discover only at some late stage that various assumptions made fail when the application is translated into another language.

For example, if the English text `"Weekly"` is to be read into some resource, it may be tempting to write some code as follows:

```
TEXT buf[8];

p_entersend4(rcb,O_RS_READ_BUF,WEEKLY_INDEX,&buf[0]);
...
```

However, when the application is translated into German (say), with the entry for `"Weekly"` in the resource file being changed into `"Wöchentlich"`, the new application will most likely crash when the above code is run. The reason is that the buffer of eight bytes, which was long enough to contain the text `"Weekly"`, is not long enough to contain `"Wöchentlich"`.

Better therefore to decide in advance what a reasonable limit on the translation of this term should be, and use *that* as the size of the buffer in the code (not forgetting to inform the translators what the limit is).

The basic principle of independence of code from resource file

One basic guideline is that the code itself should not have to be altered, just because a new translation has been undertaken. The original code should be general enough to start with.

The main problem with allowing code to change at a later date, to simplify the task of translators, is that it is often difficult to foresee the side-effects of such a change. In practice, the most intense testing an application receives is just prior to its launch in the original language; if changes are made at a later date, these may introduce bugs which slip through subsequent testing, on account of that testing being less severe.

For this approach to work, a special test plan has to be devised, focussing on the purely language dependent parts of the application. This test plan should include means of loading, one by one, *all* the resources from the resource file, and displaying them on the screen for validation.

Provided changes made in response to problems thrown up by this test plan are restricted to the resource files themselves, there can be some confidence that the original intensive testing still holds good. But if *code* has to be changed, there is the risk of *regression* - something that used to work now no longer works.

Careful design of screen layout

Design of screen layout is another area where things can go unexpectedly wrong when the contents of a resource file is changed.

In some cases, screen layouts which (just) work in one language, become untenable in another language, because there simply is no acceptable way of translating the text on the screen and still fitting within the allowed display area. For this reason, displays which are already cramped in the original language should be avoided: if they are cramped in one language, they will likely become "grid locked" in some other language, with slightly longer words.

Even if there is ample room in some screen display, care should be taken to calculate various dimensions *dynamically*, ie at run-time, using the widths of the actual characters used, rather than *statically* (ie at compile-time).

Codesize problems

For related reasons, an application which, together with its resource file, only just fits on an SSD of a certain size (say 128k), will be unlikely to fit on a similarly-sized SSD when the resource file has been translated into another language.

Of course, as with the other problems above, it is always possible to *insist* that a sufficiently brief translation be found, but this can result in abbreviations the user is likely to consider ridiculous. It is far better to include some "spare" in the original budget, to allow for some measure of growth as the translation takes place.

Varying keyboards

As well as the text of messages varying from one language to another, it is also possible for the keyboard layout to alter. This does not just mean changing from QWERTY to AZERTY, but changes in which characters can be typed *in combination with various modifiers*.

For example, an application in one translation may define the hot-key PSION+/ as the accelerator for some menu command. However, a foreign language keyboard may move the / key into a place where it cannot be pressed in conjunction with the PSION modifier. Thus on the Series3 keyboard, PSION together with some keys *changes* the characters delivered, into altogether different ones. Therefore, the accelerator would have to alter to some other keypress.

Something else that may have to change, on account of the keyboard changing, is *references to the keyboard* within eg Help text (or inside "Action buttons"). For example, the DELETE key may become the EFF key in a different language variant.

Conclusion

The possible problems of multi-lingual code form another item in the list of things that need to be constantly under background consideration as an application is written.

The discipline of separating text into a resource file is a vital step in the right direction, but by itself, it does not guarantee that all related problems will be solved. Every time text is used in an application, the writer must ask the question: not what *is* the length of this text, but what *might* the length of this text be in some foreign translation - and what would the consequences of that be?

Creating .rsc files using rcomp.exe

The *resource compiler* is a tool *rcomp.exe* that operates on a so-called resource script, which is a text file, to produce a resource file as output.

The process is akin to ordinary compilation:

```
*.c + compiler -> *.obj

*.rss + resource compiler -> *.rsc
```

with *.rss* being the usual extension for a resource script.

As an example, suppose a file *eg.rss* has the following contents:

```
STRUCT STRING
    {
    TEXT str; /* zero terminated text string */
    }

RESOURCE STRING res_start_calc {str="Starting calculation";}
RESOURCE STRING res_scanning {str="Scanning";}
RESOURCE STRING res_items_found {str="%d items found";}
```

Then invoking the command line

```
rcomp eg
```

produces as output a file *eg.rsc* whose contents are exactly as described in the earlier section on the format of *.rsc* files.

Note: some earlier versions of *rcomp.exe* do not accept the syntax

```
RESOURCE <struct-name> <identifier> <definition>
```

instead requiring the addition of the keyword `GLOBAL`:

```
GLOBAL RESOURCE <struct-name> <identifier> <definition>
```

Generated .rsg files

As well as producing a *.rsc* resource file, running *rcomp.exe* also has the effect of creating a *generated header file*, with extension *.rsg*.

Thus the output of typing `rcomp eg` is not only the file *eg.rsc* but also the file *eg.rsg*, having the following contents:

```
#define RES_START_CALC 1
#define RES_SCANNING 2
#define RES_ITEMS_FOUND 3
```

In turn, C source files that need to specify resource indices ought to `#include` these generated *.rsg* files, so that they can include code such as

```
InfoMsg(RES_SCANNING);
```

The present value of `RES_SCANNING` is 2. Suppose however that a new resource is added at the beginning of *eg.rss*. This means that the resource "Scanning" is of course no longer the second in the resultant *.rsc* file, but the third. Accordingly, any calls such as

```
InfoMsg(2);
```

have to change into calls such as

```
InfoMsg(3);
```

in order that they have the same effect as before. Note that having these lines of code instead as

```
InfoMsg(RES_SCANNING);
```

and *recompiling the C source files after changing the resource script* automatically ensures that the desired outcome transpires.

The syntax of the rcomp command

The syntax for invoking *rcomp.exe* includes:

```
rcomp [-s]<name> [-o<oname> -h<hname>]
```

where

`<name>`	is the name of the resource script
`<oname>`	is the name of the resource file output
`<hname>`	is the name of the generated header file.

One possible use of the fuller syntax is to redirect the generated header file to an *..\include* directory.

Include files within a resource script

A resource file can contain lines such as

```
#include "archive.dh"
```

or

```
#include <archive.rh>
```

(no particular significance should be attached to the extensions used in these examples). As would be expected, files specified using the quote form of `#include` are expected to be found in the local directory. However, files specified using the angle bracket form of `#include` are expected to be found in the directory (if any) specified by the value of the DOS environment variable `INCLUDE`.

If required, a batch file such as follows could be used to invoke *rcomp.exe*:

```
set OINCLUDE=%INCLUDE%
set INCLUDE=..\include
\sibosdk\sys\rcomp %1
set INCLUDE=%OINCLUDE%
set OINCLUDE=
```

preserving any previous value of `INCLUDE` for other purposes that may apply on a PC.

Conditional compilation in resource files

Note that the resource compiler supports conditional compilation such as

```
#ifdef BUILD_ONE
...
#endif
```

and

```
#ifndef BUILD_ONE
...
#endif
```

Names (such as `BUILD_ONE`) may be defined when invoking the resource compiler using a `-d` flag as follows:

```
rcomp resfile -dBUILD_ONE
```

This has the same effect as including the corresponding `#define` in the resource file, for example:

#define `BUILD_ONE`

Contents of .rss files

Resource scripts (together with other files they `#include`) are made up of three types of statement:

- comments (identified by C-style `/*` and `*/` delimiters)
- declarations of `STRUCT`s and constants
- declarations of `RESOURCE`s, which are instances of the `STRUCT`s defined.

All the declarations of `STRUCT`s have to precede the first definition of a `RESOURCE`.

White space is ignored (after the first white space character), except within quoted strings. Thus the definitions

```
RESOURCE STRING res_start_calc {str="Starting calculation";}
```

and

```
RESOURCE STRING res_start_calc
    {
    str="Starting calculation";
    }
```

are equivalent. Any indentation of source lines in resource scripts is purely for convenience.

Declaring STRUCTs

STRUCTs are formed of a *name* and a series of *member definitions*. STRUCT names must always be given in upper case, whereas member names are given in lower case. For example,

```
STRUCT STRING
    {
    TEXT str;
    }
```

defines a STRUCT with name STRING and just one member, which has *type* TEXT and *member name* str.

Again, the definition

```
STRUCT MENU_BAR_ITEM
    {
    LINK menu_id;
    TEXT mb_item;
    }
```

defines a STRUCT with name MENU_BAR_ITEM and with two members, the first with type LINK and the second with type TEXT.

As is discussed below, member definitions can also include *default initialisations*.

Possible member types in STRUCTs

The set of allowed STRUCT member types is as follows:

BYTE	stores a numerical value in one byte
WORD	stores a numerical value in two bytes
LONG	stores a numerical value in four bytes
DOUBLE	stores a floating point numerical value in eight bytes
TEXT	stores a zero-terminated sequence of bytes, including the terminating zero
LINK	stores a two-byte *reference to another* RESOURCE
STRUCT	stores a sub-STRUCT in-line.

Of these types, only TEXT and STRUCT have variable length (see below for more on variable length items).

WORDs are stored low byte first then high byte. Similarly, LONGs are stored low word first, then high word.

The resource compiler accepts any value from -128 to +255 for a BYTE, so that C programs are free to interpret the contents as either signed or unsigned. WORD and LONG can similarly be interpreted either as signed or unsigned.

The resource compiler can undertake some limited arithmetical evaluation of data supplied in numeric fields (eg val=4*3.18).

TEXT data can be entered as a combination of quoted strings, binary values, and symbolic constants:

```
str="This is a string.";
```

or

```
str=<84><104>"is a str"<0x69>"ng"<46>;
```

or even

```
#define DOUBLE_QUOTE <34>
...
str="Missing "DOUBLE_QUOTE;
```

Standard C-style processing of backslashes applies within quoted strings. Thus to enter a single backslash in a string, the backslash character has to be repeated in the input, and so on. Thus the final example above could also be given as

```
str="Missing \"";
```

Allowed values of LINK items are constants, symbolic constants, or (lower-case) identifiers of other resources. In referring to another resource, both forward and backward references are possible.

Declaring RESOURCEs

A RESOURCE is declared by specifying:

- the name of the STRUCT being instanced
- the *identifier* of the RESOURCE
- *initialisers*: values of all the members of the STRUCT.

The name of the STRUCT must always be given in upper case, whereas the identifier of the RESOURCE must always be given in lower case.

For example:

```
RESOURCE MENU_BAR_ITEM file_mbar_item
    {
    menu_id=file_menu;
    mb_item="File";
    }
```

During resource compilation, the various identifiers encountered are assigned the values 1, 2, 3, ...

There is no requirement to list the members of the STRUCT in the same order as their definition. Nor is it always necessary to give values for every member:

- any member omitted will be given the *default value* supplied for that member, in the declaration of the STRUCT, if any
- failing this, a "default default" value of zero will be supplied
- however, it is an error to omit altogether to give a value for a LINK member.

For example, it is possible to declare an instance of

```
STRUCT NCEDIT
    {
    WORD current;
    WORD low;
    WORD high=65535;
    }
```

just by the line

```
RESOURCE NCEDIT nc_edit { }
```

in a resource script, in which case a 6-byte long resource will be created, with the three consecutive words containing the values 0, 0, and 65535.

Again, the result of resource compiling the following:

```
STRUCT TEST
    {
    TEXT str;
    BYTE byt;
    STRUCT more;
    }

RESOURCE STRUCT TEST empty { }
```

is a 2-byte long resource, with each byte being set to zero. (Note that "zero" sub-STRUCTs are omitted in their entirety, ie taking up zero length in the resource file.)

Declaring the values of sub-STRUCTs

Whereas the way to define the value of most members of RESOURCEs is by a statement in the form

```
<member-name> = <constant>;
```

the way to define the value of a sub-STRUCT member is

```
<member-name> = <struct-name> {<initialisations>};
```

with the form of <initialisations>, if present, matching that of the definition of a RESOURCE itself. For example:

```
STRUCT NCEDIT
    {
    WORD current;
    WORD low;
    WORD high=65535;
    }

STRUCT TEST
    {
    TEXT str;
    BYTE byt;
    STRUCT more;
    }

RESOURCE TEST values
    {
    str="This is a string";
    byt=42;
    more=NCEDIT {current=100;};
    }
```

Sometimes when there is a sub-STRUCT member in a STRUCT, there will only be one intended STRUCT to fill this slot, but in other cases, there may be more than one possible type of sub-STRUCT.

Leading byte and word length values

In cases when STRUCTs contain sub-STRUCTs, it is frequently helpful to have the instance of the sub-STRUCT preceded by a byte or word giving the length of the instance. This is achieved by amending the definition of the sub-STRUCT: the keyword BYTE or WORD should be included before the opening curly bracket prior to the definitions of the members of the STRUCT.

For example, resource compiling

```
STRUCT FIRST BYTE
    {
    BYTE one;
    }

STRUCT SECOND WORD
    {
    BYTE two;
    }

STRUCT THIRD
    {
    BYTE three;
    }

STRUCT FOURTH BYTE
    {
    STRUCT a;
    STRUCT b;
    STRUCT c;
    }

RESOURCE FOURTH test
    {
    a=FIRST {one=1;};
    b=SECOND {two=2;};
    c=THIRD {three=3;};
    }
```

results in the following 6-byte long resource:

```
<01><01><01><00><02><03>
```

in which the first byte gives the length of the following sub-resource, the third and fourth bytes together constitute a word giving the length of the second sub-resource, and the final sub-resource has no preceding byte- or word- length value.

Note that the resource as a whole lacks a leading byte- or word- length value, despite the presence of the BYTE qualifier in the definition of FOURTH. These leading length values are emitted only when the instance of the STRUCT is as a *sub*-resource.

Incidentally, it is common for definitions such as

```
STRUCT FIRST BYTE
    {
    BYTE one;
    }
```

to be given instead in the equivalent form

```
STRUCT FIRST
BYTE {
    BYTE one;
    }
```

Arrays within resource files

The resource compiler is at perhaps its most powerful in dealing with *arrays* of resources - strictly speaking, arrays of *sub*-resources.

In order to declare an array of sub-resources, a member definition in a STRUCT definition such as

```
<type> <member-name>;
```

has to be changed into one of the forms

```
<type> <member-name>[<array-size>];

<type> <member-name>[ ];

LEN <type> <member-name>[ ];
```

or

```
LEN BYTE <type> <member-name> [ ];
```

For example,

```
STRUCT HELP_ARRAY
    {
    LINK topic_id=0;
    TEXT topic;
    LEN BYTE STRUCT strlst[];
    }
```

in which the initial LINK and TEXT sub-resources are followed by a variable number of sub-STRUCTs (the number varying between different instances of HELP_ARRAY).

In all cases with arrays, the corresponding initialiser statement in a RESOURCE definition

```
<member-name> = <constant>;
```

changes into the form

```
<member-name> = { <constant>, <constant>, ..., <constant> };
```

with <constant> being replaced by <struct-name> {<initialisations>} in the case of an array of sub-STRUCTs.

For a variable sized array, the array of sub-resources may be preceded in the resource file by a byte or word giving the number of elements actually in the array. This count is recorded in a word if the prefix LEN is used, and in a byte if the prefix LEN BYTE is used.

Evidently, the number of sub-resources that actually occur in any given instance of the STRUCT is determined by the syntax of the intialiser, with all but the last element being preceded by a comma.

For example, resource compiling

```
STRUCT STRING
    {
    TEXT str;
    }

STRUCT HELP_ARRAY
    {
    LINK topic_id=0;
    TEXT topic;
    LEN BYTE STRUCT strlst[];
    }

RESOURCE HELP_ARRAY sys_help_print
    {
    topic="How to print";
    strlst=
        {
        STRING {str="Set Printer Model with 'Print setup'";},
        STRING {str="in Word/Agenda/Data, then use 'Print'";}
        };
    }
```

produces a single resource in which:

- the first word is zero (the supplied default value for the *topic_id* member)
- then there follows the sub-resource `"How to print"`
- next comes a byte containing the value 2, being the count of the items in the following array
- finally the strings `"Set Printer ..."` and `"...Print'"` are juxtaposed.

Creating SYSTEM resource files

For completeness, it should be mentioned that the resource compiler enters a special mode if the first line of a resource script is found to consist of precisely the single word

```
SYSTEM
```

In this mode, all LINK references are automatically resolved *with the negative of the correct value.*

Thus whereas the resource script

```
STRUCT STRING {TEXT str;}
STRUCT TEST {LINK lnk;}

RESOURCE STRING alpha {str="Xyz";}
RESOURCE TEST beta {lnk=alpha;}
```

results in the second resource consisting of the word 1, inserting a line

```
SYSTEM
```

at the beginning of the resource script changes the value of this resource to -1.

The rationale of this behaviour is connected with the facility offered to object-oriented programmers by the appman class in *olib.dyl*, to load in resources from the so-called *system resource file* instead of from an application-specific resource file, if the resource identifiers passed are negative. See the documentation on appman for more details.

CHAPTER 3

WDR PRINTING

Introduction

SIBO computers which contain *form.dyl* in their ROM (or on an SSD) support a wide range of services connected with so-called *WDR printing*. These services include the interpretation of printer driver files in the Psion-proprietary *.wdr* format, and the generation of suitable printer command sequences to effect printing operations requested by applications. One key idea is that applications do not need to know which particular printer driver has been selected by the user; system software takes care of converting requests made by applications into command sequences suited to the current printer.

Another service which code in *form.dyl* can provide is opening the correct *printer device* - whether that be the parallel port, the serial port, or a file. Once again the idea is that users can specify the printer device, and have their choice picked up by system software, without any conscious intervention to this effect by individual applications.

All versions of the Series3 have *form.dyl* in their ROM, as do suitably reprogrammed versions of the HC.

Whilst the full power of the WDR printing system can be utilised only by object-oriented programmers, applications that are not themselves object-oriented may still be able to make considerable use of these services:

- picking up the choices made by the user as regards the printer device (these choices are stored in environment variables)
- reading the contents of *.wdr* files directly (by themselves)
- reading the contents of *.wdr* files using services of the `wdr` class in *form.dyl*.

In addition, the Hwif library contains routines `hPrintSetupDialog`, `hPrinterSetupDialog`, `hPrint`, `hPrintSetSI`, `hPrintSensePageWidth`, and `hPrintSenseBufWidth`, which layer over WDR printing services to achieve impressive results adequate for many purposes - without requiring any explicit use of object-oriented techniques (nor any explicit reference to the contents of *.wdr* files). See the *Hwif Manual* for more details.

Creating .wdr files

The creation of *.wdr* files is a quite separate process from their use, once created. Psion can supply a set of *.wdr* files covering some common printers, and other *.wdr* files may be available from third parties. However, it may prove desirable to produce a *.wdr* file for another printer not presently supported - or to enhance the *.wdr* file for a new version of a given printer.

Note that whatever the origin of the *.wdr* file, the file name must not begin with fax. Such filenames are reserved for fax driver files and are automatically interpreted as such.

One way to create *.wdr* files is by using the Sibo printer driver translator, *wdtran.exe*, which creates *.wdr* files from plain text input known as *printer scripts* typically having extension *.wd*.

The operation of *wdtran.exe* is described later in this chapter.

The WDR printing environment variables

WDR printing software may at various times attempt to read or write the four environment variables PD, PS, P$F, and P$M:

P$D is one byte long, having the value '0' to denote that the user has chosen to print using the parallel port, '1' to denote the choice of the serial port, and '2' to denote printing to file

P$S is twelve bytes long, being a copy of the P_SRCHAR struct matching the choices last made by the user in any serial port and handshaking dialog(s)

P$F can be up to P_FNAMESIZE (128) bytes long, being a copy of the filename last chosen by the user as the recipient of any data printed to file

P$M can be up to P_FNAMESIZE+1 (129) bytes long, the first byte recording the *printer model number* (see below), and the remainder giving the full path name of the last *.wdr* file chosen by the user.

Any software that attempts to read these environment variables should bear in mind that these variables do not always exist. Ordinarily, they are created only when the user makes an explicit choice via a dialog box. In the absence of one of the environment variables, the following defaults are assumed to apply:

P$D effectively has the value '0', meaning that printing should be via the parallel port

P$S see below

P$F any printing to file is to the file *p.lis*

P$M the printer model number is 0 and the printer driver file *rom::bj.wdr* should be used.

The default values of the P_SRCHAR struct are as follows:

```
P_SRCHAR ser;

ser.tbaud=P_BAUD_9600;
ser.rbaud=P_BAUD_9600;
ser.frame=P_DATA_8;
ser.parity=0;
ser.hand=P_OBEY_XOFF|P_OBEY_DSR|P_IGN_CTS;
ser.xoff=0x13;
ser.xon=0x11;
ser.flags=0;
ser.tmask=0;
```

For more details on the P_SRCHAR struct, see the *Serial Port* chapter in the *I/O Devices Reference* manual. The essential point is that the contents of the P_SRCHAR struct should be used to P_FSET the serial port after opening it.

For convenience, the printer model number is stored in printable form, with the value of '0' being added to its numerical value. Thus to use the second model in a *.wdr* file - so that the printer model number would be 1 - the first byte in P$M should be 1+'0', ie '1'.

A note on reading environment variables

See the discussion on p_getenviron and p_getenv in the *Plib Reference* manual.

Note that it is also possible to read and modify environment variables (eg for experimental purposes) by using the SIBO Debugger.

Overview of the contents of a .wdr file

This section provides an overview of the contents of a *.wdr* file. More details are provided in later sections.

Just about the simplest possible *.wdr* file would have the following contents, when dumped:

```
 0:  89 00 0c 00 57 44 52 30  35 00 00 00 01 00 05 00   ....WDR0 5.......
10:  47 65 6e 65 72 61 6c 00  00 00 00 00 00 00 00 00   General. ........
20:  00 00 00 00 00 00 00 00  17 00 00 00 00 00 00 00   ........ ........
30:  00 00 00 00 00 00 00 01  0c 01 0d 02 2a 0a 00 02   ........ ....*...
40:  2a 20 00 00 00 00 00 00  01 00 02 05 23 4d 6f 6e   * ...... ....#Mon
50:  6f 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   o....... ........
60:  00 00 00 00 00 03 00 01  00 f0 00 00 00 00 00 00   ........ ........
70:  00 00 01 00 01 00 01 00  01 16 00 90 00 f0 00 00   ........ ........
80:  00 00 00 00 00 01 00 04  00 04 00 28 00 48 00 4d   ........ ...(.H.M
90:  00 7b 00 89 00                                     .{...
```

This conforms to the pattern

```
<header><resources><index table>
```

of Sibo standard *.rsc* resource files, as discussed in the *Resource Files* chapter in this manual.

Indeed, *.wdr* files are but examples of *.rsc* files, with the extension changed to denote the particular purpose of being a wdr printer driver file.

In fact, *.wdr* files come in two types - *compressed* and *uncompressed* (*standard*), corresponding to the *.rzc* and *.rsc* forms of resource file. Uncompressed files can be read by a variety of methods, as discussed in the *Resource Files* chapter. However, in order to read compressed *.wdr* files, it is more or less necessary to use some of the functionality of object-oriented classes in the ROM, using either:

- the `rscfile` class in *olib.dyl*
- the `wdr` class in *form.dyl*

with the preference being for the latter, since it contains more explicit knowledge of the particular contents of *.wdr* files.

Various services provided by the `wdr` class are described later in this chapter.

Deciphering general.wdr

The above dump is in fact that produced from an (uncompressed) version of the file *general.wdr* that is part of the Series3 ROM. This file describes the most basic kind of printer possible, possessing only one font, which is monospaced, and assumed to be 12 point (ie six lines per inch - one point corresponding to 1/72 of an inch) and 10 cpi (characters per inch). For this printer, the only way to position the print head horizontally is by emitting space characters or carriage returns, and the only way to position the print head vertically is by emitting line feeds or form feeds.

The index table for the file starts at file offset `0x89` (as contained in the first word in the file). Reading successive words from this index indicates that the individual resources in the file are to be found at file offsets `0x04`, `0x28`, `0x48`, `0x4d`, and `0x7b`.

The header resource

The first resource in any *.wdr* file is always the *header resource*, having the following structure:

- the first six bytes give a signature, which must always be `"WDR05"` for a valid *.wdr* file
- the next word gives the so-called *wdr-flags* for the file - evidently `0` in this case
- the word after that gives the number of different *printer models* in the file - in this case, there is only one in the file
- finally there is an array of `WDR_MODEL_INDEX` structs, one struct for each printer model in the file
- each `WDR_MODEL_INDEX` starts with a word giving the resource identifier for the *model resource* (see later) giving more information about the printer model - this identifier has value `5` in this case
- the `WDR_MODEL_INDEX` struct concludes with up to 24 bytes giving the *public name* of the model, in a zero-terminated string - `"General"` in this case.

For *general.wdr*, the total size of the first resource is evidently `6+2+2+1*(2+24)`, ie `0x24`. For other *.wdr* files which contain more than one printer model, the first resource will be larger.

The commands resource

The second resource in any *.wdr* is always the *commands resource*. This consists of:

- an array of *command strings*
- preceded by a count byte (the first byte in the resource)
- followed by a trailing word whose meaning is reserved for future expansion.

In *general.wdr* above, the value of the count byte is 0x17, ie 23.

Each command string that follows is given with a leading byte count, and without any terminating zero (as is appropriate, given that the command strings can in general contain *embedded* zeros).

As can be seen, for *general.wdr*, all the 23 command strings are zero, except for (starting counting at zero):

string 14	which has the value 0x0c
string 15	which has the value 0x0d
string 16	which has the value '*' 0x0a
string 18	which has the value '*' 0x20

The first 21 of the command strings in the command block have reserved meanings, best described in simple terms by giving the descriptive text associated with these commands in *.wd* files (see *Contents of .wd files* section later):

```
0   "RESET"
1   "FORM_LENGTH"
2   "PREAMBLE"
3   "POSTAMBLE"
4   "UNDERLINE_ON"
5   "UNDERLINE_OFF"
6   "BOLD_ON"
7   "BOLD_OFF"
8   "ITALIC_ON"
9   "ITALIC_OFF"
10  "SUPERSCRIPT_ON"
11  "SUPERSCRIPT_OFF"
12  "SUBSCRIPT_ON"
13  "SUBSCRIPT_OFF"
14  "NEW_PAGE"
15  "CARRIAGE_RETURN"
16  "MOVE_DOWN"
17  "MOVE_RIGHT_PREFIX"
18  "MOVE_RIGHT"
19  "MOVE_RIGHT_SUFFIX"
20  "LANDSCAPE"
```

Additional command strings can be used to set various fonts, and are referenced by *typeface resources*, discussed below.

Model resources

The resource identifiers of all model resources are listed in the header resource. As mentioned above, the only model resource in *general.wdr* has identifier 5. Consulting the index table for the file indicates that this resource starts at file offset 0x7b (recall that resource identifiers start at 1).

The structure of a model resource is as follows:

- the first five words give the values of the so-called *minx*, *miny*, *skipx*, *skipy*, and *model-flags* for the printer model
- next comes a word giving the number of following *typeface resource* identifiers
- this is followed by an array of the specified number of identifiers of typeface resources.

For *general.wdr*, the values of the various values are evidently as follows:

minx	0x90 (144)

miny	0xf0 (240)
skipx and *skipy*	both zero
model-flags	zero

and there is just one reference to a typeface resource, this having resource identifier 4 (whence it can be found at file offset 0x4d).

Typeface resources

The structure of a typeface resource is as follows:

- the first 20 bytes give the *public name* of the typeface, as a zero-terminated string
- the next word gives the *typeface number* of the typeface
- the word following contains *typeface-flags*
- next comes a word which, if non-zero, contains the identifier of a *translates resource* to be used by the typeface
- the word following that gives a count of the number of different *sizes* (or *fonts*) that the typeface comes in
- finally there is an array of WDR_FONT structs, one struct for each font size supported by the typeface
- the WDR_FONT struct consists of nine words: *height*, *height_max*, *height_delta*, *width_scale*, *width_normal*, *width_italic*, *width_bold*, *width_italic*, *width_bold_italic*, and concluding with the number of the command string, in the commands resource, of the associated printer instruction to set this particular font.

The difference between a "typeface" and a "font" is discussed in more detail below.

In *general.wdr*, the public name of the only typeface in the file is "Mono", the typeface number and the typeface flags are both zero, the translates resource with identifier 3 is to be used, and there is only 1 WDR_FONT struct following in-line.

In a WDR_FONT struct:

- The fields *height_max* and *height_delta* are only relevant for so-called *scalable* typefaces
- The field *width_scale* is only relevant for proportional typefaces
- For monospaced typefaces, the values of the four fields *width_normal* through *width_bold_italic* are to be interpreted as real numbers; for proportional typefaces, in which the widths of the characters vary from character to character, these values are identifiers of *font width table* resources.

There are no font width table resources in *general.wdr*. Incidentally, font width tables are stored in *difference* form in *.wdr* files - see later for more details.

Translates resources

The first word in a translate resource gives the number of translates that follow in-line.

Each translate starts off with a byte giving the length of the remainder of the translate. The next byte is the Ascii value of the character to be translated, and that is followed by a sequence of bytes into which the character is to be translated.

In *general.wdr*, there is only one translates resource, which in turn only contains one translate, whose effect is to convert every character with Ascii value 0x05 into one with value 0x23. This results in telephone symbols (recorded internally on the Series3 as 0x05's) being printed as hash signs.

Summary of resource types in a .wdr file

The above survey contains one example of every possible type of resource in a *.wdr* file, except for font width table resources.

In summary, the possible resource types are:

header	(always the first resource in the *.wdr* file) containing an index of all the printer models supported by the file, as well as some important wdr-flags

commands	(always the second resource in the *.wdr* file) containing the character command sequences for resetting the printer, controlling the text format, moving the print head position, selecting specified fonts, and so on
translates	used to map the printer's character set onto that used by the SIBO computer (which is based on IBM code page 850)
font width tables	defining the widths of all the characters in proportional typefaces
models	giving essential data governing the capabilities of a printer, and listing the typefaces supported by the printer
typefaces	describing the typefaces supported by a printer model, including the various "font sizes" available for that typeface.

More details on the contents of .wdr files

Possible wdr-flags

The only wdr-flag of any general significance is `WDR_DYL_LOAD`, with value `0x01`. If set (in the header resource), this means that the contents of the *.wdr* file are insufficient, by themselves, to describe the behaviour of the printer fully, and that the wdr printing system software should load a suitable external dyl to additionally customise the print behaviour.

Examples of *.wdr* files with the `WDR_DYL_LOAD` flag set are the printer driver files for postscript printers.

Further discussion of loading additional printer dyls is beyond the scope of this document.

Other bits set in the wdr-flags may have special significance for code in the extra print dyls.

The notion of "printer models"

The notion of a *printer model* is essentially a device to cover more than one printer using the same data.

Different printer models can be described in the same *.wdr* file, even if they support different sets of typefaces or have other differing characteristics, so long as they have the same basic set of printer command strings (and the same wdr-flags).

The public name of a printer model is what is presented to the user in any dialog offering a list of "printer models" for selection.

Overview of the different command strings

As well as the command strings to select various fonts, a *.wdr* file contains commands to have the printer perform other functions. These commands are by and large clearly named in the listing given earlier, eg `"ITALIC_ON"` and `"ITALIC_OFF"`, `"BOLD_ON"` and `"BOLD_OFF"`, and `"SUPERSCRIPT_ON"` and `"SUPERSCRIPT_OFF"`.

If a printer cannot support a given feature, the corresponding command string would generally be left null (""). One possible exception is italic which, if not supported, could be implemented as an underline. (Note incidentally that it is possible for a printer which supports italic in one font not to support it in another font, and so on. Again, a printer may support both italic and superscript, but not both at the same time.)

The `"LANDSCAPE"` command, if non-null, is the command to cause the printer to enter landscape mode (as opposed to portrait mode).

The string of commands sent to the printer when printing starts are among the most important, as regards influencing the printed outcome. The very first command the software sends the printer is the `"RESET"` command. Then it sends a `"FORM_LENGTH"` command, and then the `"PREAMBLE"` command, before starting to print the document proper (together with headers and footers, etc). At the very end, a `"POSTAMBLE"` command is sent.

The `"POSTAMBLE"` command may be needed to flush the printer buffer, and to restore the printer to its default settings.

The `"PREAMBLE"` command may have such drastic consequences as choosing the basic configuration of the printer (possibly overriding defaults set via dip switches). In any case of doubt, the documentation for a particular printer should be consulted carefully.

Special characters in command strings

The commands "MOVE_RIGHT", "MOVE_DOWN", and "FORM_LENGTH" are each used in conjunction with a value passed by the printer subsystem software. For example, the commands are to set the form length *to a given value*, or to move the printer position right *by a given amount*. This value may either end up in the command string by a process of substitution, or it may result in the command being repeated as required:

- if any of these commands is defined as starting off with an asterisk character ('*'), what is actually sent to the printer is the *remainder* of the command string (ie minus the initial asterisk) *repeated* the specified number of times
- if the string "%d" is contained within the command string, the specified value is converted into decimal representation and is substituted for the "%d" (like `printf` in C)
- likewise the string "%c" means to substitute the specified value as a single byte (character), and "%w" means to substitute it as a pair of bytes (low byte first).

For example, in some printer drivers "MOVE_DOWN" is defined as "*<10>", so that "MOVE_DOWN n" will be sent to the printer as n line feed characters (line feed is Ascii 10).

Again, in the HP Laserjet III printer driver, "MOVE_DOWN" is defined as "<27>&a+%dV", so that "MOVE_DOWN 6" (say) will be sent to the printer as "<27>&a+6V".

This kind of substitution can also take place in the command strings to select a specific size of a so-called *scalable font* - see below.

The MOVE_RIGHT commands

For some printers, the command to move right by a certain amount is of the general form

```
<prefix><repeated body><suffix>
```

with the central part being repeated as many times as required, depending on the amount by which the print position is to be adjusted.

It is to cope with this case (as well as ones even more complicated) that the commands "MOVE_RIGHT_PREFIX" and "MOVE_RIGHT_SUFFIX" are provided. These will be left null for most printers.

Printer units

The units for the "FORM_LENGTH" command are *always* 1/6 of an inch. Thus if the print software wishes, as part of initialising a printer, to set the form length to 12 inches, the command

```
"FORM_LENGTH 72"
```

should always be sent.

However, the units used in many other features of printer driver files varies from printer to printer. The key quantities are the values of *minx* and *miny*, as specified in the model resource for a printer.

Minx and miny are themselves standardly given in so-called *twips*, where twenty twips make a point (so that 1440 twips make an inch). For example, in the file *general.wdr* discussed above, minx has the value 144 twips, ie 1/10 of an inch, and miny has the value 240 twips, ie 1/6 of an inch. This matches the basic assumptions made in *general.wdr* that the font printed is 12 point and 10 cpi (see earlier).

The fundamental significance of minx is that this is the smallest amount by which the print position can be adjusted horizontally. Similarly, miny is the smallest amount by which the print position can be adjusted vertically. Clearly, the smaller minx and miny are, the higher the resolution of the printer.

The above values make sense for *general.wdr* since the only way the print position can be adjusted horizontally is by emitting a space character (or by emitting a carriage return, which resets the horizontal position), and the only way the print position can be adjusted vertically is by emitting a linefeed character (or by emitting a formfeed character, which effectively resets the vertical position).

A command such as "MOVE_RIGHT n" actually means to move the print position right by n times minx, and similarly a command such as "MOVE_DOWN m" means to move the print position down by m times miny.

The value of skipx for a printer model is subtracted from the first "MOVE_RIGHT" command in each line of text, to compensate for the fact that many printers cannot print at the left edge of the paper. The value of skipy is likewise subtracted from the first "MOVE_DOWN" command in each page, to compensate for the fact that many printers cannot print at the very top of the paper.

The values of skipx and skipy are themselves expressed in terms of minx and miny, respectively. Thus if skipy is given as 36 whereas miny is given as 20, this translates to an actual height of some 20*36 twips, ie half an inch, at the top of the paper which is inaccessible to the printer.

Possible model-flags

There are two bits that can be set in the model-flags in a model resource:

- `WDR_MODEL_LANDSCAPE_AVAILABLE` (0x01) has to be set if the model supports being put into landscape mode
- `WDR_MODEL_MINX_IS_DOTS_PER_INCH` (0x04) should be set if the value of minx is expressed, not in twips (as standard), but in reciprocal inches (so that a minx of 300 would correspond to 1/300 of an inch - which is not expressible as an exact number of twips).

Note that even if `WDR_MODEL_MINX_IS_DOTS_PER_INCH` is set, the value of miny is *always* expressed in twips.

Typefaces and fonts

A *typeface* is considered to be a set of characters in a particular style, whereas a *font* is a particular size of a typeface.

The public name of a typeface is what is presented to the user in any dialog offering a list of "fonts" (actually typefaces) for selection. The various different fonts within a chosen typeface will be selected via a *secondary* choice list, keyed by the notional height of the fonts.

There can be considerable scope for authors of *.wdr* files in deciding how to represent the different fonts supported by a printer (especially dot matrix printers). For example, many dot matrix printers support a condensed font: this may be represented as a typeface called `"Pica condensed"` (say) or as a smaller font of the `"Pica"` typeface. It is generally more useful for the user to have a number of size variants (fonts) of one typeface rather than a number of typefaces each having only one size. For this reason the second of the above approaches is the recommended one. It also has the advantage that the typeface names will then (generally) be language independent, whereas the addition of a phrase such as `"double width"` or `"condensed"` immediately makes the *.wdr* file language dependent.

A dot matrix printer may support a number of variants of a font including: condensed, double height, double width, condensed double width etc. For a twelve point base font it is recommended to map these to the following heights:

condensed	7 point (140 twips)
normal	12 point (240 twips)
condensed double width	13 point (260 twips)
double width	16 point (320 twips)
double height	22 point (440 twips)
double height double width	24 point (480 twips)

Note that a taller font *must* have a larger size than a shorter font, in particular the point sizes of all the double height fonts must be larger than all the single height fonts.

If a dot matrix printer supports a large number of variations on a base font, it may turn out that two different variations would be mapped onto the same point size. In this case it would be perfectly acceptable to just omit one of the fonts: if there are already 12,13,14,15 and 16 point fonts available then omitting (say) a second 14 point is not really a hardship to the user. Alternatively, the font could be incorporated in another typeface. In case it is decided to omit a font, bear in mind that double height fonts generally look much better than double width fonts, so given a choice it is better to omit the latter.

Typeface numbers

The main significance of the typeface number of a typeface is when a document set up for one printer is subsequently printed on another printer. Font "substitutions" have to be made - and these are done according to the values set for the typeface number.

Thus if a document is prepared for one printer model, and some text is to be printed in a typeface having typeface number 2, say, and then the user changes the printer model setting for the document to another printer, that text, when printed, will be printed in the first typeface found in the new printer model, having the same typeface number.

In case no exact match in typeface number is possible, the first typeface defined in the new printer model is used instead.

There are a large number of allowed typeface numbers, listed later in this chapter. Note that typeface numbers are completely independent of the public names of typefaces.

Typeface numbers may also be used in some forms of RTF file conversion.

Possible typeface-flags

There are three bits that can be set in the typeface-flags in a typeface resource:

- `WDR_TYPF_PROPROTIONAL` (0x01) set if the widths of the characters in a font in this typeface can vary among themselves (the alternative is that the fonts are *monospaced*)
- `WDR_TYPF_SCALED` (0x02) set if the fonts supported by the typeface are all generated by the printer as being different scaled versions of one common pattern
- `WDR_TYPF_SERIF` (0x04) set if the typeface is serif.

The `WDR_TYPE_SERIF` flag has significance only in certain types of RTF file transfer.

Note that scalable monospaced typefaces are not supported, so that scalable fonts always have to be regarded as proportional, even if the widths of their characters do not vary in fact.

For scalable typefaces, there is only one `WDR_FONT` struct per typeface, with all required information for differently sized fonts being generated from this by arithmetical scaling.

Heights of fonts

The command string of a scalable typeface must include some kind of parameter (eg "%d") for the particular size required to be specified when setting the font. This parameter will be filled in by system software giving the height of the required font *in point units*.

However, heights of fonts in *.wdr* files are always given in twips (thus 240 for a 12 point font).

For non-scalable fonts, the height_max and height_delta fields in a `WDR_FONT` struct are meaningless. For scalable fonts, the set of supported font sizes is obtained by repeatedly incrementing by height-delta, from the value of height to the value of height_max (so that the "height" field actually plays the role of a "height_min" field).

In general, 4 points (80 twips) is a sensible minimum height for a font, whereas the maximum allowed height is determined by the fact that the widths of characters must at all times remain less than 255.

Widths of characters in fonts

Widths of characters in fonts in *.wdr* files are expressed - as are *all* horizontal measurements in *.wdr* files - in units of minx.

The wdr system allows for widths of characters altering as they are italicised or bolded, and again when they are simultaneously italicised *and* bolded. In case there is no such variation, the values of the fields width_normal, width_bold, width_italic, and width_bold_italic, will merely duplicate each other.

For proportional fonts, these four fields each refer to font width tables. These are stored in *difference* form in *.wdr* files. Thus if the array of stored widths is `stored[]`:

- width of character 0 = `stored[0]`
- width of character 1 = width of character 0 + `stored[1]`
- width of character 2 = width of character 1 + `stored[2]`
- width of character 3 = width of character 2 + `stored[3]`

and so on. The rationale for this is that the differences are frequently zero, so that the font width table is stored as an array, many of whose values are zero. In turn, this compresses much more markedly (for compressed versions of *.wdr* files) than tables containing many different values - with the result that smaller *.wdr* files get produced (bear in mind that font width tables potentially make up large parts of these files).

The values obtained from a font width table should all be multiplied by the width_scale value for the font. This mechanism avoids needless duplication of data in which two font width tables would otherwise both be present in a *.wdr* file, even though one is merely a scaled version of the other.

For scalable fonts themselves, the values in the font width table (once multiplied by any width_scale value for the font) are what would apply to a fifty point high version of the font. These have to be further scaled, in general, to match the chosen height of the font.

Note that in all cases, widths of characters cannot exceed 255.

Creating .wdr files using wdtran.exe

The *printer driver translator* is a tool *wdtran.exe* that operates on a so-called printer script, which is a text file, to produce a printer driver file as output.

The process is akin to ordinary compilation:

```
*.c + compiler -> *.obj

*.wd + printer driver translator -> *.wdr
```

with *.wd* being the usual extension for a printer script.

Some sample *.wd* files are distributed as part of the optional component of the Sibosdk.

To produce eg *general.wdr* from *general.wd*, simply type

```
wdtran general
```

Contents of .wd files

The basic contents of *.wd* files correspond to the different resources in *.wdr* files (see earlier in this chapter).

A *.wd* file consists of a number of resource definitions, of which there are five types: `COMMANDS`, `TRANSLATES`, `WIDTHS`, `TYPEFACE`, and `MODEL`.

There must be one and only one `COMMANDS` resource in a *.wd* file. There must be at least one `MODEL` resource and at least one `TYPEFACE` resource, though there can be more of each. There can be any number (including zero) of `WIDTHS` and `TRANSLATES` resources.

Note that there are no definitions in a *.wd* file directly corresponding to the header resource in a *.wdr* file. The header resource is specially created by *wdtran.exe*.

Each resource definition consists of a header, a number of commands, and then a footer. The header is of the form

```
<RESOURCE> [identifier]
```

with the identifier being required only if the resource is to be referenced by another resource (it may be the name of a width table, for example).

The resource footer is of the form:

```
END_<RESOURCE>
```

Each intervening command is of the form:

```
keyword [parameter]
```

For example, the definition of a `TYPEFACE` resource looks like

```
TYPEFACE pica
...
END_TYPEFACE
```

and the definition of a COMMANDS resource looks like (this is the COMMANDS resource for the file *general.wd*):

```
COMMANDS
    RESET               ""
    FORM_LENGTH         ""
    PREAMBLE            ""
    POSTAMBLE           ""
    BOLD_ON             ""
    BOLD_OFF            ""
    ITALIC_ON           ""
    ITALIC_OFF          ""
    UNDERLINE_ON        ""
    UNDERLINE_OFF       ""
    SUBSCRIPT_ON        ""
    SUBSCRIPT_OFF       ""
    SUPERSCRIPT_ON      ""
    SUPERSCRIPT_OFF     ""
    MOVE_DOWN           "*<10>"
    MOVE_RIGHT_PREFIX   ""
    MOVE_RIGHT          "*<32>"
    MOVE_RIGHT_SUFFIX   ""
    NEW_PAGE            "<12>"
    CARRIAGE_RETURN     "<13>"
END_COMMANDS
```

Parameters to keywords inside a resource definition may be of four types:

Numeric	a decimal number which is interpreted as a value (eg a font width)
Constant	an unquoted string which is interpreted as a numeric value (used only by TYPE in a TYPEFACE resource definition)
Identifier	a lower case unquoted string, used by a command to reference another specified resource
Quoted string	a sequence of characters in quotes that is sent to the printer to perform a specific function (eg BOLD_ON); the string may contain control characters (see below).

The printer driver translator ignores any characters between an exclamation mark ! and the end of a line (unless the ! is in a quoted string). Blank lines are also ignored.

The block indentation scheme in standard *.wd* files is for convenience only.

Acceptable syntax within a COMMANDS resource definition

Any of the commands listed in the earlier section on command resources in *.wdr* files can be given. However, command strings used to set individual fonts should be given within the relevant TYPEFACE resource definition.

Control codes may be needed in the COMMANDS resource definition, or in the COMMAND keyword of a font definition. A control code is specified by placing its decimal value in angle brackets ("<" and ">"). For example the ESC control code (decimal value 27) is specified as <27>.

Other syntax that may occur within a COMMANDS resource definition are:

USE_DYL	to set the WDR_DYL_LOAD flag in the wdr-flags in the header resource for the *.wdr* file
FLAGS <flags>	to *or* in <flags> to the wdr-flags in the header resource
HP_PCL_COMPATIBLE	of purely historical significance.

Acceptable commands within a TRANSLATES resource definition

Each line within the definition of a TRANSLATES block should be made up of one or more *translations*, with adjacent translations on any one line being separated from each other by white space.

Individual translations should have the form

```
<byte>:<byte>
```

or else

```
<byte>:<string>
```

For example:

```
5:35

156:"<27>R<3><35><27>R<0>"
```

Note that any width table for a font which uses translates must contain the correct width for each character *after it is translated*. The printer driver translator does not check this.

Acceptable commands within a WIDTHS resource definition

Each line within the definition of a `WIDTHS` block should be made up of one or more *character widths*, with adjacent character width definitions on any one line being separated from each other by white space.

Individual character width definitions should have the form

```
<byte>:<width>
```

Note that in contrast to *.wdr* files, which store font width tables in differenced form (see earlier in this chapter), *.wd* files should define the widths of all characters in absolute terms. The differencing is performed by *wdtran.exe* (just as the converse integration is performed by the `wdr` class in *form.dyl* without the conscious intervention of any application software).

The printer driver translator will give an error if the widths of the non-breaking hyphen, potential hyphen, and standard hyphen (character codes 7, 14, and 45) are not all the same.

Likewise, the widths given for the non-breaking space, the standard space, and the tab (character codes 15, 32, and 9) also all have to agree.

Acceptable commands within a TYPEFACE resource definition

The following commands may be present within the definition of a `TYPEFACE` resource:

`PROPORTIONAL`	to set `WDR_TYPF_PROPORTIONAL` in the typeface-flags
`SCALED`	to set `WDR_TYPF_SCALED` in the typeface-flags
`SERIF`	to set `WDR_TYPF_SERIF` in the typeface-flags
`MULTIPLE_FONT_WIDTH_TABLES`	of purely historical interest
`NAME <name>`	to give the public name of the typeface
`TYPE <type>`	to give the typeface number of the typeface
`TRANSLATE <identifier>`	to specify a `TRANSLATES` resource to be used
`FONT`	to commence the definition of a `FONT` sub-resource.

The `FONT` keyword must appear at least once in each `TYPEFACE` definition, and can appear more than once. The other keywords should only appear once, at the most.

Each `FONT` sub-resource definition follows the same general pattern as the other resources:

```
FONT
<keywords>
...
END_FONT
```

Possible keywords in the body of a `FONT` resource definition are `HEIGHT`, `HEIGHT_MAX`, `HEIGHT_DELTA`, `WIDTH_SCALE`, `WIDTH`, `WIDTH_BOLD`, `WIDTH_ITALIC`, `WIDTH_BOLD_ITALIC`, and `COMMAND` - each having the straightforward meaning of specifying a corresponding field within the `WDR_FONT` structure for that font (see earlier in this chapter).

Acceptable commands within a MODEL resource definition

The following commands may be present within the definition of a `MODEL` resource:

`LANDSCAPE_AVAILABLE`	to set `WDR_MODEL_LANDSCAPE_AVAILABLE` in the model-flags
`MIN_X_IS_DOTS_PER_INCH`	to set `WDR_MODEL_MINX_IS_DOTS_PER_INCH` in the model-flags

NAME <name>	to give a public name for the printer model
TYPEFACE <identifier>	to specify a TYPEFACE resource supported by the printer model.

Additionally, the keywords MIN_X, MIN_Y, SKIP_X, and SKIP_Y straightforwardly specify the minx, miny, skipx, and skipy values for the printer model.

In contrast to the case for TYPEFACE resources, which can only have one public name each, MODEL resources can have more than one public name each. This avoids needless duplication in a *.wd* file, if it turns out that two MODEL blocks would otherwise be identical. Note that one MODEL resource having two public names is *not* in general the same thing as there being two different MODEL resources in the same *.wd* file (though in each case, there will be two distinct entries in the Printer Model choice list).

Allowed typeface numbers

TYPE may take any of the values:

COURIER	OPTIONAL_SB	RUSSIAN
PICA	OPTIONAL_SC	OPTIONAL_B
ELITE	TIMES_ROMAN	OPTIONAL_C
PRESTIGE	CENTURY	OPTIONAL_D
LETTER_GOTHIC	PALATINO	NARRATOR
GOTHIC	SOUVENIR	EMPHASIS
CUBIC	GARAMOND	ZAPF_CHANCERY
LINEPRINTER	CALEDONIA	OPTIONAL_DA
HELVETICA	BODONI	OLD_ENGLISH
AVANT_GARDE	UNIVERSITY	OPTIONAL_DB
SPARTAN	SCRIPT	OPTIONAL_DC
METRO	SCRIPT_PS	COOPER_BLACK
PRESENTATION	OPTIONAL_SCA	SYMBOL
APL	OPTIONAL_SCB	LINE_DRAW
OCR_A	COMMERCIAL_SCRIPT	MATH_7
OCR_B	PARK_AVENUE	MATH_8
STANDARD_ROMAN	CORONET	DINGBATS
EMPEROR	OPTIONAL_SCC	EAN
MADELEINE	GREEK	PC_LINE
ZAPF_HUMANIST	KANA	OPTIONAL_SYA
CLASSIC	HEBREW	
OPTIONAL_SA	OPTIONAL_A	

Suppose, for example, a user has a document written when the Apple Laserwriter printer driver was selected, and the document uses two typefaces, *Times Roman* and *Palatino*. The user then changes to a HP Laserjet III printer driver. *Times Roman* on the Laserwriter and *CGTimes* on the HP III both have a TYPE of TIMES_ROMAN and so the *Times Roman* typeface would be mapped to *CGTimes*. The HP III driver has no typeface with a TYPE of PALATINO and so the *Palatino* font would be mapped to *Courier*.

The following rules are useful for guidance:

One of the printers monospaced typeface should be assigned a TYPE of COURIER: this should be the typeface described as Courier or perhaps Pica in the printer's manual. The typeface's NAME should be as in the manual (eg "Pica").

If a printer supports only one proportional typeface it should be assigned a TYPE of TIMES_ROMAN, although it may be given a different NAME (eg "Proportional").

If a printer supports more than one proportional font then the one with serifs (if available) should be assigned a TYPE of TIMES_ROMAN and the one without serifs should be assigned a TYPE of HELVETICA; their NAMES should, however, be those used in the printer's manual, eg *CGTimes* and *Univers* for the HP Laserjet III.

CHAPTER 4

DBF FILES

Introduction

Database files (*DBF* files) are binary files containing typed, variable length records. Many SIBO applications (for example, the MC Diary and the Series 3 Database) store their data in DBF files. The data files created and manipulated by Opl are also examples of DBF files.

Database files are designed to be *Flash-friendly*, that is, they may be stored and manipulated in Flash SSDs (or any other EPROM medium). A DBF file stored on such a medium may be modified by appending, deleting, or replacing records without having to make a new copy of the entire file.

The chapter *Database Files* in the *Plib Reference* manual describes DBF files from the point of view of reading and writing them using Plib function calls such as `DbfOpen`, `DbfAppend`, `DbfFindRead`, and `DbfNextRead`. The *ISAM Manual* discusses more advanced techniques for using DBF files, in conjunction with independent keyed index files. The present chapter focuses instead upon a description allowing access to DBF files *independently* of these specialised Plib and ISAM functions.

First, the basic structure of a general DBF file is reviewed, and then particular examples are given of how various SIBO applications use DBF files. This information should assist the creation of *file format conversion* programs such as might run on another computer, for example converting between Series 3 Agenda files and files that can be read directly by PC-based PIMs (Personal Information Managers).

Basic structure of DBF files

The following discussion is based around a DBF file created by a simple Opl program. It is not necessary to be familiar with Opl to follow the discussion, since the contents of the DBF file are described independently of the Opl program. It just happens that Opl is a convenient way to create DBF files quickly, so that experimentation is easier.

The Opl program creating the file is as follows:

```
PROC writedbf:
    if exist("test.dbf") :delete "test.dbf" :endif
    create "test.dbf",a,f1$,f2$,f3%,f4
    a.f1$="Hello"
    a.f2$="2"
    a.f3%=3
    a.f4=4
    append
    a.f1$="World"
    append
    close
    beep 5,300
ENDP
```

The `create` statement creates a DBF file with the name `test.dbf`, by default in the *\opd* top-level directory on the default drive. This file is assigned the logical handle `a` inside the program. Each record in the file is to have four fields (called `f1$`, `f2$`, `f3%`, and `f4` inside the program). Standard Opl naming conventions mean that these fields have types *string*, *string*, *integer*, and *double*, respectively.

The two append statements mean that two records are written to the file, before it is closed.

Dumping the resulting file yields the following:

```
 0:  4f 50 4c 44 61 74 61 62  61 73 65 46 69 6c 65 00   OPLDatab aseFile.
10:  0f 11 16 00 0f 11 04 20  03 03 00 02 12 10 05 48   ....... ....... H
20:  65 6c 6c 6f 01 32 03 00  00 00 00 00 00 00 10 40   ello.2.. ....... @
30:  12 10 05 57 6f 72 6c 64  01 32 03 00 00 00 00 00   ...World .2......
40:  00 00 10 40                                       ...@
```

This conforms to the standard DBF format of

```
<standard header><extended header><field information record><other records>
```

where

`<standard header>`	always has length 22 bytes.
`<extended header>`	can have variable length and is frequently of zero length (as here).
`<field information record>`	gives the structure of all *type 1* records following.
`<other records>`	are the main body of the DBF file.

The standard header

The first sixteen bytes of all standard DBF files are the zero-terminated string `"OPLDatabaseFile"`. This string is used by all SIBO applications. Applications are allowed to use alternative strings although such DBF files can not be read by Opl.

The two bytes at file offset `0x10` will in practice always contain the bytes `<0f><11>`. The same applies to the bytes at file offset `0x14`.

The two bytes at file offset `0x12` contain the file offset for the start of the *field information record*. Thus in the absence of an extended header they would contain `<16><00>` and for an extended header of length 256 bytes they would contain `<16><01>`.

The extended header

A DBF file has an extended header only if an application calls the Plib function `DbfExtHeaderWrite`. At the time of writing SIBO applications neither make this call nor make use of the extended header.

The field information record

The field information record is a *type 2* record.

The first byte contains the number of fields defined for each *type 1* record (see below for an explanation of *type 1* records). In the above example four fields were defined. The number of fields defined must be an integer between 1 and 32 inclusive

The second byte always contains `<20>` (see below for an explanation).

The remaining bytes contains the field types. The first byte contains the type of the first field, the second byte contains the type of the second field and so on. The following types are allowed:

- a type of `0` means that the field is an *integer*: a numeric value stored in two bytes, low byte first.
- a type of `1` means that the field is a *long*: a numeric value stored in 4 bytes.
- a type of `2` means that the field is a *double*: a floating point value stored in 8 bytes in standard IEEE format.
- a type of `3` means that the field is a *string*: a sequence of up to 255 bytes preceded by a byte giving the length of the sequence.

The format of all records

All records contain a two byte header followed by a sequence of bytes constituting the body of the record.

The highest nibble of the header word contains the record's type. The lowest three nibbles contain the length of the record body (a nibble is four bits, thus each byte consists of two nibbles). This explains why the second byte in the header of the field information record is always `<20>`.

In the example DBF file (see above) the record immediately following the field information record has <12><10> as its header and is thus a *type 1* record, with a body of length 0x12 bytes.

Counting past another 0x12 bytes leads to the header of the following record, which is also <12><10>. Counting along yet another 0x12 bytes leads precisely to the end of the file.

Since each of these records are *type 1* they must all conform to the internal structure specified in the field information record:

- a leading byte-counted string, for the first string field.
- a second leading byte-counted string.
- two bytes for the integer field.
- eight bytes for the double field.

Looking more closely at the above dump, it can now be appreciated how the details of the file contents match the earlier Opl program.

Generally speaking, the only types of record that will be found in any DBF file produced by a SIBO application are:

type 1	standard record.
type 2	field information record.
type 3	descriptive record.
type 0	deleted record.

Deleted records

Consider the following Opl program, which differs from the earlier one in only one line:

```
PROC writedbf:
    if exist("test.dbf") :delete "test.dbf" :endif
    create "test.dbf",a,f1$,f2$,f3%,f4
    a.f1$="Hello"
    a.f2$="2"
    a.f3%=3
    a.f4=4
    append
    a.f1$="World"
    update
    close
    beep 5,300
ENDP
```

The change is that the second `append` instruction has become an `update` instruction, so that the end result is that the file has only one record.

Dumping the DBF file output by this second program gives (provided the file is created on a Flash SSD):

```
 0:  4f 50 4c 44 61 74 61 62  61 73 65 46 69 6c 65 00   OPLDatab aseFile.
10:  0f 11 16 00 0f 11 04 20  03 03 00 02 12 00 05 48   ....... .......H
20:  65 6c 6c 6f 01 32 03 00  00 00 00 00 00 00 10 40   ello.2.. .......@
30:  12 10 05 57 6f 72 6c 64  01 32 03 00 00 00 00 00   ...World .2......
40:  00 00 10 40                                        ...@
```

This differs from the earlier dump only with regard to *one nibble* which gives the record type for the first record in the file. The two bytes <12><10> at file offset 0x1c have changed into <12><00>, indicating that while the same length of data remains on the file, the first record is no longer *type 1* but type 0, i.e. it has been *deleted*.

Erased records remain in a DBF file until such time as the file is written out again, say in response to a "Save As" menu command, or until the file is compressed (say in response to a "Compress" menu command). (In fact, Opl programs *automatically* attempt to compress their DBF files whenever they are closed, which explains why the above example gives different results unless the file is created on Flash.)

Although erased records may remain part of a DBF file long after the application has "deleted" or "updated" them, they are inaccessible to normal software (eg to the Plib `DbfXxx` calls and the Opl database functions such as `find` and `count`).

For more discussion about the mechanism of deleting records, see the *Database Files* chapter in the *Plib Reference* manual.

The remaining examples of DBF files in this chapter all assume that any deleted records have been removed.

Descriptive records

From most points of view, records of *type 3* and upwards are all potentially "singular" records without the usual software support. Operations such as "Find" do not usually find text within such records as these record types are not used by standard SIBO applications (with one important exception, discussed below). See the *Database Files* chapter in the *Plib Reference* manual for more details of these record types.

The exception is with a record of *type 3*, which is a so-called "descriptive" record. See later in this chapter for examples of data that may be stored in a descriptive record.

There is usually at most one record of *type 3* in any one DBF file.

By convention, a descriptive record's data is composed of a number of typed fields, with the types having variable meaning depending on the application. Given that different versions of an application may define different types of field within the descriptive record, it is good practice for applications that encounter fields in a descriptive record that they do not understand, to preserve these fields and to write them out again whenever the descriptive record needs to be changed.

More on type 1 records

It is not always necessary for a *type 1* record to have entries for each field defined in the field information record. Depending on the application, any trailing omitted fields will usually be assumed to be zero or null.

It is also possible for a record to have more than 32 fields. This is allowed in the case where the descriptive record explicitly defines 32 fields. In this case, any extra data in a record, beyond the 32nd field, is interpreted as a sequence of additional string fields.

The Series 3 Database

In this section reference to the Series 3 Database is taken to include the Series 3a Database except where explicitly stated otherwise.

The Series 3 Database stores its files in the DBF file format, as described in general terms at the beginning of this chapter. By convention, the Series 3 Database uses file with extension *.dbf*.

Field information record

The field information record of a Series 3 Database file is always as follows:

```
<20><20><03><03><03> ... <03><03><03>
```

there being 32 <03>'s in all. As mentioned above, this actually means that any *type 1* record in the file consists of a variable arbitrary number of string fields (where this number can exceed or fall short of 32).

Extended header

There are no extended headers on any Series 3 Database files.

Descriptive record

For an example of a descriptive record (*type 3*) consider the following dump of a default newly-created and exited Series 3 Database *.dbf* file (see below for a Series 3a example):

```
 0:  4f 50 4c 44 61 74 61 62  61 73 65 46 69 6c 65 00   OPLDatab aseFile.
10:  0f 10 16 00 0f 10 20 20  03 03 03 03 03 03 03 03   ......   ........
20:  03 03 03 03 03 03 03 03  03 03 03 03 03 03 03 03   ........ ........
30:  03 03 03 03 03 03 03 03  31 30 02 10 04 00 02 50   ........ 10.....P
40:  05 00 27 40 05 4e 61 6d  65 3a 07 05 20 48 6f 6d   ..'@.Nam e:.. Hom
50:  65 3a 07 05 20 57 6f 72  6b 3a 08 41 64 64 72 65   e:.. Wor k:.Addre
60:  73 73 3a 00 06 4e 6f 74  65 73 3a                 ss:..Not es:
```

The header of the first record after the field information record is <31><30>. This indicates that the record has *type 3* and body length 0x31 bytes and therefore extends to the end of the file.

Series 3a Database files are longer as the descriptive records contain more fields. Here is a newly-created and exited Series 3a Database *.dbf* file.

```
0:  4f 50 4c 44 61 74 61 62  61 73 65 46 69 6c 65 00   OPLDatab aseFile.
10:  0f 10 16 00 0f 10 20 20  03 03 03 03 03 03 03 03   ......   ........
20:  03 03 03 03 03 03 03 03  03 03 03 03 03 03 03 03   ........ ........
30:  03 03 03 03 03 03 03 03  aa 30 02 10 04 00 02 50   ........ .0.....P
40:  14 00 3a 60 82 2e c6 41  08 07 08 07 72 20 b6 33   ..:`...A ....r .3
50:  d0 02 d0 02 00 00 00 00  01 00 ff ff 00 00 00 00   ........ ........
60:  f0 00 00 00 00 00 00 00  f0 00 02 00 00 00 00 00   ........ ........
70:  00 00 00 00 00 00 f0 00  00 00 00 00 00 00 0d 70   ........ .......P
80:  00 52 4f 4d 3a 3a 42 4a  2e 57 44 52 00 0c 80 00   .ROM::BJ .WDR....
90:  00 00 25 50 00 82 2e c6  41 08 07 0c 90 25 50 00   ..%P.... A....%P.
a0:  82 2e c6 41 08 07 08 07  72 03 a0 01 00 01 04 b0   ...A.... r.......
b0:  00 00 ff ff 2e 40 05 4e  61 6d 65 3a 07 05 20 48   .....@.N ame:.. H
c0:  6f 6d 65 3a 07 05 20 57  6f 72 6b 3a 06 05 20 46   ome:.. W ork:.. F
d0:  61 78 3a 08 41 64 64 72  65 73 73 3a 00 06 4e 6f   ax:.Addr ess:..No
e0:  74 65 73 3a                                       tes:
```

The header of the first record after the field information record is <aa><30>. This indicates that the record has *type 3* and body length 0xaa bytes and therefore extends to the end of the file.

The body of a Series 3 Database descriptive record contains:

- a field of type 1 and length 2.
- a field of type 5 and length 2.
- a field of type 4 and length 0x27.

The body of a Series 3a Database descriptive record contains the above three fields and adds the following additional fields:

- a field of type 6 and length 58.
- a field of type 7 and variable length.
- a field of type 8 and variable length.
- a field of type 9 and variable length.
- a field of type 10 and length 3.
- a field of type 11 and length 4.

The above field types are as follows:

type 1	width of a tab in columns.
type 5	general flags options.
type 4	template data.
type 6	printer setup information.
type 7	printer model: a zero terminated string that starts with the model number and continues with the path for the printer driver file.
type 8	text for header: a zero terminated string.
type 9	text for footer: a zero terminated string.
type 10	diamond bar settings: bytes zero, one or two are set to 0xff if 'Find','Change' or 'Add' are included in the diamond bar respectively.
type 11	current search field: the first two bytes specify the start field with 0x00 meaning search in all fields, the second two bytes specify the end field with 0xff meaning search from start field to last field in record. Thus values of 0x10 and 0x20 would imply a search from field sixteen to field thirty two inclusive.

Other types may be used by other versions of the Database application (eg types 2 and 3 are used by the MC Database) and should be preserved intact when encountered.

In the above example it can be seen that the width of the tab is equal to 4 columns as given by the third and fourth bytes of the *type 1* field which starts at offset 0x3b (remember that the first two bytes are the record header). The value of the flags options is 0x05, and the contents of the labels can easily be read as a sequence of leading byte-counted strings.

Note that the first character in a template field name can be a telephone symbol (0x05) in which case each part-paragraph in the field matching the template entry can be used for automated dialling.

Flags options for the Series 3 Database

The meanings of possible bits in the flags options are as follows:

0x01	the permanent status window is on
0x02	entries are word-wrapped
0x04	a template should be displayed.

Type 1 records

Entries in the Database are stored as single *type 1* records in the corresponding *.dbf* file.

The following special (non-text) characters may occur within any string in a *type 1* record in a *.dbf* file, with these meanings:

0x15	forced line feed (with the two part-paragraphs on either side of the forced line break just counting as one paragraph for the purposes of the template, even though each part word-wraps separately)
0x05	prefixes a diallable telephone number (over and above any indication of diallability given in the template text)
0x14	if this occurs as the first character in a string, it means this field should in fact be joined together with the preceding one, as discussed in the following section.

Continuation sub-fields

The Series 3 Database uses a special mechanism in order to store fields of text exceeding 255 characters in length. Any such fields are broken down into *sub-fields* with at most 255 characters of text:

- the first 255 characters form the first sub-field
- up to the next 254 characters form the second sub-field, with the special character 0x14 being placed at the beginning of the string (and included in the preceding byte count)
- additional continuation sub-fields of up to 254 characters are used as required, in each case again being preceded by the 0x14 character.

The continuation sub-field prefix character was in fact specially chosen so as to give a suggestive display on the MC, should the *.dbf* file be read into the MC Database application.

The Series 3 Agenda

This section describes only the Series 3 Agenda. The Series 3a Agenda is described in a separate chapter.

The Series 3 Agenda stores its files in the DBF file format, as described in general terms at the beginning of this chapter. By convention, the Series 3 Agenda uses files with extension *.agn*.

Field information record

The field information record of a Series 3 Agenda file is always as follows:

<05><20><00><00><00><00><03>

indicating that each type 1 record in a *.agn* file consists of four integer fields followed by one string field. More details of these fields are given below.

Extended header

There are no extended headers on any Series 3 Agenda files.

Descriptive record

For an example of a descriptive record, consider the following, which is a dump of a default newly-created (and exited) Series 3 *.agn* file:

```
 0:  4f 50 4c 44 61 74 61 62  61 73 65 46 69 6c 65 00   OPLDatab aseFile.
10:  0f 10 16 00 0f 10 05 20  00 00 00 00 03 0e 30 0c   ....... ......0.
20:  f0 a4 01 3c 00 01 00 0f  00 1c 02 2e 00            ...<.... .....
```

The header of the first record after the field information record is <0e><30>. This means that the record has *type 3* and body length 0x0e bytes. Evidently, this record extends to the end of the file.

The body of this descriptive record is made up of a single field of *type 15* and length 0x0c. The body of this single field consists of 6 integers, with the following meanings:

- the time (in minutes since midnight) for the first appointment slot of the day
- the default length of an appointment (in minutes)
- whether or not alarms are on by default
- the default advance time (in minutes) before a timed appointment for an alarm
- the default time (in minutes since midnight) for the alarm for an untimed appointment
- the character (low byte only - the high byte is ignored) to be used as the time separator.

Evidently, these sub-fields match the various lines in the Settings dialog within the Agenda application.

Type 1 records

Entries in an Agenda are stored as single type 1 records in the corresponding *.agn* file.

Individual ToDo items and Repeated items are also stored as single type 1 records.

The five fields in Agenda type 1 records are

integer:	*DayNumber*
integer:	*Duration*
integer:	*Time*
integer:	*AlarmTime*
string:	*Text*

DayNumber is the number of days since 1/1/1900 (which is day zero). As far as the Series 3 Agenda is concerned, the first legal day is 1/1/1980, and the last legal day is 31/12/2049.

Duration, *Time*, and *AlarmTime* are all stored in minutes (ignoring for the moment the fact that some calculations with these values have to be performed in places - as described below).

The MSB (most significant bit) of *Time* is a flag that indicates whether the item is timed or untimed. If the MSB is set then the item is untimed. Accordingly, to extract the time from the *Time* field this value must be anded with 0x7fff to remove the MSB

The LSB (least significant bit) of *Duration* is a flag that indicates whether the item has an alarm attached. If the LSB is set then no alarm is attached. Accordingly, to extract the duration from the *Duration* field, divide this value by 2 (thus removing the LSB and shifting down the duration).

If the item is untimed then the *Time* field contains the *day note slot number* and its MSB must be set (ie or in 0x8000). In this case, the *Duration* field simply equals 0 if an alarm is attached or 0x01 if no alarm is attached.

Note that the end time of a timed item (obtained by adding its start time and its duration) must always be less than 1440 (ie midnight).

Calculating with AlarmTime

For internal efficiency reasons, the way in which the alarm pre-time (as defined by the user) is stored in the *.agn* file is somewhat counter-intuitive.

To recover the pre-time of the alarm for a timed appointment, the values of *Time* and *AlarmTime* for the entry should be added, and then the value (23*60+59) subtracted from this result.

For example, if the value of *Time* is 0x3fc and the value of *AlarmTime* is 0x1b2, the actual alarm pre-time is

```
0x3fc + 0x1b2 - (23*60 + 59)
```

ie 15 minutes (for an appointment actually at 5pm in the afternoon).

In the case of untimed appointments, the integer value obtained when *AlarmTime* is divided by 24*60 gives the number of days previous to the appointment when the alarm is due. The time of day when the alarm is due is obtained by subtracting the *remainder* when *AlarmTime* is divided by 24*60, from 23*60+59.

For example, if the value of *Time* is 0x8000 and the value of *AlarmTime* is 0xe87, note that

```
0xe87 = 2*(24+60) + 839
```

so that the alarm is due two days in advance of the appointment, at 10am (since 23*60+59-839=600).

If required, appropriate values of *AlarmTime* to ensure given alarm pre-times can easily be calculated by reversing the above formulae.

Finally, note that if an appointment does not have an alarm set for it, the value of *AlarmTime* should be set to 0xffff.

The text of an appointment

The text for an appointment is contained within the *Text* field of the record.

In all but the cases of repeated entries, the text is simply the entire contents of the field. For repeated items, the final six bytes of the *Text* field have a special meaning, as discussed below.

In all cases, the length of the actual text for an appointment cannot exceed 63 characters.

ToDo items

ToDo items have a *DayNumber* of 0xffff, and must be stored as timed items (and so the MSB of the *Time* field must be clear).

The *Time* field contains the item priority, that is a value from 1 to 9.

The *Duration* field contains a secondary integer key, which orders the ToDo items within a given priority.

ToDo items cannot have alarms attached.

Repeat items

Timed and untimed repeat items are stored in the same general way as normal timed and untimed items, except that they have a *DayNumber* of 0xfffe.

The specific repeat details are stored in the six bytes at the end of the *Text* field and have the following format:

Byte:	*Type*
Byte:	*Interval*
Integer:	*StartDayNumber*
Integer:	*EndDayNumber*

The *Type* field can take the following values:

0	Repeat yearly
1	Repeat monthly by date
2	Repeat monthly by day
3	Repeat weekly
4	Repeat daily
5	Repeat workdays.

StartDayNumber and *EndDayNumber* are stored as the number of days since 1/1/1900 (which is day 0). Setting *EndDayNumber* to zero means the item repeats "forever".

CHAPTER 5

SERIES 3A AGENDA FILE FORMAT

Introduction

Series 3a Agenda files are binary files containing typed variable length records. They use the same basic file and record structure as DBF files, but the file signature and internal record structure are NOT the same.

In contrast, Series 3 Agenda files are DBF files and are described separately, in the *DBF Files* chapter.

Basic structure of Agenda files

The basic file structure for Agenda files is as follows:

`<standard header>`	Used to identify the file type and the version of the file structure.
`<extended header>`	For future use, currently omitted.
`<data records>`	The main body of the file, containing entries and preferences.

The standard header

The standard header is present at the start of all Agenda files and is always 32 bytes long.

```
AGD_SIG_SIZE 16
AGD_SPARE_SIZE 12

typedef struct
    {
    UBYTE sig[AGD_SIG_SIZE];
    UWORD version;
    UWORD hSize;
    UBYTE spare[AGD_SPARE_SIZE];
    } AGD_FILE_HEADER;
```

The first 16 bytes of the file are always the zero terminated string `AgendaFileType*`. This is used to identify the file as an Agenda file.

The two byte parameter `version` at file offset `0x100f` should be interpreted as a hexadecimal word that gives the version of the file format. This is currently always *0x100F*. The most significant nibble (the version number is in the format described in the *General System Services* chapter of the *Plib Reference* manual) is the major version number and any change in this indicates that the file format may not be backwards compatible.

The two byte parameter `hSize` gives the combined size of the standard and extended header. This effectively gives the file offset of the first data record within the file. Currently this is always `0x0020`.

The array `spare` should not be used and is reserved for future use.

The extended header

At present this is never used and is reserved for future expansion.

The data records

As in DBF files all data is stored as variable length, typed records, with the type and length combined into a single word (two bytes). The most significant nibble of the word gives the record type. The type determines how the contents of the record are to be interpreted. The remainder of the word gives the length of the data that follows. This file structure is designed to be flash friendly in that deleted records are not normally removed from the file but are marked with record type 0 which can be done in place.

This structure allows 16 record types `0x0` to `0xf`, each of which are allowed to be up to `0xffe` (4094) bytes long. Although a record length of `0xfff` is not explicitly illegal it is not used.

Record Types

There are sixteen record types as follows.

Type	Record
0	Deleted
1	Appointments (timed day entries)
2	Day notes (un-timed day entries)
3	Anniversaries
4	To-do entries
5	Repeat records
6	Anonymous data
7	Reserved
8	Reserved
9	To-do list information
10	Descriptive records 1
11	Descriptive records 2
12	Descriptive records 3
13	Descriptive records 4
14	Descriptive records 5
15	Illegal (used to mark write failure)

Currently record types 6, 7 & 8 are never generated by the Series 3a Agenda.

A number of the records contain day numbers and times. Unless stated otherwise all dates are given as a *daynum*. A daynum is the number of days from 1 Jan. 1970. For technical reasons dates before 1 Jan. 1980 (*daynum* 3652) or after 31 Dec. 2049 (*daynum* 29219) are ignored by the Agenda and where appropriate will be 'clipped' to one or other of these dates (for example a repeating entry that starts on 10 June 1970 will have its start date 'clipped' to 1 Jan. 1980).

Type 0 - deleted record

As for DBF files, deleted records are not normally removed from the file but have their record type changed to type 0. As any of the above record types may be converted to a type 0 record, there is nothing that can usefully be said about the contents of such a record (indeed the record may not even have been a valid Agenda record before it was deleted). Records of this type should be ignored except to calculate the amount of space that would be freed by compressing the file.

There may be any number of such records in the file.

Types 1 to 4 - entry records

Records of type 1 to 4 contain details of individual Agenda entries. Each has the same conceptual structure as follows:

`<Entry details field>`	This is dependent on the entry type.
`<Title field>`	Obligatory, variable length field containing the text of the entry.
`<Alarm field>`	Optional, fixed length field containing alarm time and sound.
`<Memo field>`	Optional variable length field containing any memo for the entry.

The record type is used to determine the length and meaning of the first field. Although these have some similarities all four are individually described in details below.

Entry details field

This field contains most of the non-textual information describing when an entry occurs, what other fields it has and various bits of type specific information, such as the duration for timed appointments.

Type 1 (timed day entry/appointment)

A type 1 record stores details of entries that occur at a specific time on a specific day. In the Series 3a Agenda these are called *timed day entries*.

The *details field* for a *timed day entry* consists of eight bytes structured as follows:

```
UWORD day;
UWORD time;
UBYTE attr;
UBYTE code;
UWORD dur;
```

`day`	is the *daynum* of the day on which the entry appears.
`time`	is the time of the start of the appointment in minutes from midnight.
`attr`	is a byte containing flags for attributes the entry may or may not have (see below).
`code`	is the ASCII character code for the symbol that is to be associated with the entry when it is visible in the Year view. Values less than 32 are ignored and treated as if the entry should not appear in the Year view.
`dur`	is the duration of the appointment measured in minutes. It is constrained such that the appointment cannot end after 11:59 PM. i.e. this field is between 0 and 1439 - time (inclusive).

Type 2 (untimed day entry/note)

A type 2 record stores details of entries that appear on a specific day but do not have a time associated with them. In the Series 3a Agenda these are called *untimed day entries*.

The *details field* for an *untimed day entry* consists of six bytes structured as follows:

```
UWORD day;
UWORD slot;
UBYTE attr;
UBYTE code;
```

`day`	is the *daynum* of the day on which the entry appears.
`slot`	is the time slot (in minutes from midnight) in which the entry will appear in the Day and Week views. For example a `slot` value of 780 would show the entry at the start of the 1pm slot. If this is `0xffff` then the entry will appear in the default slot.

`attr` is the attributes byte (see below).

`code` is the ASCII character code for the symbol that is to be associated with the entry when it is visible in the Year view. Values less than 32 are ignored and treated as if the entry should not appear in the Year view.

Type 3 details (anniversaries)

A type 3 record stores details of anniversaries (entries which appear in the Anniversary view). Although these are usually repeated there are cases where a single *anniversary entry* will exist. For details of repeated entries see type 5 records below.

The details field for an anniversary record consists of nine bytes structured as follows:

```
UWORD day;
UWORD slot;
UBYTE attr;
UBYTE code;
UWORD baseYear;
UBYTE displayAs;
```

`day` is the *daynum* for the day the anniversary entry will appear on.

`slot` is the time slot (in minutes from midnight) in which the entry will appear in the Day and Week views. For example a `slot` value of 780 would show the entry at the start of the 1pm slot. If this is `0xffff` then the anniversary will appear in the default slot.

`attr` is the attributes byte (see below).

`code` is the ASCII code for the character to display when the entry is visible in the Year view. Values less than 32 are illegal and cause the entry not to appear in the Year view.

`baseYear` is the year of the event that the anniversary commemorates. Positive values indicate AD years e.g. 55 means 55 AD and negative values indicate BC e.g. -5 means 5 BC. A value of zero indicates that there is no `baseYear`. The allowed range is from 30000 BC to 2049 AD

`displayAs` contains flags detailing how the entry is to be displayed. The flags are as follows: `0x01` for baseYear displayed, `0x02` for elapsed years displayed, `0x03` for both of the preceding options and `0x00` for none of them.

Type 4 details (To-do)

A type 4 record holds details of to-do entries.

These are entries which usually have an associated due date and a display from date. They appear in the corresponding to-do list, and depending on the preference settings will appear in the Day view from the *displayFrom* date until they are crossed out or deleted.

The details field for a to-do entry consists of 14 bytes structured as follows:

```
UWORD displayFrom;
UWORD slot;
UBYTE attr;
UBYTE code;
UWORD dueDate;
UBYTE listNo;
UBYTE priDisp;
ULONG order;
```

`displayFrom` is the *daynum* of the day on which the entry first appears in the Day/Week views of the Agenda. This must normally be the same or less than the `dueDate` value.
If the entry is crossed out (see the description of the `attr` byte below) this is the *daynum* of the day on which the entry was crossed out. In this case, and only in this case, the `displayFrom` can be later than the `dueDate`.
If the `displayFrom` *daynum* is `0xffff`, then this is an un-dated to-do i.e. one that always appears on today. In this case the `dueDate` will also be `0xffff`.

`slot`	is the time slot (in minutes from midnight) in which the entry will appear in the Day and Week views. If this is `0xffff` then the entry will appear in the default slot of the appropriate to-do list.
`attr`	is the attributes byte (see below).
`code`	is the ASCII code for the character to display when the entry is visible in the Year view. Values less than 32 are illegal and cause the entry not to appear in the Year view.
`dueDate`	is the *daynum* of the day the to-do should be done by. If this value is `0xffff` the record is an undated to-do (i.e. one which always appears on *today* when it appears in the Day/Week views).
`listNo`	is the internal number (0-255) of the to-do list on which the entry will appear. Note that a value of zero does *not* necessarily mean that the entry appears on the first to-do list in the To-do view. See type 9 records for further details of the meaning of this byte.
`priDisp`	this byte consists of two nibbles that give the priority and the method of displaying the to-do. The least significant nibble (bottom four bits) has a value one less than the priority of the to-do. Thus a priority one to-do has value zero, priority two has value one etc. This will currently always be in the range zero to eight inclusive and all other values are illegal. The most significant nibble (top four bits), determines how the due date should be displayed there are currently four legal values: 0 - automatic, shown as date until within a week then shown as e.g. Next wed. 1 - Always shown as date. 2 - Shown as number of days until due date. 3 - Due date never shown.
`order`	this determines the position of the entry in its to-do list when the list is displayed in manual order. This field is not assigned consecutively. Thus a value of 3 for example in this field does *not* necessarily mean that the entry appears third (or fourth) on the to-do list.

The attributes byte '*attr*'.

The above record types (1 to 4) contain an attributes byte as the fifth byte within the record data. This byte contains flags which indicate which of the two optional fields are present in the record, whether or not the entry is repeated etc. The following flags are currently defined for the attributes byte (all other bits should be 0).

Bit	Meaning
00000001	Once only: if this bit is set the entry appears only once in the Agenda. If it is clear the entry is repeating and there will be an associated type 5 repeat record elsewhere in the file.
00000010	Pending: this bit is set if the entry has not been crossed out. If it is clear the entry has been crossed out. In the case of to-do entries this alters the meaning of the *displayFrom* field of the details.
00000100	Display code: if this bit is set the entry should be displayed in the Year view (providing it has a legal code field): if it is clear the entry should not appear in the Year view, regardless of the value of the code field.
00001000	No alarm: this bit is set if the entry does not have an associated alarm in which case there will be no alarm field at the end of the record. If it is clear there is an alarm field immediately after the title field.
00010000	No memo: this bit is set if there is no memo associated with the entry, in which case there will be no memo field at the end of the record.

The title field

The *title field* must be present for entry records of all four types. This field follows immediately after the *details field* and contains the text for the entry as well as flag for how that text should be displayed e.g. bold, italic etc.

The format of the title field is:

```
UBYTE style;
UBYTE len;
title text.
```

`style` contains flags indicating how the text should appear. The flags are as follows: `0x01` for bold, `0x02` for underline, `0x20` for italic.

`len` is the length of the text comprising the title, and the number of bytes following. This may take any value from 0 - 254.

`title text` is `len` bytes of the title. Note that this text is *not* zero terminated.

The alarm field

This fixed length field is only present if the attributes byte does not have bit 3 (`0x08`) set, i.e. there is an alarm set for the entry.

When present this entry immediately follows the title field and has the following format:

```
UWORD preTime;
UBYTE len;
UBYTE sound[8];
```

`preTime` is the time at which the alarm should occur given in minutes before 11.59 on the day the entry appears on. (In the case of a type-4 to-do record this is the due date). This field can be between 0 and 46079 (midnight before, 31 days before the entry).

`len` this gives the length of the text in the sound element.

`sound` is always eight bytes long the first `len` of which contain the name of the WVE file for the alarm. A number of WVE files having names of the form *SYS$ALnn* are built-in to the *ROM*. In addition name can be set to an ASCII character with value between 1 and 16 inclusive: currently only "\0x01", "\0x02" and "\0x10" are used corresponding to the rings, chimes and silent alarms. If the name of the file is less than eight bytes long, all remaining bytes should be zero.

The memo field

This variable length field is only present if the attributes byte (in the *details field*) does not have bit 4 (`0x10`) set, i.e. there is a memo attached to the entry.

When present this field comes at the end of the record immediately after the *alarm field* if there is one, and after the *title field* if there is not. The field has the following format:

```
UWORD dataLength;
UBYTE data[];
```

`dataLength` is the length (in bytes) of the data in the memo field. This can be between 0 and 3600 (inclusive).

`data` is a block of data consisting of `dataLength` bytes containing the memo - details of the memo are beyond the scope of this manual.

Type 5 - repeats

When an entry is set to repeat in the Agenda, a second record is written to the file in addition to the type 1 - 4 entry record. This record contains the date to repeat until, the days to repeat on and a list of those dates for which the repeat should be suppressed.

Repeat records are always paired with an entry record which has bit 0 (`0x01`) of the attributes byte clear. Because of the way the Series3a Agenda writes the file a type 5 repeat record will always occur after the associated entry record, but this is not necessary and there is no reason why it should not occur earlier in the file. If a repeat record is missing its associated entry record or if a repeated entry record exists for which there is no associated repeat record, then the record in the file will be ignored.

The repeat record is structured as follows:

```
UBYTE alg;
UBYTE ival;
UWORD endDate;
UBYTE type;
UBYTE tags[n];
ULONG filePos;
UWORD exceptions[];
```

`alg` is the repeat algorithm and various flags. The bottom three bits of this byte may take one of the following values: 0 - repeat daily, 1 - repeat weekly, 2 - monthly by date, 3 - monthly by days, 4 - repeat annually.
If bit 3 (`0x08`), of this byte is set, the repeat should only appear once in the dated views on the first occurrence after today. Otherwise all valid occurrences are shown. All other bits in this byte should be 0.

`ival` is the daily repeat interval. This is zero if the entry repeats daily, 1 if the entry repeats every other day etc. A value of 255 is *not* valid.

`endDate` this is the *daynum* of the last day on which the entry can repeat. This is not necessarily the last day on which it will appear. The start date is taken from the `day` value at the start of the details field in the entry record. In the case of a *repeating to-do* this is the `displayFrom` field that is normally used to give the date from which to display the entry. Here it is used to determine the start date of the repeat algorithm and hence determine which due dates will be associated with the todos. To work out the display from dates for each instance of the repeated to-do use the `dueDate-displayFrom` in the entry record to determine the number of days warning for each instance of the repeat.

`type` is the type (1-4) of the associated entry record. This information is actually redundant since it can be determined by reading it directly once the associated record has been found but is a useful optimisation internally to the Agenda.

`tags` is `n` bytes that determine which days occur in the repeat sequence. The number of bytes `n` and their meaning is determined by the repeat algorithm. For repeat daily/annually there are no tag bytes since the start date and `ival` determine which dates to repeat over.
For *weekly repeats* there are two tag bytes. The first has a bit set for each day of the week on which the repeat occurs. Bit 0 for Monday, bit 1 for Tuesday etc. Bit 8 is not used and should always be 0. The second byte determines which is the first day of the week, 0 for Monday, 1 for Tuesday etc. This is significant when the repeat does not occur every week. A repeat which occurs on say, Tuesday and Thursday of every other week, occurs on different dates if the week starts on Monday than it does if the week starts on Wednesday.
For *monthly by date repeats* there are four tag bytes. Each bit in these bytes represents a different day of the month. Bit 0 in the first byte is the 1st of the month, bit 1 the second and bit 7 the eighth of the month. Bit 0 of the second tag byte is set if the repeat occurs on the 9th and so on. Bit 7 of the fourth byte is not used and should be 0.
Monthly by days repeats have five tag bytes. Bytes 0 to 3 correspond to the first, second, third and fourth occurrences of each day, for example bit 1 of byte 2 is set if the algorithm repeats on the 3rd Tuesday of each month. The last tag byte contains bits set if the repeat should occur on the last Monday, say, of the month.

`filePos` is the offset from the start of the file at which the corresponding entry record can be found. This is given in bytes from the start of the file (not the start of the data records), and gives the position of the type length word at the start of the record. Note that this mechanism of associating repeat records with the underlying entry relies on the deleted entries not being removed from the file: they are just marked with the deleted record type. Removing such records would require all `FilePos` fields to be recalculated.

`exceptions` The remainder of the record consists of words, each of which is the *daynum* of a day on which the repeat should be suppressed. The number of exceptions is determined by the length of the record. Although the Series 3a Agenda currently always writes these exceptions in strictly increasing order, there is no guarantee that this will always be the case. Any illegal values (either outside the valid range for the Agenda or on a day which is not normally a repeat instance) should be ignored but preserved.

Type 6 - anonymous data

This record type is currently not used and is set aside for storing non-displayable text carried with the Agenda. It is intended that this information will be used by, say, conversion programs that convert other Agenda file formats to that of the Series 3a. This allows the file to be converted back without losing information from the original file. These records are ignored by the series 3a engine except when merging files. In this case incoming anonymous data records are added to the file into which data is being merged.

Types 7 and 8 - reserved

These record types are reserved for future expansion and should not be used.

Type 9 - to-do list information

There is one record of this type for each to-do list in the Agenda. Each contains the setting for the to-do list and the to-do list number (as in type 4 to-do records) that corresponds to the list.

The format for these records is as follows:

```
UBYTE sig;
UBYTE data[]
```

`sig` a signature byte which determines the format of the rest of the record. Currently the only legal value is `0xff`.

`data` is data for the to-do list: the details of the data are beyond the scope of this manual.

Types 10 to 14 - descriptive records

Type 10 to 14 records contain the preferences settings for the Agenda (these may be set via the preferences dialog). Only one record of each type is allowed per Agenda file. If there is more than one record of any type, only the last record is significant.

A descriptive record consists of the usual header word followed by the record body. The record body contains either simple data or a set of *type/length/value* (*TLV*) fields. A *TLV* field consists of a one word header followed by the field body. The top four bits of the field header contain the field type. The bottom 12 bits contain the length of the field body. No more than one of each of the specified fields can be present in a given record.

The structure of these records should not be extended (extra *TLV* fields should not added even though these would be ignored).

Type 10 to 14 records are described in greater detail below.

Type 10 - styles descriptive record

The styles record contains the Memo editor preferences and its styles and emphases.

The styles record gets written after a memo has been created for the first time, and thereafter whenever its content has changed.

The details of this record are beyond the scope of this manual.

Type 11 - to-do manager descriptive record

This record is written on creation of a new Agenda and so is always present. It gets rewritten whenever changes have been made.

This record contains the information that the Agenda needs to determine which position each to-do list has in the To-do view. The record consists of a header word followed by a body of three bytes structured as follows:

```
UBYTE sig;
UBYTE ncats;
UBYTE catid[ncats]
```

`sig` is the signature and is *always* `0x6c`.

`ncats` is the number of categories.

`catid[ncats]` contains ids for each category (i.e. to-do list) in the display order. Thus `catid[0]` holds the id of the first displayed category, `catid[1]` the id of the second, and so on up to a maximum of `catid[98]`. These are used to identify each record as coming from a given category in a record.

Type 12 - frequently changing data descriptive record

This record is written on creation of a new Agenda and so is always present. However, as it contains data that changes frequently, it is only rewritten when the file is closed.

This record contains zoom, wrap and status window settings for each view. It consists of a header word followed by six `VIEW_SCREEN_CFG` structures for the Day, Week, Year, To-do, Anniversary and List views respectively. A `VIEW_SCREEN_CFG` structure consists of three bytes and is structured as follows:

```
UBYTE statmode;
UBYTE wrapmode;
UBYTE zoom;
```

`statmode` this field is a flag for the status window. It is 0 if a status window is not visible, 1 if the small status window is visible and 2 if the large status window is visible. This flag applies to all views.

`wrapmode` in all views except the Year view the `wrapmode` field is 1 if wrap is on, and is otherwise 0. In the Year view (which does not use wrapping) the `wrapmode` field contains the index of the month that is displayed in the first row of the planner (0 = January, 11 = December).

`zoom` this field takes the value 0 to 3 corresponding to Roman fonts of height 8, 11, 13 and 16 respectively. It is relevant for the Day, Week, To-do, Anniversary and List views. The `zoom` field is unused in the Year view entry and is set to 0.

Type 13 - general descriptive record

This record is written on creation of a new Agenda and so is always present. It stores data shown in the dialogs under the Preferences menu. It gets written whenever the values shown in one of these dialogs are changed.

This record contains entry defaults and all individual view preferences (to-do entry defaults are stored in to-do list records). Each record has the normal Agenda *type/length* header followed by one or more *TLV* fields. There are sixteen possible type fields only some of which are currently used: the rest are reserved for future use. If a field is not found the Agenda will use the corresponding default values. The fields can be in any order.

Type	Field
4	Diamond list setup
5	Day entry defaults
6	Anniversary defaults
7	General defaults
9	Day view settings
10	Week view settings
11	Year view settings
12	To-do view settings
13	Anniversary view settings
14	List view settings

Diamond list setup field

The diamond list setup field indicates which views should be included in the diamond list. It consists of six bytes structured as follows:

```
UBYTE fnbar[6];
```

`fnbar` contains six bytes corresponding to the Day, Week, Year, To-do, List and Anniversary views respectively. Each byte is either `TRUE` or `FALSE` depending on whether or not the view is to be included in the diamond list.

Day entry defaults field

The day entry defaults field contains the defaults for entries in the Day view. The field consists of 38 bytes structured as follows:

```
UWORD DefUntimedEntViewTime;
UWORD DefTimedEntTime;
UWORD DefTimedEntDuration;
UBYTE DefTimedByDefault;
UBYTE DefYearSym;
AGPREF_ALARM UntimedAlarmDefs;
AGPREF_ALARM TimedAlarmDefs;
UBYTE style;
UBYTE spare;
```

`DefUntimedEntViewTime` is the default display time for untimed entries in minutes since 00:00.

`DefTimedEntTime` is the default display time for timed entries in minutes since 00:00.

`DefTimedEntDuration` is the default duration for a timed entry in minutes.

`DefTimedByDefault` is 1 if entries are timed by default, otherwise it is 0.

`DefYearSym` is the character code for the default year symbol.

`UntimedAlarmDefs` contains details of the default alarm for untimed entries (see below for a description of the `AGPREF_ALARM` structure).

`TimedAlarmDefs` contains details of the default alarm for timed entries (see below for a description of the `AGPREF_ALARM` structure).

`style` is the style of the font used for the entry. It should be a suitable `OR`ed combination of the Wserv `G_STY_XXX` flags: `0x00` for normal, `0x01` for bold, `0x02` for underline and `0x20` for italics.

spare — is reserved and is set to 0.

An AGPREF_ALARM structure contains default alarm details. It consists of 14 bytes structured as follows:

```
UBYTE on;
UBYTE ndays;
UWORD minutes;
SE_SND snd;
```

on — is 1 if an entry has an alarm by default and 0 otherwise.

minutes — for timed entries minutes is the time interval in minutes between the alarm going off and the start of the entry. For untimed entries minutes is the default alarm time in minutes from midnight.

ndays — for timed entries ndays is unused but should be set to 0. For untimed entries ndays is the default number of days between the alarm going off and the start of the entry.

snd — contains details of the default alarm sound (see below for a description of the SE_SND structure).

The SE_SND structure contains details of the default alarm sound. It consists of ten bytes structured as follows:

```
UBYTE len;
TEXT name[8];
UBYTE zero_term;
```

len — is the length of the name of the alarm.

name — contains the name of the alarm stored as a sequence of len characters. When len is less than eight the first unused byte contains a NULL character. Any remaining unused bytes can take any value.

zero_term — contains the NULL character.

Anniversary entry defaults field

An anniversary entry defaults field contains the defaults for entries in the Anniversary view. It consists of 20 bytes structured as follows:

```
UWORD DefEntViewTime;
UBYTE AutoApplyYearSym;
UBYTE DefYearSym;
AGPREF_ALARM AlarmDefs;
UBYTE style;
UBYTE spare;
```

DefEntViewTime — is the default display time for anniversaries in minutes since midnight.

AutoApplyYearSym — is 1 if the year symbol is on by default, otherwise it is 0.

DefYearSym — is the character code of the default year symbol.

AlarmDefs — contains the default alarm details for timed and untimed entries (see the *Day entry defaults field* for details of the AGPREF_ALARM structure).

style — is the style of the font used for the entry. It should be a suitable ORed combination of the Wserv G_STY_XXX flags: 0x00 for normal, 0x01 for bold 0x02 for underline and 0x20 for italics.

spare — is reserved and is set to 0.

General defaults field

```
UBYTE p_psion_enter;
UBYTE timesep;
```

p_psion_enter — is set to TRUE if PSION+ENTER is used to complete an entry without going into the Entry details dialog. Otherwise p_psion_enter is FALSE.

timesep — is the character code for the Agenda time separator character.

Day view settings field

```
UWORD agnv_flags;
ADENTVU_PREF left;
ADENTVU_PREF right;
```

`agnv_flags` this field contains both general view flags and Day view flags (all bits not used are reserved). The general view flags are as follows:
`0x01` for show appointment duration (Day, List, Week and Year views),
`0x02` for show appointment end time (Day, List, Week and Year views only),
`0x100` for show untimed day notes (Day, List and Week views only),
`0x200` for show anniversaries (Day, List and Week views only),
`0x400` for show to-dos (Day, List and Week views only) and
`0x800` for show timed day notes (Day, List and Week views only).
The Day view flags are as follows:
`0x04` for title to go on right hand side,
`0x08` for slot compression off,
`0x10` for duration arrows off and
`0x20` for show overlap bars off.

`left` contains the preferences for the left hand side of the Day view (see below for a description of the `ADENTVU_PREF` structure).

`right` contains the preferences for the right hand side of the Day view (see below for a description of the `ADENTVU_PREF` structure).

An `ADENTVU_PREF` structure consists of 12 bytes structured as follows:

```
UWORD flags;
UWORD begintime;
UWORD beginvis;
UWORD endvis;
UWORD endtime;
UWORD slotdur;
```

`flags` contains a combination of the slot lines on flag (`0x01`) and the slot times on flag (`0x02`).

`begintime` is 0 for the left hand side and `beginvis` for the right hand side.

`beginvis` is the start time of the first slot in minutes from midnight.

`endvis` is the start time of the last slot in minutes from midnight.

`endtime` is right.beginvis for the left hand side and 1440 for the right hand side.

`slotdur` is the slot duration in minutes.

Week view settings field

The Week view settings field consists of two bytes as follows:

```
UWORD pref_flags;
```

`pref_flags` contains both general view flags (see above under the *Day view settings field*), and Week view flags. Currently there is only one Week view flag: `0x4` for show title on right hand side.

Year view settings field

The Year view settings field consists of two bytes as follows:

```
UWORD pref_flags;
```

`pref_flags` contains a combination of the general flags (see under the *Day view settings field* above) for showing appointment duration and/or end time or neither.

To-do view settings field

The To-do view settings field consists of two bytes structured as follows:

```
UWORD ncols;
```

`ncols` contains the number of columns to be shown in the To-do view. The value of `ncols` should not be more than the number of existing categories.

Anniversary view settings field

The Anniversary view settings field consists of two bytes structured as follows:

```
UWORD ncols;
```

`ncols` contains the number of columns to be shown in the Anniversary view (1 to 4).

List view settings field

The List view settings field consists of two bytes structured as follows:

```
UWORD pref_flags;
```

`pref_flags` contains a combination of general view and Day view flags (see under the *Day view settings field* above) and the show-repeats-once flag (0x04).

Type 14 - print setup descriptive record

This record does not initially exist. It is created/rewritten to contain a new copy of the data for the Agenda print setup after using the Agenda Print setup dialog. It is also created/rewritten after a memo has been created or edited and Print setup data has been changed. Hence it is possible that the record will only contain Agenda Print setup data or Memo Print setup data.

The record consists of a one word header followed by a series of *TLV* fields.

Field types 0 to 3 and 6 to 9 are allowed. Field types 0 to 3 are used for the main Agenda print setup. Field types 6 to 9 are identical to field types 0 to 3 and are used for the memo print setup.

Field type 0 and 6

Contains printer parameters in a `PRINTER_PARAMS` structure as returned by the `PR_GET_PARAMS` method of the printer active object of the FORM dyl.

Field types 1 and 7

Contain printer model data from the `PR_SENSE_MODEL` method of the printer active object. The first byte is the model number as returned by `PR_SENSE_MODEL`, followed by the name, up to and including the `NULL`.

Field types 2 and 8

Contain printer header text from the `PR_GET_HD` method of the printer active object including the terminating `NULL`.

Field types 3 and 9

Contain printer footer text from the `PR_GET_HD` method of the printer active object including the terminating `NULL`.

Type 15 - illegal

This record type is illegal and is used to protect the Agenda against write failures on a flash SSD. Writing to a flash SSD can fail at any time due to, say, a low battery. To protect as much as possible against this the Agenda will write the whole of the rest of the record before 'blowing down' the byte containing the record type to its correct value. This means that any file which contains a record with type 15 (0xf) has suffered a write failure and data beyond this point cannot be trusted.

This record type is probably best thought of as an End Of File record, and any program finding a file with a type 15 record should start by setting the end of the file to the start (the type length word) of the record.

CHAPTER 6

WORD PROCESSOR FILE FORMAT

This document describes the structure of a Psion word processor document to a degree which allows external software to read and write non-password-protected documents.

Psion word processor document files contain a file header, followed by a number of type-length-value records, from the following list:

- options data
- printer-related data
- printer model data
- page header
- page footer
- style data
- emphasis data
- document text
- document index

Document files created by the word processor will contain records in the order listed above. Each record consists of:

- a two-byte record `type`
- a two-byte record length, `len`
- `len` bytes of data

Unless stated otherwise, all dimensions are stored in twips (1/1440 inch).

Reserved locations

All locations reserved for future use contain a value of zero.

The document header

The document header is 40 bytes long. It starts with the 16-byte zero terminated file signature "PSIONWPDATAFILE" followed by a two byte file version number.

Following this is a 20-byte struct, containing password data. For a non-password-protected document there are two bytes of zero followed by eighteen bytes, each containing `0xEA`.

The remaining two bytes of the header are reserved for future use.

Record types

The options data record (type 1, length 10)

The options data record contains the following items:

`UWORD cpos`	saved cursor position
`UBYTE symbols`	show screen symbols
`UBYTE backup`	`TRUE` to keep backups (this item is used on the MC only: it is not used on Series3 machines)
`UBYTE statzoom`	top four bits indicate the current zoom state (0x22 by default), lowest four bits indicate the status window size: 0, 1 or 2 for off, small or big respectively (this item is used by the Series3a only)
`UBYTE style`	`TRUE` to show style bar
`UBYTE mono`	`TRUE` to load text by line, else by paragraph
`UBYTE outlevel`	lowest outline level to display
`UBYTE spare1`	reserved for future use
`UWORD spare2`	reserved for future use

Displayed screen symbols are determined by any combination of the following bit fields in `symbols`:

0x01	show tabs
0x02	show spaces
0x04	show paragraph end markers
0x08	show hyphens
0x10	show forced line breaks

Printer-related data record (type 2, length 58)

This record contains information required to format the document for the printer and to control the printing process. It includes, amongst other items, descriptions of the page size and margins, the page numbering style, header and footer position and alignment.

This record consists of a structure that is used to control the display and WDR printing of formatted text in all applications that require such services. It therefore contains some fields that are not relevant to the word processor.

The following description is in terms of the `P_EXTENT`, `SCRLAY_FONT` and `PAGES_HEADER` structs, defined as:

```
typedef struct
    {
    WORD x;
    WORD y;
    } P_POINT;

typedef struct
    {
    P_POINT tl;
    WORD width;
    WORD height;
    } P_EXTENT;

typedef struct
    {
    UWORD fid;          /* typeface number */ (1)
    UWORD style;        /* font style */ (2)
    UWORD height;       /* height of font in twips */
    } SCRLAY_FONT;
```

```
typedef struct
    {
    SCRLAY_FONT f;      /* font data */
    UBYTE align;        /* header alignment */ (3)
    UBYTE first_page;   /* TRUE to emit on first page */
    } PAGES_HEADER;
```

In these terms, the content of the printer-related record is:

PAGE DATA

WORD width	page width
WORD height	page height
P_EXTENT body	body print region, with respect to top left of page (4)
WORD hdtop	page header position (5)
WORD hdbot	page footer position (6)
WORD pdrflags	0 for portrait, 1 for landscape
WORD docflags	always 3 for word processor
UWORD pgbeg	page number to start printing (first page is 1)
UWORD pgend	last page number to print (7)

RUNNING PAGE HEADERS

PAGES_HEADER top	page header
PAGES_HEADER bot	page footer

PAGE NUMBERING

WORD offset	page numbering offset
WORD last	page count, for %m (always reset by pagination)
WORD style	page number style (8)

MISCELLANEOUS

SCRLAY_FONT f	base font for body print region (9)
UBYTE size_choice	paper size index (10)
UBYTE wo_control	TRUE to disable widow and orphan control
UWORD spare1	reserved for future use
UWORD spare2	reserved for future use

Notes

(1) Typeface numbers are defined in the following list:

0 COURIER	22 OPTIONAL_SB	44 RUSSIAN
1 PICA	23 OPTIONAL_SC	45 OPTIONAL_B
2 ELITE	24 TIMES_ROMAN	46 OPTIONAL_C
3 PRESTIGE	25 CENTURY	47 OPTIONAL_D
4 LETTER_GOTHIC	26 PALATINO	48 NARRATOR
5 GOTHIC	27 SOUVENIR	49 EMPHASIS
6 CUBIC	28 GARAMOND	50 ZAPF_CHANCERY
7 LINEPRINTER	29 CALEDONIA	51 OPTIONAL_DA
8 HELVETICA	30 BODONI	52 OLD_ENGLISH
9 AVANT_GARDE	31 UNIVERSITY	53 OPTIONAL_DB
10 SPARTAN	32 SCRIPT	54 OPTIONAL_DC
11 METRO	33 SCRIPT_PS	55 COOPER_BLACK
12 PRESENTATION	34 OPTIONAL_SCA	56 SYMBOL
13 APL	35 OPTIONAL_SCB	57 LINE_DRAW
14 OCR_A	36 COMMERCIAL_SCRIPT	58 MATH_7
15 OCR_B	37 PARK_AVENUE	59 MATH_8
16 STANDARD_ROMAN	38 CORONET	60 DINGBATS
17 EMPEROR	39 OPTIONAL_SCC	61 EAN
18 MADELEINE	40 GREEK	62 PC_LINE
19 ZAPF_HUMANIST	41 KANA	63 OPTIONAL_SYA
20 CLASSIC	42 HEBREW	
21 OPTIONAL_SA	43 OPTIONAL_A	

A typeface that is not supported by the current printer will be mapped to a supported typeface.

(2) The font style may be zero, or any sensible combination of the following attributes:

0x01	underline
0x02	bold
0x04	italic
0x08	superscript
0x10	subscript

Styles or style combinations that are not supported by the current printer are ignored.

(3) The running page header alignment may be any one of:

0	left aligned
1	right aligned
2	centred
4	2-column
5	3-column

(4) In terms of the `body` struct and the page `width` and `height`, the page margins are:

left	`body.tl.x`
top	`body.tl.y`
right	`width-body.tl.x-body.width`
bottom	`height-body.tl.y-body.height`

(5) The page header position measures the vertical distance between the bottom of the header text and the top edge of the page body print region.

(6) The page footer position measures the vertical distance between the bottom of the footer text and the bottom edge of the page body print region.

(7) To print all pages, the last page number should be set to `0xffff`.

(8) The page number style is one of:

0	Arabic
1	Roman, upper case
2	Roman, lower case.

(9) The body area base font data is not used by word processor documents. It should always specify the default font, with font number 0 (Courier) a style of 0 (normal) and a size of 240 (12 points).

(10) The page size choice should correspond with the earlier page width and height. The following (English) page sizes are recognised:

Choice	Page type	Width	Height
0	A4	11906	16838
1	Custom	-	-
2	Executive	10440	15120
3	Legal	12240	20160
4	Letter	12240	15840
5	Monarch	5580	10800
6	DL	6236	12472

Note that items 2 to 6 may be different in non-English versions of the software.

Printer model data record (type 3, variable length)

The record consists of a zero terminated string, containing the full path name of the current printer driver file, preceded by a one byte index to the printer model within the file.

The default value is:

```
0
"ROM::\BJ.WDR"
```

Page header record (type 4, variable length)

The record contains the page header text as a zero terminated string. The string may not exceed 80 bytes.

Page footer record (type 5, variable length)

The record contains the page footer text as a zero terminated string. The string may not exceed 80 bytes.

Style data record (type 6, length 80)

The file may contain up to 64 such records, each of which defines a single style.

The following description is in terms of the SCRLAY_FONT struct (described above) and the SCRLAY_MARGINS, SCRLAY_SPACING and SCRLAY_TABSTOP structs, defined as:

```
typedef struct
    {
    UWORD left;         /* left margin */ (1)
    UWORD right;        /* right margin */
    UWORD indent;       /* left margin for first line of a paragraph */
    UWORD align;        /* alignment */ (2)
    } SCRLAY_MARGINS;

typedef struct
    {
    UWORD line;         /* space between lines in a paragraph */
    UWORD above;        /* space above paragraph */
    UWORD below;        /* space below paragraph */
    UWORD flags;        /* keep together/next and new page */ (3)
    } SCRLAY_SPACING;

typedef struct
    {
    UWORD x;            /* tab position */
    UWORD type;         /* tab type */ (4)
    } SCRLAY_TABSTOP;
```

In these terms, the content of the style data record is:

TEXT sc[2]	two-letter short code
TEXT tag[16]	style tag name
UWORD sflags	style control flags (5)
SCRLAY_FONT f	paragraph base font (6)
UWORD inherit	inherited attributes (7)
SCRLAY_MARGINS marg	margin positions
SCRLAY_SPACING spc	paragraph vertical spacing
UWORD olevel	outliner level
UWORD ntabs	number of tabstops in following table
SCRLAY_TABSTOP tab[8]	up to 8 tabstops, in ascending position order

Notes

(1) Paragraph margins are relative to the page margins (the left and right edges of the body print region).

(2) The paragraph alignment may be one of:

0	left aligned
1	right aligned
2	centred
3	justified

(3) The spacing flags may be any combination of:

0x01	keep on same page as following paragraph
0x02	keep whole paragraph on one page
0x04	paragraph starts a new page

(4) The tab type may be any one of the following:

0	left tab
1	right tab
2	centred tab

(5) The style control flags may contain any combination of:

0x02	undeletable
0x04	default

There must be one default style and at least one undeletable style in every document (in all Psion documents they are the same - style BT). It does not make sense for the default style to be deletable.

(6) In addition to the typeface numbers listed earlier, a typeface number of -1 is used to signify that the typeface and font size are to be inherited from the default style. In such a case the font size is conventionally set to zero.

(7) The inherited attributes field may contain any combination of:

0x01	underline
0x02	bold
0x04	italic

For each bit that is set, the corresponding bit in the `style` field of the `SCRLAY_FONT` struct must be clear.

Inherited attributes are taken from the default style.

Emphasis data record (type 7, length 28)

The file may contain up to 16 such records, each of which defines a single emphasis.

The following description is in terms of the `SCRLAY_FONT` struct (described above). In these terms, the content of the emphasis data record is:

`TEXT sc[2]`	two-letter short code
`TEXT tag[16]`	emphasis tag name
`UWORD sflags`	emphasis control flags (1)
`SCRLAY_FONT f`	emphasis font (2)
`UWORD inherit`	inherited attributes (3)

Notes

(1) The emphasis control flags must contain the value `0x01`, together with any combination of:

0x02	undeletable
0x04	default

There must be one default emphasis and at least one undeletable emphasis in every document (in all Psion documents they are the same - emphasis NN). It does not make sense for the default emphasis to be deletable.

(2) In addition to the typeface numbers listed earlier, a typeface number of -1 is used to signify that the typeface and font size are to be inherited from the enclosing paragraph style (which may itself inherit from the default style). In such a case the font size is conventionally set to zero.

(3) The inherited attributes field may contain any sensible combination of:

0x01	underline
0x02	bold
0x04	italic
0x08	superscript
0x10	subscript

For each bit that is set, the corresponding bit in the `style` field of the `SCRLAY_FONT` struct must be clear.

Inherited attributes are taken from the enclosing paragraph style (which may itself inherit from the default style).

Document text record (type 8, variable length)

This record contains the entire text of the document. Each paragraph, except for the final one, is terminated by a zero.

The content conforms with the IBM Code Page 850 symbol set, together with the following additional symbols:

`HARD_HYPHEN 7`	unbreakable hyphen, not a word delimiter
`SOFT_HYPHEN 14`	optional, or potential, hyphen
`HARD_SPACE 15`	unbreakable space, not a word delimiter

Document index record (type 9, variable length)

This record contains a number of six byte entries which are used to apply style and emphasis to the document text. Each entry consists of:

- a length (word)
- the two character short code of a style
- the two character short code of an emphasis

The entries conform to the following rules:

- the sum of their lengths is one more than the length of the document text record (i.e. the document size, including the final paragraph terminator).
- there is at least one entry per paragraph
- the final entry for each paragraph includes the zero that terminates the paragraph
- the short codes must correspond with styles and emphases contained in the earlier style data and emphasis data records

CHAPTER 7

WRITING DEVICE DRIVERS

Introduction

This chapter is aimed at the programmer who wishes to write an installable device driver and anyone who wishes to improve their general device driver background. The details of communicating with device drivers can be found in the *I/O System* chapter of the *PLIB Reference* manual. Details of the resident device drivers can be found in the appropriate chapters of the *I/O Devices* manual and the *PLIB Reference* manual. The Borland Turbo assembler was used throughout. See also the following *Example Device Drivers* chapter.

Psion SIBO machines are supplied with a set of resident device drivers built in to the ROM each of which can be replaced with an installable device driver having the same name. Installable device drivers can also be added to increase the number of available device drivers. Installing a device driver is carried out dynamically without resetting the machine (this is not the case with many operating systems).

The device driver performs the logical processing required to translate low level hardware instructions into high level services suitable for an application. Conventionally, device drivers are divided into a logical layer riding astride a physical layer. The physical device driver (PDD) contains the code required for talking directly with the hardware device and provides a set of low level hardware specific services. The logical device driver (LDD) performs the logical processing that transforms these low level services into the high level services used by an application.

The following example illustrates the two layer nature of device drivers. An application using the serial driver decides that it requires RTS/CTS handshaking. It calls an LDD which decides whether or not a line should be driven. If the answer is yes the LDD calls the appropriate PDD and asks for a specific line to be driven to a specific state. The PDD duly carries out the requested service.

In the above example the LDD could have talked directly with the hardware. However Psion SIBO machines will often use the same LDD with a PDD written specifically for each version of the hardware device. Splitting the device driver is thus highly desirable.

An LDD must provide eight functions for use by the operating system. The functions are passed to the operating system via a table of function offsets (sometimes called the vector function table). These functions are mandatory. Similarly a PDD must provide two functions for use by the operating system and may provide a further two if required.

An LDD will usually provide further services/functions for use by an application. The form that these take is dependent on the LDD requirements and the functions supplied by the associated PDD(s). It is advisable to adopt the predefined system defines for these services as this allows the LDD to receive I/O requests via the usual route (`p_read`, `p_write` etc).

A PDD will usually define further services specifically for use by LDDs or (less frequently) applications.

The operating system will send device drivers system events not sent to other applications. Examples are events generated by the machine being switched on or off, memory segments being moved about and the owning application being panicked.

Any device driver configuration that has associated hardware interrupts must contain at least an LDD.

The EPOC operating system can handle a maximum of 32 device drivers on a Series3 machine and 48 on other machines.

The location of device drivers

Resident device drivers are built into the operating system, with the code residing in the ROM.

Installable device drivers are loaded into a device memory segment from the file in which they exist. The device memory segments are created, owned and managed by the SYS$FSRV process.

Under no circumstances should any application or device driver attempt to create, delete or change the size of a device memory segment.

Device Driver Names

Device drivers are known to the EPOC operating system by their names.

A logical device driver name always has three characters followed by a colon. For example:

- `TTY:` is the serial LDD
- `TIM:` is the timer LDD
- `SND:` is the sound LDD

A physical device driver name always has three characters followed by a period, a further three characters and a colon. For example:

- `TTY.UAR:` is the 16450 UART driver
- `TTY.AS5:` is the ASIC5 driver

The first three characters of a PDD name are the name of the LDD to which the PDD belongs. The second set of three characters uniquely identify the PDD. In the above examples both PDDs belong to the `TTY:` LDD.

The name of the device driver is the mechanism by which an application can obtain a 'channel' to the device driver.

Device Driver Channels

To obtain a channel to an LDD, an application should call the `IoOpen` operating system service. A channel can be opened by calling the PLIB library function `p_open`. For example:

- `p_open(&chan,"SND:",-1)`
- `p_open(&chan,"TIM:",-1)`

To obtain a channel to a PDD, an application should call the `DevOpenPDD` operating system service. Typically only LDDs open PDDs. The `p_open` library function can be used to open a PDD indirectly as described below.

For a device driver configuration consisting of an LDD and a PDD the application will usually open a channel to the LDD only: the LDD as part of its initialisation would open a channel to the required PDD. A channel can be opened by calling the PLIB library function `p_open`. For example:

- `p_open(&chan,"TTY.UAR:",-1)`
- `p_open(&chan,"TTY.AS5:",-1)`

If the LDD requires a PDD and none is specified, it is up to the LDD to either fail the open request or hunt for a loaded PDD that it can use. The `TTY:` device hunts for an appropriate PDD.

A device driver may be capable of supporting more than one open channel at a time. In order to distinguish the channels, a qualifier can be added to the open request as part of the device name. It is up to the device driver to specify the format of the qualifier. By convention channels are allocated a single character sequentially from the character 'A'. For example, the parallel driver can support two open channels, 'A' and 'B'. The LDD requires one of these qualifiers in order to open a parallel driver channel.

- `p_open(&chan,"PAR:A",-1)`
- `p_open(&chan,"PAR:B",-1)`

LDDs have been designed to be accessed via the I/O system. I/O requests on the opened channel will reach the 'strategy vector' of the device driver.

PDDs have been designed to be accessed by an LDD either via far calls or the DevVector operating system function.

Searching for PDDs

In the case that an LDD requires a PDD to provide some hardware specific functionality and the open LDD request does not specify a PDD, the LDD should search for a PDD to use. In this case either there is only one PDD for that machine (but is different across machines) or the PDDs are capable of determining whether it can drive the specified channel.

For example the serial LDD requires a PDD. However there is currently only one serial PDD on each machine (each machine has a different PDD though). By searching for the appropriate serial PDD, the serial LDD can be the same on all machines.

On the other hand there are several filing system PDDs each of which is capable of reading an ID byte from a SIBO pack. The PDD can then determine if it is the correct PDD for that hardware or not.

To search for a PDD, an LDD should use the DevFind service. For example the FSY: LDD would search for all FSY.* PDDs. As each PDD is found, it can be requested to open the appropriate channel by using the DevOpenPDD operating system service.

Device Driver Hierarchies And Attached Drivers

LDDs are classified as either root or attached drivers.

Device drivers exist in a hierarchy the first of which is termed the root driver. The other drivers in the hierarchy are termed attached drivers. In the language of object oriented programming, an attached driver subclasses the root driver. In this document, the driver to which another driver is attached is referred to as the underlying driver.

An attached device driver requires the underlying driver to provide a specified set of functions. How these are implemented is of no concern to an attached driver. For example, the Xmodem device driver is an attached driver which can, for example, attach to the serial driver which happens to be a root device driver. The Xmodem driver requires the underlying driver to support the serial sense, serial set, read, write and close functions. The power of attached drivers comes from the fact that the Xmodem driver does not need to know anything about the physical transmission medium, and can run on either serial port quite happily. In fact the Xmodem driver could run over any physical medium e.g. telephone, parallel, radio, infra-red etc as long as the underlying driver supported the small set of functions required.

Additional power comes from the fact that a driver does not have to be attached to the root driver directly: other attached drivers may exist in the hierarchy. For example an Xmodem driver can be attached to a modem driver that provides modem configuration and dialling functions. This can in turn be attached to the resident serial driver.

Notice that the Xmodem driver neither knows nor cares about the driver hierarchy.

There is no limit to the number of drivers that exist in a device driver hierarchy.

When the operating system routes an I/O request to a device driver, it follows this hierarchy and calls the strategy vector of the device driver at the top of the hierarchy. The device driver at the top of the hierarchy is the last opened device driver on that I/O channel.

The routing mechanism is best explained by an example of an application that wishes to use the parallel driver with a timeout facility. In its raw form, the parallel driver does not allow for timeouts. Although the application could handle this, a neater solution (in terms of application code) is to use an attached driver. The application opens the parallel driver in the normal way and then opens the attached driver (written as part of the application) passing the currently opened parallel device channel handle to the open vector. The attached driver will open a timer channel for itself and use the IoFuncAttach I/O service on the passed opened channel. This request will go to the parallel driver since it is the next down the hierarchy. The parallel driver, not supporting this function, passes it on to the operating system (using the IoRoot service) to perform the attach service. The parallel drivers open channel handle is returned by the attached driver to the caller of the IoOpen service. From this point on *all* I/O requests made on the parallel drivers I/O channel handle will be directed to the strategy vector of the attached driver first which can then process it and pass on any requests it feels necessary. For example, the IoFuncWrite request would go to the attached drivers strategy vector. It would queue a timer for an appropriate length of time and then pass on the IoFuncWrite request to the parallel driver. If the timer expired before the write completed, the attached driver would cancel the outstanding write request on the parallel driver and inform the application that the timer expired.

In the above example the same attached driver could in fact attach itself to any device driver that requires a timeout on the `IoFuncWrite` request.

An attached driver is written in exactly the same way as any other driver. However, if the driver does not support a requested function, the strategy vector of an attached driver calls the `IoSuper` operating system service rather than the `IoRoot` system service.

Interrupts and Interrupt Service Routines

Device drivers that talk to hardware tend to have interrupt service routines associated with them, especially if they are receiving data from an external source.

The EPOC operating system provides a framework within which an interrupt service routine can be written relatively easily.

The SIBO architecture allows for eight independent hardware interrupt sources, some of which are pre-allocated to system components (see the ASIC1 section of the *Hardware Reference* manual for details).

The operating system provides the `GenSetRevector` service to allow a device driver to install an interrupt service routine for any of the eight hardware interrupt sources.

A device driver should use this system service and not poke directly into the 8086 interrupt vector table. The address passed to the `GenSetRevector` service is not written into the interrupt vector table but to an internal table.

When an interrupt occurs, the operating system builds the mandatory operating system call frame, preserving *all* registers on route. The interrupt service routine is then called as a *FAR* routine. Since the operating system preserves all registers the interrupt service routine is free to use any register.

To remove the interrupt service routine address, the operating system service `GenResetRevector` should be used. This will reset the internal table entry to the default held in the ROM.

As with all interrupt service routines various rules apply:

- Interrupt service routines should execute as fast as possible. Operating system interrupt service routines are tuned to last no longer than one millisecond.
- Typically, interrupt service routines do not enable interrupts unless the routine can handle reentrancy.
- Interrupt service routines run in the context of whatever process is running at the time of the interrupt. An interrupt service routine should not attempt to obtain admissibility to the process that opened the channel but access the internal driver space only which in general is its own code space.
- An interrupt service routine must not directly cause a reschedule as this would significantly delay its completion. It must use the `IoSignalByPidNoReSched` system service in order to indicate that an event has occurred to the owning process. The handler function of the device driver must pick up the event and inform the owning process.
- An interrupt service routine should return with the carry flag clear if it requires a reschedule to occur (it has called `IoSignalByPidNoReSched`) otherwise return with the carry flag set. This will cause the operating system to reschedule if the internal state allows such an action otherwise the reschedule request is effectively queued until such time that the operating system can reschedule.

Device Driver I/O Semaphore Waithandlers

An LDD may nominate one of its functions to be called by the operating system every time the I/O semaphore of the process that opened the channel is signalled. The nominated function will only be called if the application is waiting for an outstanding I/O request to complete. For well written applications this is practically all the time.

By convention the vector table entry after the mandatory vectors contains the handler vector.

A handler routine is similar to an interrupt service routine in that it appears to run 'from nowhere'.

Comparing handlers and interrupt services routines shows that:

- A handler will always run in the context of the process that has opened a channel. An interrupt service routine will run in the context of whatever process happens to be running at the time of the interrupt.
- A handler can access the data space of the process that opened the channel. The interrupt service routine must not. An interrupt service routine should only access the data space in the driver which is usually its own CS space.
- A handler can cause a reschedule. An interrupt service routine must not cause a reschedule. If it did, the interrupt would not be fully serviced (the rest of the interrupt service routine would not be executed until a reschedule back to the process running at the time of the interrupt, which may not happen for a significant length of time). The interrupt service routine must only use the `IoSignalByPidNoReSched` to signal the channel owner.

The handler is the mechanism by which hardware interrupt events can be filtered through to the process using the I/O channel.

Loadable Logical Device Driver Structure

A loadable LDD must obey the following rules:

- The must be a single code segment and no data segments.
- The code segment must start with a `LibEnt` structure.
- There must be at least eight supported functions.

Single Code Segment

An LDD must be written to contain any internal variables within its own code segment. In general these variables are only concerned with unit allocation and the hardware state.

Data space for a particular open channel can be allocated in the heap space of the process that opens the device. This data space will however disappear if the process terminates. Therefore any variables required for 'freeing' the hardware after a process terminates must exist in the code space of the device driver.

The LibEnt Structure

A `LibEnt` structure has the following format:

- two byte signature
- eight byte name
- two byte vector count
- A vector table

The two byte signature should contain the `'LDDSignature'` define.

The eight byte name contains the name of the device driver stored as a zero terminated string. Note that the trailing colon is omitted.

The two byte vector count contains the number of vectors that follow immediately after the count. This should be equal to at least eight.

For example:

```
        dw      LDDSignature                    ; Its an LDD
        db      'DVR',0,0,0,0,0                 ; Name of the driver
        dw      (VectorEnd-Vector)/2            ; Number of vectors
Vector:
        dw      DvrInstall                      ; Install vector
        dw      DvrRemove                       ; Remove vector
        dw      DvrHold                         ; Hold vector
        dw      DvrResume                       ; Resume vector
        dw      DvrReset                        ; Reset Vector
        dw      DvrUnits                        ; Units Vector
        dw      DvrOpen                         ; Open Vector
        dw      DvrStrategy                     ; Strategy vector
VectorEnd:
```

The vector table contains the offsets within the device drivers code segment for the functions required by the EPOC operating system. Throughout this document the terms vector and function are used interchangeably. The vector table must have the entries in the order shown in the example.

Mandatory LDD Functions

All LDDs must support the following eight functions:

- `DevFuncInstall` called on device installation.
- `DevFuncRemove` called on device removal.
- `DevFuncHold` called to temporarily disable the driver.
- `DevFuncResume` called to enable the driver after it has been temporarily disabled.
- `DevFuncReset` called when an application terminates without closing the channel.
- `DevFuncUnits` called to query the number of supported units (i.e. channels).
- `DevFuncOpen` called to open a channel to an LDD.
- `DevFuncStrategy` called to access the device drivers functionality from the I/O system.

All of the routines pointed at by the function vector table will be called *FAR* by the operating system and should consequently use a *FAR* return machine code instruction to return back to the operating system.

Since the *FAR* return address is to the operating system it does not matter if the operating system moves memory whilst code in the LDD is being executed: the operating system cannot move its own code.

DevFuncInstall

This function is called by the operating system when the device driver is loaded in order to initialise any internal variables. It can not be called directly by an application process.

The `DevInstall` operating system service will cause this function to be called. Applications should not call this service directly and should call instead the `DevLoadLDD` service.

An installable device driver may have the same name as a resident device driver and when loaded is placed at the end of the device driver table. When the operating system wishes to establish a channel with a device driver, it searches for the device driver starting at the end of the table. It will thus find the most recently loaded device driver having the required name. By this mechanism an installable device driver can replace a resident device driver of the same name.

An installable device driver may have the same name as a resident device driver. When the operating system loads a device driver, it places it at the end of the device driver table. The operating system will search this table for the appropriate device driver when it wishes to establish a channel. The search starts at the end and thus will locate the most recently installed device driver (if any) or if not, the resident driver. By this mechanism an installable driver can replace any resident driver.

When called, the DS and ES segment registers are in an unknown state. The device driver should take whatever steps necessary to obtain direct addressability to its data. For loadable device drivers this involves setting the DS and ES registers to the CS register.

The operating system will not move memory whilst in this function, thus the normal rules governing DS and ES may be ignored.

All operating system services may be called, except those concerning file or device access.

PASSED

No values are passed to the install vector.

RETURN

If the installation was successful, return with the carry flag clear.

If the installation failed, return with the carry flag set and the error number in the AL register.

PANIC

The install vector must not panic: it will cause an operating system kernel fault if it does.

PRESERVE

The SS, SP and BP registers must be preserved by the install function.

DevFuncRemove

This function will be called by the operating system when the device driver is requested to be unloaded. It can not be called directly by an application process.

The `DevRemove` operating system service will cause this function to be called. Applications should not call this directly, they should use the `DevDelete` service.

Before the remove function is requested, the device driver will have received a hold request. Thus devices will only ever be removed when in a held state.

If the device driver is currently busy serving a client, the remove request should return an error.

All resident device drivers will return an error since there is no mechanism by which they can be re-installed.

When called, the DS and ES segment registers are in an unknown state; the device driver should take whatever steps necessary to obtain direct addressability to its data. For loadable device drivers this involves setting the DS and ES registers to the CS register.

The operating system will not move memory whilst in this function, thus the normal rules governing DS and ES may be ignored.

All operating system services may be called, except those concerning file or device access.

PASSED

No values are passed to the remove vector.

RETURN

If the remove was successful, return with the carry flag clear.

If the remove failed, return with the carry flag set and the error number in the AL register.

PANIC

The remove vector must not panic: it will cause an operating system kernel fault if it does.

PRESERVE

The SS, SP and BP registers must be preserved by the remove vector.

DevFuncHold

This vector will be called by the operating system when the device driver is requested to be held. The hold vector is called in the context of the operating system.

The `DevHold` operating system service will cause this vector to be called. Applications should not call this service.

The operating system will call the hold vector under three conditions:

- Device memory segments are about to be moved.

- The machine is about to switch off due to the auto switch off timeout or user request, it enters the standby state.
- The machine is about to switch off due to the power source being removed.

In all cases the device driver must respond to the request as quickly as possible. It must also ensure that ALL interrupts from the hardware device that it is driving are disabled.

Device memory segments can only be moved if an installable device driver is being installed or removed. If the LDD uses an attached PDD and uses the faster *FAR* call mechanism to call the PDD strategy vector, the PDD strategy vector address will potentially move, thus the *FAR* address will be wrong. This address can be resolved in the resume vector. The LDD must not call the PDD between a hold and resume. Typically, the device driver only needs to disable its interrupts. When a resume occurs, the device driver should continue as though nothing had happened.

If the machine is about to switch off due to the auto switch off or user request mechanisms (enter the standby state), the device driver should make an orderly shut down of the device such that the state before the shut down can be recovered when the system powers up again. The device driver should also attempt to ensure that no data is lost. For example, in the serial driver the current state of the hardware handshaking lines should be noted so that each state can be restored on power up. For this type of power down the hold vector is allowed to take a significant length of time to shut down a device. For example in a serial driver the hold vector should wait until the remote end stops transmitting data after any hardware handshaking has been applied. Of course, the time taken should be kept to a minimum: in the case of the serial driver above the time is roughly equivalent to 3 character transmission times. When a resume occurs the device driver should continue as though nothing had happened.

If the machine is about to switch off due to the power source being removed, the device driver should reset the device in the minimum possible time: no attempt should be made to perform an orderly shut-down. The device driver is not expected to be able to recover the hardware state. When a resume occurs, the device driver would typically fail any outstanding application requests. If the hold vector takes too long the voltage will fall below the threshold to hold the state of the internal RAM. If this occurs the machine will perform a warm re-boot when powering up, all data in the internal memory of the machine will be lost including the device driver code! On power fail there is about 2ms available to power down all devices.

On a power failure hold, the operating system will already have sent a 'reset' to all the SIBO serial channels. Any device drivers using these channels need only record the hold reason for the resume vector. Any other peripherals should be designed to allow a power fail mechanism with the minimum amount of code.

It *must* be noted that the power fail type hold can occur whilst the device driver is in the memory move hold state. In this case, the device driver will receive two hold requests before seeing a resume request. A device driver must be capable of handling this. In this case, the device driver will also receive two resume requests. A device driver will not get a power fail hold whilst in power down hold.

A call to the hold vector will always be followed by a call to the resume vector (except when a device is requested to be removed).

When called, the DS and ES segment registers are in an unknown state; the device driver should take whatever steps necessary to obtain direct addressability to its data. For loadable device drivers this involves setting the DS and ES registers to the CS register.

The device driver should not call any operating system services due to the time taken, especially on power failure.

PASSED

The AH register takes one of the following

- `DevHoldNormal` Device memory is about to be moved.
- `DevHoldPowerDown` The system is about to enter the standby state.
- `DevHoldPowerFail` The system has lost its power supply.

RETURN

None.

PANIC

The hold vector must not panic: it will cause an operating system kernel fault if it does.

PRESERVE

The SS, SP and BP registers must be preserved by the hold vector.

DevFuncResume

This vector will be called by the operating system when the device driver is requested to be resumed. The resume vector is called in the context of the operating system.

The `DevResume` operating system service will cause this vector to be called. Applications should not call this service.

The resume vector will be called either when memory has finished being moved or when the machine powers back up. In both cases the hold vector will have been called before this vector is called.

The device driver is expected to recover from the previous hold request (except power fail) and resume any I/O that was suspended.

If the device driver has an interrupt service routine, it should reset the interrupt service routine's address since the device driver may have moved in memory; its absolute segment address will be different.

If the hold was a device memory segment move type hold, interrupts should be re-enabled. If the LDD uses an attached PDD and uses the *FAR* call mechanism to access the PDD strategy vector, the address of the PDD should be reset by using the `DevGetPDDAddress` operating system service before enabling interrupts. Typically, the PDD will have a call back to the LDD and it needs to be informed of the change of address of the LDD call back function; the LDD-PDD interface definition should allow such a function request.

If the hold was a power down type hold, the resume vector needs to power up the peripheral and set it to the state that it was in before the power down occurred. If this is not possible or data has been lost, the device driver should inform any outstanding requests of this fact.

It is also possible that the hardware device that the driver is associated with has been removed. The driver should be able to handle this properly.

If the device driver is expected to generate events due to an external state change, the driver should check the external state and generate appropriate events. For example, the serial driver may be requested to inform an application when the DTR line changes state. The remote end may have changed the state of DTR whilst the driver is held.

If the hold was a power failure type hold, the resume vector should power up the peripheral and put it into a known state, preferably the state that the application software thinks that the device is in and fail any outstanding requests as data is quite likely to have been lost.

When called, the DS and ES segment registers are in an unknown state. The device driver should take whatever steps necessary to obtain direct addressability to its data. For loadable device drivers this involves setting the DS and ES registers to the CS register.

All operating system services may be called, except those concerning file or device access.

PASSED

None

RETURN

None.

PANIC

The resume vector must not panic, it will cause an operating system kernel fault if it does.

PRESERVE

The SS, SP and BP registers must be preserved by the resume vector.

DevFuncReset

This function will be called by the operating system when the device driver is requested to reset a channel. The reset function is called in the context of the operating system.

The device driver must request that the operating system call the reset function. This is achieved by calling the `IoRequestReset` system service, usually in the open vector. To cancel this request, the device driver should call the `IoRequestResetCancel` system service. The cancel service is usually called as part of the close functionality in the strategy vector.

The reset vector will be called when the operating system is tidying up resources owned by a process that has terminated. If a process terminated before it closed the device driver channel and no reset service is requested, that channel would remain allocated; no process will ever close the channel. The reset vector allows a device driver to reset itself and allow the channel to be opened again.

Any data required to perform the reset must be stored in the device driver. The data space belonging to the process that originally opened the channel has been returned to the operating system memory pool and is no longer valid.

If a device driver can handle multiple channels then the data passed to the `IoRequestReset` system service should identify the channel. This data will be passed in the CX register to the reset vector.

The device driver should only have a reset request outstanding with the operating system while a process has a channel open.

When called, the DS and ES segment registers are in an unknown state; the device driver should take whatever steps necessary to obtain direct addressability to its data. For loadable device drivers this involves setting the DS and ES registers to the CS register.

All operating system services may be called, except those concerning file or device access.

PASSED

This function is passed data in the CX register that the device driver requested it be sent to determine which channel should be reset.

RETURN

None.

PANIC

The reset vector must not panic; it will cause an operating system kernel fault if it does.

PRESERVE

The SS, SP and BP registers must be preserved by the reset vector.

DevFuncUnits

This function will be called by the operating system when the device driver is requested to report the number of units (i.e. channels) the device driver can support. This function is called in the context of the operating system.

The operating system places no significance on the number of channels a device driver can support. It is primarily used for informational purposes.

An application may use the number of units to attempt to open any available channel on that device driver.

When called, the DS and ES segment registers are in an unknown state; the device driver should take whatever steps necessary to obtain direct addressability to its data. For loadable device drivers this involves setting the DS and ES registers to the CS register.

All operating system services may be called, except those concerning file or device access.

PASSED

None.

RETURN

The AX register should contain the number of channels supported. If a device driver can support multiple channels (limited only by memory constraints) then the driver may return -1. A serial device driver, for example, might only support two channels (`TTY:A` and `TTY:B`) whereas the file device driver can open an unlimited number of files.

PANIC

The channels units vector must not panic; it will cause an operating system kernel fault if it does.

PRESERVE

The SS, SP and BP registers must be preserved by the units vector.

DevFuncOpen

This function will be called by the operating system when a channel to the device driver is required to be opened. This function is called in the context of the process that called the `IoOpen` system service.

The device driver is passed two parameters, its device handle and a pointer to an `OpenEnt` structure.

The device handle is the entry in the system device table of this device driver. The device driver is required to place this handle in the `ChanLibHandle` field of the `ChanEnt` structure which must be allocated in the user's data space. The operating system uses the device handle to route any I/O requests on the opened channel to the correct device driver.

The `OpenEnt` structure contains three fields, `OpenNamePtr`, `OpenMode` and `OpenChan`.

The `OpenNamePtr` field contains a pointer to the character that exists after the device name as passed to the `IoOpen` system service. For example, if the `IoOpen` service was passed a name of `PAR:A`, the `OpenNamePtr` field would point to the colon. If the `IoOpen` service was passed a name of `TTY.AS5:B` the `OpenNamePtr` field would point to the full stop. The device driver should process the name appropriately, opening the correct PDD as required.

The `OpenMode` field contains the mode for opening the device driver. The available modes are specified by the device driver writers. For example, a combined Xmodem and Ymodem device driver could use the mode to specify whether the Xmodem or the Ymodem protocol is to be used.

The `OpenChan` field contains the I/O channel handle of the device that this driver is required to 'attach' to. Attached device drivers are dealt with later in the chapter.

The code in a device driver open vector tends to follow a very similar pattern. This is demonstrated by the following code fragments and associated comments.

The first stage is to allocate some data space in the calling process' heap space. This will contain the I/O channel control block:

```
mov cx, (size DeviceEnt)
HeapAllocateCell
jc  noMemory
mov bx, ax                  ; cell handle
```

If the device driver requires a WaitHandler (described later):

```
mov al, (VectorHandler-Vector)/2
IoAddHandler
jc  endFreeMemory
mov [bx].DriverHandler, ax
```

If the device driver's `DevFuncReset` vector is required to be called:

```
pushbx
mov cx, ChannelIndicator    ; unique per channel
mov bx, dx                  ; the device handle
IoRequestReset
pop bx                      ; restore alloc cell
```

The `ChanEnt` field of the `DriverEnt` structure must be initialised:

```
mov [bx].DriverIo.ChanNext, bx
mov [bx].DriverIo.ChanSignature, IoChanSignature
mov [bx].DriverIo.ChanLibHandle, dx
```

The `ChanNext` field is used by attached drivers and will usually be set to be the allocated cell handle of the device driver being opened. The `IoFuncAttach` and `IoFuncDetach` functions manipulate these fields. The I/O system uses this field to direct the I/O request to the correct driver.

The ChanSignature field is checked by the operating system during any I/O requests for the value IoChanSignature. If it does not contain that value, the process calling the I/O service will be panicked for having passed an invalid I/O channel handle.

The ChanLibHandle field is used by the operating system to route an application's I/O request to this device. The I/O request will call the DevFuncStrategy vector of the device driver.

If the driver is an attached driver the following is required:

```
mov cx, bx                  ; allocated channel
mov bx, [si].OpenChan       ; channel attaching to
mov al, IoFuncAttach        ; return in BX the
IoWithWait                  ; channel attached to
```

Finally, if the channel has been successfully opened:

```
clc                         ; Opened Ok
ret                         ; return BX and DX
```

The error recover code typically follows the following pattern:

```
endFreeReset:
    push    ax
    push    bx
    mov     cx, ChannelIndicator
    mov     bx, dx
    IoRequestResetCancel
    pop     bx
    pop     ax
endFreeHandler:
    push    ax
    push    bx
    mov     bx, [bx].DriverHandler
    IoRemoveHandler
    pop     bx
    pop     ax
endFreeMemory:
    push    ax
    HeapFreeCell
    pop     ax
    stc
noMemory:
    ret
```

If a device driver supports a fixed number of channels, it typically contains static control blocks. In order to determine if a requested channel is currently open, a field should be interrogated. The device driver should ensure that interrupts are disabled during this sort of check since a context switch could occur and another process request the opening of the same channel. This is the classic 'test and set' problem encountered in multi-tasking environments.

When called, the DS and ES segment registers point to the data segment of the application process attempting to open a device channel. The application should ensure that the DS and ES segment registers do in fact point to its data segment. The device driver must obey the normal rules concerning segment register manipulation. The DS and ES segment registers can be reloaded if required from the IntEnt structure pointed at by the BP register.

All operating system services may be called.

PASSED

DX contains the device handle of the device driver.

SI is a pointer to the OpenEnt structure

BP is a pointer to the IntEnt structure.

RETURN

If the channel open was successful, return with the carry flag clear and the BX register containing the open channel.

If the open failed, return with the carry flag set and the error number in the AL register.

PANIC

The open vector can panic; it will cause the process requesting the device open to terminate. It is however more usual to return an error to the calling process.

PRESERVE

The DS, ES, SS, SP, BP and DX registers must be preserved by the open vector.

DevFuncStrategy

When an application makes an I/O request on the opened device driver channel the request is routed to this vector by the operating system. A device driver defines the set of functions that it supports. These typically include `IoFuncSet`, `IoFuncSense`, `IoFuncRead`, `IoFuncWrite` and `IoFuncClose`. A device driver does not have to support any particular function, as it is a matter of design between a device driver writer and application writer as to what functions and associated parameters are provided.

To obtain the power of attached device drivers, it is recommended that the device driver use the system defines with their appropriate functionality, for example, the `IoFuncWrite` function number should always be associated with writing data.

The strategy function is passed the channel handle as allocated in the open vector in the BX register. This typically contains control information concerning the current state of the I/O channel.

The SI register contains a pointer to a `RqEnt` structure. This structure contains four fields, `RqFunction`, `RqStatusPtr`, `RqA1Ptr` and `RqA2Ptr`.

The `RqFunction` field contains the function number as passed to the `IoWithWait` (or `IoAsynchronous`) I/O request by the application. If a device driver does not support the specified function, it should pass the request on to its 'parent' device driver.

The `RqStatus` pointer contains a pointer to a memory location in the application process's data space that receives the I/O requests completion status. The device driver must set this memory location to the value `PendingErr` whilst the I/O request is outstanding and a completion code when the I/O request completes. An I/O request may complete within the strategy vector or it may complete some time in the future, presumably from some interrupt.

The `RqA1Ptr` and `RqA2Ptr` fields contain the argument 1 and 2 parameters as passed to the `IoWithWait` (or `IoAsynchronous`) system services. The device driver is free to specify what these parameters are (if any).

The operating system defines a set of common function numbers used by device drivers referred to as the `IoFuncXXX` set of defines. By convention, a device driver should select from this list, particularly if some of the more advanced features of the I/O system are to be used, such as attached device drivers. The more common defines are:

- `IoFuncRead` ; read from the device.
- `IoFuncWrite` ; write to the device.
- `IoFuncClose` ; close device channel.
- `IoFuncCancel` ; cancel an I/O request.
- `IoFuncSet` ; set driver characteristics
- `IoFuncSense` ; sense driver characteristics.
- `IoFuncFlush` ; flush any buffers.

The PLIB library functions `p_read`, `p_write` and `p_close` will call the device driver with the `IoFuncRead`, `IoFuncWrite` and `IoFuncClose` function numbers. Thus, if the device driver choses an alternative function number set, an application will not be able to use the supplied library functions.

All resident device drivers obey the following conventions:

- A cancel request will cancel any outstanding requests. A cancel request will not return any error.
- A close request will ensure that any outstanding requests are completed before closing the channel. A close request will not return any error.
- Only one request of a particular type can be outstanding at any one time. If a second request is made the device driver will panic the calling application.

Any functions that the strategy function does not support should be passed on to the next driver down the driver hierarchy. If the driver is a root driver (attached driver), this is achieved using the `IoRoot` (`IoSuper`)system service. If the requested function is not supported by any driver, the operating system will return a `NotSupported` error.

When called, the DS and ES segment registers point to the data segment of the application process making the I/O function request. The application should ensure that the DS and ES segment registers do in fact point to its data segment. The device driver must obey the normal rules concerning segment register manipulation. The DS and ES segment registers can be reloaded if required from the `IntEnt` structure pointed at by the BP register.

All operating system services may be called.

PASSED

BX contains the allocated channel control block.

DX contains the device handle of the device driver.

SI is a pointer to the `RqEnt` structure.

BP is a pointer to the `IntEnt` structure.

RETURN

If the function request is successful, the strategy vector should return with carry clear. A request typically causes some I/O. If the I/O is completed by the strategy vector (eg the close function), the completion status should be written back to the `RqStatusPtr` location and the I/O semaphore signalled (using the `IoSignal` system service). If the request has not yet completed, the `RqStatusPtr` location should contain the value `PendingErr` and the I/O semaphore should *not* be signalled.

If the function request failed the strategy vector should return with carry set and the error code in AL. Typically no I/O requests will be completed.

PANIC

The strategy vector can panic; it will cause the process making the I/O request to terminate. In most cases it is usual to return an error to the calling process. A major exception to this is if the calling process makes an I/O request of the same type as one that is currently outstanding and the device driver only supports one I/O request of a particular type at a time; by convention the device driver should panic the calling process with the `PanicIoPending` panic code.

PRESERVE

The DS, ES, SS, SP and BP registers must be preserved by the strategy vector.

Loadable Physical Device Driver Structure

A loadable PDD must obey the following rules:

- There must be a single code segment and no data segments.
- The code segment must start with a `LibEnt` structure.
- There must be at least two supported functions, with typically a further two defined.

Single Code Segment

A PDD must be written to contain any internal variables within its own code segment. Typically, these variables are only concerned with unit (i.e. channel) allocation and hardware state.

Data space for a particular open channel can be allocated in the heap space of the process that opens the device. This data space will however disappear if the process terminates, thus any variables required for 'freeing' the hardware after a process terminates must exist in the code space of the device driver.

The LibEnt Structure

A `LibEnt` structure has the following format:

- A two byte signature
- An eight byte name
- A two byte vector count
- A vector table

The two byte signature should contain the 'PDDSignature' define.

The eight byte name contains a zero terminated name, being that of the device driver. Note that there is no trailing colon.

The two byte vector count contains the number of vectors that follow immediately after the count. There should be at least two.

For example:

```
    dw      PDDSignature            ; Its an PDD driver
    db      'DVR.HW1',0             ; Name of the driver
    dw      (VectorEnd-Vector)/2    ; Number of vectors
Vector:
    dw      DvrInstall              ; Install vector
    dw      DvrRemove               ; Remove vector
VectorEnd:
```

Most PDDs also define a further two vectors:

```
    dw      DvrOpen                 ; Open Vector
    dw      DvrStrategy             ; Strategy vector
```

The table of vectors is a table of offsets within the device drivers code segment of the routines that implement the required functionality. The vector table must have the entries in the order shown in the example.

Mandatory PDD functions

All PDDs must support the following two functions:

- `DevFuncInstallPDD` called on device installation.
- `DevFuncRemovePDD` called on device removal.

Most PDDs will support the following two additional functions:

- `DevFuncOpenPDD` called to open a PDD
- `DevFuncStrategyPDD` called to provide PDD functionality

All of the routines pointed at by the function vector table will be called *FAR* by the operating system and should consequently use a *FAR* return machine code instruction to return back to the operating system.

Since the *FAR* return address is to the operating system, it does not matter if the operating system moves memory whilst code in the LDD is being executed; the operating system cannot move.

DevFuncInstallPDD

This vector will be called by the operating system when the device driver is loaded to initialise any of its internal variables. The install vector is called in the context of the operating system and not the process that is loading the device driver.

The `DevInstall` operating system service will cause this vector to be called. Applications should not call this directly, they should use the `DevLoadPDD` service.

An installable device driver may have the same name as a currently installed device driver. When installed, the driver is added to the end of the device driver table. When a channel to a device driver is being established by the operating system, it searches the device table from the end first, thus the latest installed device driver with the required name will be asked first for a channel. By this mechanism, installable device drivers can replace any of the resident drivers.

When called, the DS and ES segment registers are in an unknown state. The device driver should take whatever steps necessary to obtain direct addressability to its data. For loadable device drivers this involves setting the DS and ES registers to the CS register.

The operating system will not move memory whilst in this function, thus the normal rules governing DS and ES may be ignored.

All operating system services may be called except those concerning file or device access.

PASSED

No values are passed to the install vector.

RETURN

If the installation was successful, return with the carry flag clear.

If the installation failed, return with the carry flag set and the error number in the AL register.

PANIC

The install vector must not panic; it will cause an operating system kernel fault if it does.

PRESERVE

The SS, SP and BP registers must be preserved by the install vector.

DevFuncRemovePDD

This vector will be called by the operating system when the device driver is requested to be unloaded. The remove vector is called in the context of the operating system and not the process that requests the unload.

The `DevRemove` operating system service will cause this vector to be called. Applications should not call this service directly; instead, they should call the `DevDelete` service.

Before the remove function is requested, the operating system will send a `DevFuncHold` request to all LDDs. The LDD is responsible for ensuring that no activity will occur during the remove. Note that any device driver that handles hardware interrupts must contain an LDD since only LDDs receive a hold request.

If the device driver is currently busy serving a client, the remove request should return an error.

All resident device drivers will return an error since there is no mechanism by which they can be re-installed.

When called, the DS and ES segment registers are in an unknown state; the device driver should take whatever steps necessary to obtain direct addressability to its data. For loadable device drivers this involves setting the DS and ES registers to the CS register.

The operating system will not move memory whilst in this function, thus the normal rules governing DS and ES may be ignored.

All operating system services may be called except those concerning file or device access.

PASSED

No values are passed to the remove vector.

RETURN

If the remove was successful, return with the carry flag clear.

If the remove failed, return with the carry flag set and the error number in the AL register.

PANIC

The remove vector must not panic; it will cause an operating system kernel fault if it does.

PRESERVE

The SS, SP and BP registers must be preserved by the remove vector.

DevFuncOpenPDD

This function is defined as a convenience function for the LDD-PDD interface.

When an application opens a channel to an LDD, it normally uses the `IoOpen` system service. If the name specifies, or the LDD requires, a PDD then it needs to open a channel to a PDD. The `DevOpenPDD` system service will call this PDD vector to establish a channel. The LDD now has a choice of calling a PDD vector using the `DevVector` system service or calling the fourth vector in the vector table directly. The fourth vector is assumed to be a strategy vector to which any parameters as required by the LDD-PDD interface can be passed. The *FAR* address of the strategy vector is returned by the `DevGetPDDAddress`. When an LDD receives a `DevFuncResume` it should call `DevGetPDDAddress` again to ensure that if the PDD has moved the LDD still has its correct address.

As a design, a PDD could provide many vectors, one for each required function. The LDD would then use the `DevVector` system service to access each of these functions. The `DevGetPDDAddress` will only return the *FAR* address of the fourth vector.

When called, the DS and ES segment registers point to the data segment of the application process making the open function request. The application should ensure that this is indeed the case. The device driver must obey the normal rules concerning segment register manipulation.

All operating system services may be called.

PASSED

The BX register contains a pointer to the PDD unit name. The pointer passed to the `DevOpenPDD` service is used to find the PDD device to open. The BX register is loaded with a pointer to the trailing colon (if any) in the PDD unit name. For example if the name `TTY.AS5:A` was passed to the `DevOpenPDD` service, BX would contain a pointer to `:A` upon calling the open vector.

RETURN

If the open was successful, return with the carry flag clear.

If the open failed, return with the carry flag set and the error number in the AL register.

PANIC

The open vector can panic; it will cause the process requesting the device open to terminate. It is however more usual to return an error to the calling process.

PRESERVE

The SS, SP and BP registers must be preserved by the open vector.

DevFuncStrategyPDD

This function is defined as a convenience function for the LDD-PDD interface.

Typically, all application function requests are routed through the strategy vector. To speed the calling interface, the `DevGetPDDAddress` operating system function will return a *FAR* address of this vector.

The device driver writer defines all the functions and return values as required.

When called, the DS and ES segment registers point to the data segment of the application process making the function request. The application should ensure that the DS and ES segment registers do in fact point to its data segment. The device driver must obey the normal rules concerning segment register manipulation.

All operating system services may be called.

PASSED

The parameters passed are defined by the device driver write.

RETURN

All returns are defined by the device driver writer.

PANIC

The strategy vector can panic; it will cause the process requesting the function to terminate. It is however more usual to return an error to the calling process.

PRESERVE

Which registers are preserved is defined by the device driver writer.

CHAPTER 8

EXAMPLE DEVICE DRIVERS

This chapter contains explanatory notes for the example device drivers supplied with the *Psion C SDK*. The source code for these examples can be found in *\sibosdk\ldd*. The Borland Turbo assembler is used throughout.

See also the preceding *Writing Device Drivers* chapter and the appropriate chapters in the *I/O Devices Reference* manual.

An Attached Device Driver Example

The code in *atimdvr.asm* contains an example of an attached device driver. Attached drivers add functionality to, or replace, a service provided by an underlying device driver. The example is a generic timeout device driver that adds a timeout facility to the underlying device driver's `P_FREAD` requests. The example driver may be attached to either the serial or the parallel drivers.

The example code in *t_atim.c* shows how the device driver is loaded and attached to a serial driver. It also shows how the additional functionality becomes transparent to the application once the driver has been attached.

The device table

The start of the file consists of the device driver header. The name of the driver is specified to be ATM. The driver is an LDD type driver consisting of nine callable functions, the first eight of which are mandatory. The ninth function (a wait handler) is required by the device driver to function correctly.

Note that the following notes apply specifically to the functions as used in the example driver.

The Install Function

The install function does not have to perform any actions apart from report that it has completed successfully.

The Remove Function

The remove function does not have to perform any actions apart from reporting that it has completed successfully. This makes the reasonable assumption that the user of the device driver will not attempt to remove it if it is associated with any open channels.

The Hold Function

The hold function is not required to do anything since it does not access or use hardware directly.

The Resume Function

Since the hold function does not do anything that needs to be undone, the resume function is not required to do anything (otherwise it might have performed tasks such as stopping interrupts, shutting down hardware etc).

The Reset Function

The reset function is not required to do anything: it can never be called since the driver does not ask the operating system to call this function if its client terminates without closing an open channel. The timer

device driver and the driver attached to are responsible for releasing system resources if a client terminates. Thus this driver can leave it up to those drivers to tidy up.

The Units Function

The driver can support an unlimited number of open requests subject to system memory and timer channel availability.

The Open Function

The open function is called by the operating system when an application uses the `IoOpen` system service to open the `ATM:` device. This is usually done via the PLIB library function `p_open`.

The open function runs in the context of the process that makes the open request. Thus resource allocation (e.g. memory allocation requests) will be associated with that process.

The open function allocates enough memory to hold all of the required internal variables and initialises the memory to zero. It then makes function nine in the device table a wait handler function by using the `IoAddHandler` system service. This installs the function in the linked list of wait handler functions. The wait handler by default is not callable, and should, for performance reasons, only be made callable when it has some processing to do.

A channel to a timer device is opened, the name `TIM:` is generated on the run time stack.

The I/O system channel header is set up, the I/O system requires that a device driver has a `ChanEnt` structure as the first item in the allocated cell.

Finally, if all has gone well, our driver attaches itself to the underlying driver whose already open handle was passed to the open function.

If any errors occur, the device driver is responsible for tidying up all of the currently allocated resources.

It may be noted that any I/O requests made by the device driver on its own channel (as allocated) after the attach request has been made will be routed to the underlying driver rather than starting from the top of the driver chain. Thus for example in the `CancelRead` procedure the `IoFuncCancel` request will be routed to the strategy function of the 'attached to' driver and not to the strategy function of our attached driver.

The Strategy Function

Once opened, the strategy function will be called when an application makes an I/O request. All I/O requests on the open channel will be routed to our example driver which must decide how they are to be handled. Some I/O requests are not recognised by the example driver and should be passed to the underlying driver. This may be done using the `IoSuper` system service (note that a root device driver would make an `IoRoot` system service request to pass on any meaningless I/O requests).

Some requests may be redefined; the example driver redefines the meaning of the `IoFuncSet` and `IoFuncSense` (`P_FSET` and `P_FSENSE`) services to allow an application to set and sense the timeout values to use. The application that attaches the example driver to the serial driver must be careful when using these services, since they produce different results depending on whether the time out driver has been attached or not (in one case they set and sense the serial characteristics and in the other they set and sense a time out value).

Some requests may be modified to enhance them; this driver enhances the `IoFuncRead` (`P_FREAD`) service and, as a side effect, the `IoFuncCancel` (`P_FCANCEL`) service causing the read request to time out. This type of modification is transparent to an application. It may use the `IoFuncRead` service identically regardless of whether this time out driver has been attached (the `IoFuncRead` service will of course not time out if this driver has not been attached).

Because the `IoFuncRead` service effectively runs two I/O requests, that is the lower driver's `IoFuncRead` and a timer channel's `IoFuncRead`, both requests must be cancelled by the example driver if the application wishes to cancel the original request. Thus the example driver is required to add functionality to the `IoFuncCancel` service.

It should be noted that an `IoFuncClose` (`P_FCLOSE`) request should only detach itself from the lower driver and release any resources allocated by this drivers open function. It should not attempt to pass on the `IoFuncClose` request to the lower level driver (after a detach the I/O system will no longer be able to route any I/O requests to a lower driver).

The Wait Handler Function

When an application makes an `IoFuncRead` request, the example driver makes two asynchronous I/O requests, one on the timer and one on the lower driver. In order for the device driver to gain some

processing time, so as to find out what happened to these requests, it needs to enable the already installed wait handler routine. When the I/O semaphore is signalled, the wait handler function will be called by the operating system (only if the application is currently waiting for an I/O request to complete) so it can check to see if either of the two asynchronous requests that it made have completed.

It is possible that neither of the outstanding requests has completed in which case the wait handler function should return with the carry flag clear.

If either of the outstanding requests has completed, this driver cancels the other request using up the signal generated by cancelling. The wait handler should return with the carry flag set and the AL register set to zero since there is no more processing to do at this time.

Although synchronous I/O requests are used within the wait handler (in cancelling and using up signals) the wait handler will not be called, i.e. it is not called re-entrantly by the operating system.

Non Interrupt Based Sound Driver

The code in *snddvr.asm* contains an example of a root device driver that does not require interrupt service routines.

The driver accesses the sound chip within the Series3 and can play notes passed to it from an application.

The sound system within a Series3 can only be accessed by a single process at a time. The operating system has some state variables that can be used (via system services) as mutual exclusion semaphores.

When the channel to the sound driver is opened, it requests exclusive use of the sound system. When the channel is closed it releases this resource. The hardware sound device is switched on only when sound is to be played.

The example device driver times the duration of the notes using a system timer. This limits the device driver to ten notes per second as the system timer can not go beyond a resolution of one tenth of a second. This is not a particularly high resolution for the note duration.

As well as the system timer, the device driver makes use of a wait handler function in order play the notes.

The example code in *t_mus.c* shows how the device driver is loaded and the functions provided are used to generate sound.

The device table

The start of the file contains the device driver header. The name of the driver is specified to be `MUS:`. The driver is an LDD type driver consisting of nine callable functions, the first eight of which are mandatory. The ninth function (nominated to be a wait handler function) is required by the device driver to function correctly.

Note that the following notes apply specifically to the functions as used in the example driver.

The Install Function

The install function should indicate that the device driver has no channel open on it yet. This variable (in the device driver space) is required to know how to handle the remove, hold and resume requests.

The Remove Function

If the device driver currently owns the sound channel, the remove function will stop the playing of sound and release to the operating system the sound channel resource.

In the normal course of events, the remove function would not be called when an application has an open channel to the device driver. It is however quite possible for this to occur and a device driver should accommodate such a possibility. The `StopSound` routine and `HwFreeCombo` operating system service should only be called if the driver owns the sound channel otherwise any sound and ownership from other device drivers (e.g. alarms) will be adversely affected when this driver is removed.

The Hold Function

The hold function will stop any sound that is currently being made if this driver currently owns the systems sound resource. This device driver does not attempt to determine how far through the current note (duration) it has got. When the resume function is called, the following note (if any) will be played.

The Resume Function

The resume function will, if the driver currently owns the systems sound resource, simply switch back on the hardware sound device. No note is played at this point. Because the driver uses the services of the timer device driver, any outstanding timeout will eventually complete causing the device driver's wait handler routine to be run. This in turn determines whether or not more notes are to be played.

The Reset Function

Since the reset function can only be called when there is an open channel, the device driver does not have to check that it owns the systems sound resource. The reset function simply stops any current sound, powers down the sound system and marks the channel as closed.

The Units Function

The sound system can support no more than one user; thus the device driver supports no more than one open channel at any given time. Note that the device driver will report that it supports one sound channel whether or not that one channel is available.

The Open Function

The open function is called by the operating system when an application wishes to obtain a channel to a device called `MUS:` (the example device driver's name). The open function attempts to obtain exclusive use of the sound resource by calling the `HwGetCombo` operating system service. It returns with the carry flag clear if the driver has successfully obtained the sound resource.

The open function then allocates the I/O control block, adds function nine as a wait handler function and obtains a channel to a timer device. If any of these requests fail, the driver tidies up after itself.

It finally requests that the operating system call the reset function if the client terminates without closing the I/O channel. Once successfully opened, the internal variable is set to indicate this.

The Strategy Function

The example device driver supports three functions: playing sound, cancelling the playing and closing the channel. This implementation uses the `IoFuncWrite` (`P_FWRITE`) service to play sound, the `IoFuncCancel` (`P_FCANCEL`) service to cancel playing and the `IoFuncClose` (`P_FCLOSE`) service to close the channel.

The playing of a sound is achieved by writing the user specified note at the required volume to the sound generation chip. The specified timeout is used to queue a timeout request on the timer channel opened by this device driver. When the timeout occurs, the timer device driver will write the completion status word and signal the I/O semaphore. Providing the application is waiting for an I/O request to complete, the device driver's wait handler will be called. The wait handler writes the next note into the sound chip thus playing the required tune.

Here lies a fundamental difference between wait handlers and interrupt service routines. Not only does an application have to be waiting for an I/O request to complete but it must also be able to obtain processing time in which to run the wait handler code. If a higher priority process is using all of the CPU (even if only for a short period of time), the sound application will not get a chance to run the wait handler function. Thus applications using this device driver will find that notes sometimes play for a lot longer than originally intended.

This effect can be observed by running the example program and switching to the system task (by pressing the *System* button on the Series3 machine for example) forcing the system to update the lists.

The `IoFuncCancel` service simply cancels any outstanding write request by cancelling the outstanding timer request, waiting for its completion and then completing the write request with the `E_FILE_CANCEL` completion status. The wait handler is also disabled, primarily for system performance reasons.

The `IoFuncClose` request will cancel any outstanding write close the timer channel, cancel the reset request and release the sound resource back to the operating system.

The Wait Handler Function

This function should check whether the outstanding timer request has completed and, if so, start playing the next note. If there are no more notes to play, the original write request is completed with zero completion status.

Exercising the vectors

The hold and resume vectors can be exercised simply by switching the machine off and then back on again. The sound should stop when switched off and resume when switched back on.

The reset function can be exercised by running the example program, switching to the system task and terminating the example program. If the example program can be re-run and generate sound and/or the alarms still work then the driver has tidied up any system resources it needed to.

Interrupt Driven Sound Driver

The code in *sndfrc.asm* contains an example of a root device driver that uses the FRC (free running counter) as an interrupt source to drive a sound system.

The driver accesses the sound chip within the Series3 and has the ability to play the notes passed to it from an application.

The sound system within a Series3 should only be accessed by a single process at a time. The operating system has some state variables that can be used (via system services) as mutual exclusion semaphores.

Similarly the FRC should only be accessed by a single process at a time. If a second process grabs the FRC without the first knowing then the first will probably never receive another FRC interrupt and hence appear to hang.

When the channel to the sound driver is opened, it requests exclusive use of the sound system and the FRC; when the channel is closed it releases these resources. It is only when some sound is to be made, that the hardware is switched on and the notes played.

This device driver uses the FRC as a timer to time the duration of notes. The FRC can be programmed to run at either 32Hz or 512kHz, thus allowing a much greater resolution than the system timer device driver that can only provide 1/10th of a second resolution. This version allows for a minimum duration of 10ms, allowing notes of shorter duration and hence increasing the frequency with which the interrupt service routine is called causing a very high percentage of the processor bandwidth to be used.

For this driver an eight bit number is used to specify the duration of a note and hence the maximum duration is 2.56 seconds.

Since the notes are changed in the interrupt service routine which is independent of most other system activity, a high level of accuracy of note duration can be obtained.

The example code in *t_musfrc.c* shows how the device driver is loaded and the functions provided are used to generate sound.

The device table

The start of the file contains the device driver header. The name of the driver is specified to be `MUS::` The driver is an LDD consisting of nine callable functions, the first eight of which are mandatory. The ninth function (nominated to be a wait handler function) is required by the device driver in order to function correctly.

Note that the following notes apply specifically to the functions as used in the example driver.

The Install Function

The install function should set the fact that the device driver has no channel open on it yet. This variable (in the device driver space) is required to know how to handle the remove, hold and resume requests.

The Remove Function

The remove function will, if the driver currently owns the systems sound resource, stop any sound being made, stop the FRC from generating interrupts, remove the FRC interrupt service routine address and free the sound channel.

In the normal course of events, the remove function would not be called when an application has an open channel to the device driver. It is however quite possible for this to occur and a device driver should accommodate such a possibility. The `StopSound` routine and `HwFreeCombo` operating system service should only be called if the driver owns the sound channel, otherwise any sound and ownership from other device drivers (eg alarms) will be adversely affected when this driver is removed.

The Hold Function

The hold function will, if the driver currently owns the systems sound resource, stop any sound that is currently being made and stop the FRC from generating interrupts.

The device driver does not attempt to determine how far through the current note (duration) it has got; when the resume function is called the following note (if any) will be played.

The Resume Function

The resume function will, if the driver currently owns the systems sound resource, reset the FRC interrupt service routine address held by the operating system to point to the drivers interrupt service routine code. The device driver may have moved in memory and thus the absolute CS address will be wrong. If a write request is currently queued (i.e. a hold request was made while the driver was playing some sounds), the sound hardware and FRC interrupts are re-enabled.

After a power down type hold the FRC is reset and hence needs to be re-programmed to generate interrupts at the required frequency. With a memory move type hold the FRC will continue its countdown and, if the count reaches zero, generate the next interrupt. This will be ignored by this device driver since the FRC is re-programmed in the reset function.

The Reset Function

Since the reset function can only be called when there is an open channel the device driver does not have to check that it owns the systems sound resource. The reset function stops any current sound, powers down the sound system, disables FRC interrupts, resets the FRC interrupt address back to the default address and marks the channel as closed.

The Units Function

The sound system will support no more than one user and hence the device driver supports only one open channel at any given time. Note that the device driver does not guarantee that a channel is available by reporting that it supports one channel; another system component may own the sound resource when the open request is made.

The Open Function

The open function is called by the operating system when an application wishes to obtain a channel to a device called `MUS:` (our device driver name). The open function attempts to obtain exclusive use of the sound resource by calling the `HwGetCombo` operating system service. If this returns with the carry flag clear then the driver has successfully obtained the sound resource.

The open function then attempts to obtain exclusive use of the FRC resource by calling `HwGetChannel` with the interrupt channel number as the hardware resource required. If this returns with the carry flag clear then the driver has successfully obtained the FRC resource.

The open function then allocates the I/O control block, adds function nine as a wait handler function, and requests that the operating system call the reset function if the client terminates without closing the I/O channel.

Once successfully opened, the `SoundChan.Pid` internal variable is set to the process id of the client process (required by the interrupt service routine) and the `SoundChan.OpenFrcChan` is set to indicate the channel has been opened (as required by the hold, resume and remove vectors).

The Strategy Function

This device driver supports three functions: playing sound, cancelling the playing and closing the channel. This implementation chose to use the `IoFuncWrite` (`P_FWRITE`) service to mean play sound, the `IoFuncCancel` (`P_FCANCEL`) service to cancel playing and the `IoFuncClose` (`P_FCLOSE`) service to close the channel.

The playing of a sound is achieved by writing the user specified note at the required volume to the sound generation chip. The FRC interrupt service routine is the routine that actually writes the data to the sound chip. As it is an interrupt service routine, it cannot gain addressability to its client data space. Thus the sound data to be played must be copied into the device driver space. This has a side effect that the maximum number of notes that can be played (by this driver) is limited by the size of the internal buffer as defined by the driver. The buffer can be made any size although in the majority of cases a larger buffer will most probably waste space.

The device driver could be written to handle arbitrarily long sequences of notes. When the interrupt service routine reaches a 'low water mark' of notes to play, it can signal the wait handler routine which

will eventually run to copy more data from the client process space into the internal buffer. This is, however, a fairly complex task.

When all the notes in the device driver's internal buffer have been played, the interrupt service routine will call the `IoSignalByPidNoReSched` system service to signal the client process that the playing of sound has completed.

The wait handler function will pick up the fact that the interrupt service routine has finished (from the `SoundChan.WriteStat` variable) and report the completion status to the client; the wait handler runs in the context of the client process and can thus write back the completion status word.

The `IoFuncCancel` service simply cancels any outstanding write request by stopping any more FRC interrupts, stopping the sound generation, switching off the hardware and then completing the write request with the `E_FILE_CANCEL` completion status. The wait handler is also disabled, primarily for system performance reasons.

The `IoFuncClose` request will cancel any outstanding write, cancel the reset request and release the sound and FRC resources back to the operating system.

The Wait Handler Function

This function should check whether all the notes have been played in which case it has completed the users request.

CHAPTER 9

PCMCIA CARDS AND SSDs

Certain manufacturers of hand-held computers have recently been vigorously promoting the so-called *PCMCIA* standard for plug-in cards for portable computers. The idea is that PCMCIA cards that can plug into one hand-held computer ought to be able to plug just as easily into another.

However, the SSDs used in SIBO computers do not conform to this standard. This fact may be seen as a potential problem - possibly as a disincentive against taking the time to learn how to program within the Sibosdk system. More precisely, it may be seen as a reason why the SIBO range of computers might enjoy but a limited lifetime in the marketplace, before being eclipsed by other, PCMCIA-based portable computers - in which case, studying the Sibosdk system in any detail would be a poor investment.

This chapter aims to answer these worries by dispelling the hype around PCMCIA, through documenting some of the considerable benefits of SSDs as compared to PCMCIA cards.

Mobility and robustness

Serious doubts can be raised over the suitedness of PCMCIA cards to genuinely mobile computing. The simple fact is that SSDs are much more robust and portable than PCMCIA cards.

In the first place, these doubts revolve around the fact that PCMCIA cards require no fewer than *68* independent connections to the main body of the computer. This is an enormously large number of connections for hardware to protect, and contrasts vividly with the *6* connections of SSDs.

There are two aspects to this: *mechanical* and *electrostatic*. At the mechanical level, the 68 pin edge connector of PCMCIA cards is inevitably vulnerable to pin failure with constant removal and insertion. At the electrostatic level, the fact that there are only two data lines in the SSD Serial Protocol means that each data line can be amply electrostatically protected in a way impossible for PCMCIA cards. The PCMCIA parallel interface has many more data lines, increasing the potential for data corruption.

The very name "PCMCIA" betrays the origin of this proposed standard as an accessory to the basic "PC" architecture. PCMCIA was fundamentally designed as a plug-in extension of the memory map of PCs; in contrast, SSDs were designed with secure data storage, ruggedness, and high mobility as the foremost priorities. (SSDs actually developed out of Psion's long experience with DataPacks on the Organiser range, in which it became clear how vital it is to keep data transmission lines to a minimum.)

Size considerations

Another important point is the sheer physical size of the PCMCIA cards. Although slightly thinner than SSDs, they have a much larger body area - famously, the same profile as that of a credit card.

For example, were HCs to be converted, hypothetically, to the PCMCIA standard, it would mean only having one drive, instead of two as at present. There would be no room in a standard-sized HC for two PCMCIA card slots. Yet practical experience demonstrates how crucial having two SSD drives is - one SSD (possibly a one-time-programmable one) containing a program, and another containing data files.

In general, the greater body area of the PCMCIA card means that SSDs are suited to a wider range of different application devices.

Different standards for different purposes

In proposing their standard, the originators of PCMCIA had various different goals in mind for what PCMCIA cards could do. In retrospect, the standard seems appropriate only to *some* of these goals.

PCMCIA is at its most impressive as a standard for detachable plug-in extensions to PC RAM memory; it is significantly *less* impressive as a standard for secure off-line data storage. This is where the SSD format (in particular, the *Psion Serial Protocol*) wins out.

It seems more than likely that the Psion Serial Protocol will *also* become a de-facto industry standard, alongside the PCMCIA standard. Just as various different computers, from a variety of manufacturers, currently contain PCMCIA card slots, so too will SSDs become incorporated in an ever wider range of computers and computer peripherals. (But it should be borne in mind that there are already not one but *three* different PCMCIA standards, whereas there is only one Psion Serial Protocol standard.)

To this end, Psion is actively supporting interested third parties, supplying relevant ASICs, more detailed documentation (see the *Hardware Reference* manual in the first instance), and other practical assistance. Contact Psion for more information about the *SSD Hardware Development Kit*.

Hot insertion

Compared to PCMCIA cards, SSDs offer another significant advantage in "hot insertion". What this means is that SSDs can in general be inserted or removed from a computer *without that computer first having to be powered down*. (After all, PCMCIA cards plug directly into the bus of a PC or PC-clone.)

Now whilst this is not directly relevant to the case of the HC (which automatically switches off whenever its rear cover is opened), it *is* highly relevant for other computers in the SIBO range - such as the laptop MC and the ubiquitous Series3. It is also relevant for PCs with attached SSD drives.

More generally, support for hot insertion makes SSDs behave like natural and logical extensions of the floppy disks of a PC. An SSD can be removed from one computer and plugged into another just as easily as a floppy disk can; there is no need to power the computer down first.

In order to achieve a similar result, PCMCIA cards need the addition of extra memory buffers, both in the card and in the reading device - adding to the cost and design complexity of both.

Flash filing systems

On the particular subject of Flash PCMCIA cards, whilst it may be easy for applications to *read* data from such cards, it is much harder for them to *write data back*. This requires a sophisticated filing system which is very different from the FAT (File Allocation Table) filing systems that work so well with RAM memory.

At the time of writing, no full-featured Flash filing system exists for any PCMCIA card, and there seems little prospect of one being produced in the near future. This absence contrasts sharply with the mature SSD Flash filing system that is contained within Epoc.

Flash SSDs have been shipping since 1989, and Psion is now widely recognised as the world's leading authority in the use of Flash memory.

Cost considerations

In view of these problems facing PCMCIA cards, it is hardly surprising that more SSDs are being produced than cards conforming to PCMCIA. As a result, SSDs are well on the way (at the time of writing) to being significantly cheaper.

Another factor favouring SSDs being cheaper than PCMCIA cards is the simpler nature of their basic construction. Inevitably, it is mechanically harder to manufacture a 68 pin connector. Note that this consideration is just as pertinent for the socket the memory card plugs into, as for the memory card itself.

Architectural openness

Possibly the most significant point of all still remains to be made. Namely, the SSD interface makes no presumption on the architecture of the system that is driving it - unlike PCMCIA which is designed for PC-hardware compatible systems. This gives the designer of the system considerably greater freedom: SSDs are truly independent of the host system.

To re-emphasise: SSDs do not presuppose the Epoc or Sibosdk architectures in any way. The interface to SSDs is actually extremely flexible, and can be addressed:

- via hardware, using an ASIC-2 card, or
- via software, using just two microprocessor i/o lines.

Finally, two further points, each emphasising the greater flexibility offered by SSDs to system designers:

- at the level of engineering layout, SSDs can be located significantly further away from the main processor than PCMCIA cards (by virtue of the different interfaces)
- there is no intrinsic design constraint to the address limit of SSDs, providing potentially unlimited capacity.