SIBO 'C' Software Development Kit

PLIB REFERENCE

Version 2.10

February 3, 1995

(C) Copyright Psion PLC 1990-95

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

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

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

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

Contents

1 Introduction 1

PLIB, SIBO and EPOC 1
The SIBO architecture 1
The EPOC operating system 2

The EPOC programming environment 2
Small programming model 2
Hardware protection 3
More about memory moving and the 8086 segment registers 3
The Clarion TopSpeed C compiler 3
System services 4
PLIB header files 4
p_std.h 5
Calling conventions 6
Small programs 7
The PLIB C startup modules 7

Related reference manuals 8
TopSpeed C library reference 8
Window server reference 8
I/O devices reference 9
EPOC O/S System Services reference manual 9
Object dynamic libraries 10

2 Characters, Strings and Buffers 11

General string and buffer functions 11
Copy memory to memory (p_bcpy) 11
Return string length (p_slen) 11
Copy a string (p_scpy) 11
Copy multiple strings (p_scpym) 12
Concatenate two strings (p_scat) 12
Concatenate many strings (p_scatm) 12
Replicate a buffer (p_brep) 12
Replicate a string (p_srep) 13
Swap two buffers (p_bswap) 13
Fill a buffer with a value (p_bfil) 13
Align buffer (p_jtob) 13
Generate the CRC number (p_crc) 14

Character classification and conversion 14
Test for upper case character (p_isupper) 15
Test for lower case character (p_islower) 15
Test for alphabetic character (p_isalpha) 15
Test for numeric digit (p_isdigit) 16
Test for alphanumeric character (p_isalnum) 16
Test for hexadecimal digit (p_isxdigit) 16
Test for whitespace character (p_isspace) 16
Test for control character (p_iscntrl) 16
Test for punctuation character (p_ispunct) 16
Test for printable graphic character (p_isgraph) 16
Test for printable character (p_isprint) 16
Skip whitespace characters (p_skipwh) 16
Skip non-whitespace characters (p_skipch) 17
Fold a character (p_tofold) 17
Copy string with fold (p_scpyf) 17
Fold string (p_sconf) 17
Convert character to upper case (p_toupper) 17
Convert character to lower case (p_tolower) 17

Error returns 56
Convert error number to string (p_errs) 56

Notifier services 56
Present the user with a message and get response (p_notify) 57
Notify user of error and get response (p_notifyerr) 58
Set notify state (p_setnotify) 58
Get notify state (p_getnotify) 58
Hook the notifier interface (p_notifyhook) 58
Unhook the notifier interface (p_notifyunhook) 58

Enter and leave 58
Enter a function (p_enter) 59
Unwind stack and return from last p_enter (p_leave) 60
Unwind stack and return from last p_enter if error (f_leave) 60

7 Memory Allocation 63

Overview of system memory usage 63
Memory segments 64
Segment names 64
Segment handle, address and size 64
Process data segments 65

The heap allocator 66
Heap structure 66
Growing and shrinking the heap 66
Alloc heaven 67
Internal fragmentation 67
Allocate a memory cell (p_alloc) 67
Free an allocated cell (p_free) 68
Change cell size (p_realloc) 68
Insert or delete data in cell (p_adjust) 68
Get cell length (p_alen) 69
Set heap granularity (p_hgran) 69
Visit all cells (p_allwalk) 69
Check heap integrity (p_allchk) 70
Get heap address and potential free space (p_allspc) 70

System memory usage 71
Get addressable system RAM size (p_getram) 71
Get total system RAM size (p_totalK) 71
Get size of available segmented memory (p_sgfree) 71
Get memory used by internal RAM disk (p_sgramdisk) 71

Memory segments 71
Create memory segment (p_sgcreate) 72
Delete memory segment (p_sgdelete) 73
Open memory segment (p_sgopen) 73
Copy to memory segment (p_sgcopyto) 73
Copy from a memory segment (p_sgcopyfr) 73
Get size of memory segment (p_sgsize) 74
Adjust the size of a memory segment (p_sgadjust) 74
Find segments by name (p_sgfind) 74
Close memory segment (p_sgclose) 74
Increment segment usage count (p_sglock) 75
Decrement segment usage count (p_sgunlock) 75

Environment variables 75
Get environment variable value (p_getenv) 75
Get environment variable value (p_getenviron) 76
Set environment variable value (p_setenv) 76
Set environment variable value (p_setenviron) 76
Delete environment variable (p_delenv) 76
Delete environment variable (p_delenviron) 77
Find environment variables (p_fndenv) 77
Find environment variables (p_findenviron) 77

8 Asynchronous Requests and Semaphores 79

Semaphores 79
Process scheduling 79
Shared access 80
Supplier-consumer 80

Asynchronous requests 80
The I/O semaphore 81
Status words 81
Cancelling an asynchronous request 82
Waiting for a particular completion 82
Constructing synchronous functions 82
Wait handlers 83
Polling rather than waiting 84
Attached I/O devices 84

Primitive semaphore functions 84
Create a semaphore (p_semcrt) 84
Delete a semaphore (p_semdel) 84
Wait on a semaphore (p_wait) 85
Signal a semaphore (p_signal) 85
Signal a semaphore n times (p_signaln) 85
Signal a semaphore with no re-schedule (p_signalnr) 85

The I/O semaphore 85
Signal the IO semaphore (p_iosignal) 85
Signal the IO semaphore of another process (p_iosignalbypid) 85
Wait on the IO semaphore (p_iowait) 86
Allow any wait handlers to run (p_ioyield) 86
Wait for a particular request to complete (p_waitstat) 86

Wait handlers 87
Add a wait handler function (p_svecadd) 87
Activate/deactivate a wait handler (p_sveccall) 88
Remove a wait handler (p_svecrem) 88

9 I/O System 89

I/O Device Drivers 89
LDDs and PDDs 89
External device drivers 89
Opening a channel to a device 90
Operations on an open I/O channel 90
The file server 91
Attached drivers 91

Channel-based I/O functions 92
Open a channel to a device (p_open) 92
Start an I/O operation (p_ioa) 93
Start an I/O operation with guaranteed completion (p_ioc) 94
Start an I/O operation and wait for completion (p_iow) 95
Close an I/O channel (p_close) 96
Read from an I/O channel (p_read) 96
Write to an I/O channel (p_write) 97
Cancel requests on an I/O channel (p_iow(P_FCANCEL)) 97

Device driver functions 98
Load a logical device driver (p_loadldd) 98
Load a physical device driver (p_loadpdd) 98
Delete a device driver (p_devdel) 98
Query the number of units supported by a device (p_devqu) 98
Find all devices (p_devfnd) 99

Simple console I/O 99
Redirecting console writes 100
Changing the size of the console window 100
Changing the console window mode 100
Write a character to the console (p_putch) 101

Write a string to the console (p_puts) ... 101
Convert arguments and write line to console (p_printf) ... 101
Convert arguments and write to console (p_print) ... 101
Get a character from the console (p_getch) ... 101
Get a string from the console (p_gets) ... 101
Get a string with prompt from the console (p_getl) ... 101

10 Time, Timers and Dates ... 103

System time ... 103
Return the system time (p_date) ... 103
Set the system time (p_sdate) ... 103

Absolute and relative timers ... 103
Suspend process for n tenths of a second (p_sleep) ... 104
Suspend process for n system ticks (p_sleept) ... 104
Suspend process until absolute time (p_sleepa) ... 105

Asynchronous timers ... 105
Open a timer channel (p_open("TIM:")) ... 105
Start a relative timer (p_ioc(P_FRELATIVE)) ... 105
Start an absolute timer (p_ioc(P_FABSOLUTE)) ... 106
Cancel a timer (p_iow(P_FCANCEL)) ... 106
Close a timer channel (p_close) ... 106

Converting between binary representations of time ... 106
Convert system time to P_DAYSEC time (p_sttods) ... 107
Convert P_DAYSEC time to system time (p_dstost) ... 107
Convert P_DAYSEC time to P_DATE time (p_dstodt) ... 107
Convert P_DATE time to P_DAYSEC time (p_dttods) ... 108
Find the number of days in the specified month (p_dayinm) ... 108
Convert day since 1900 to day in week (p_wkday) ... 108
Calculate week number in year (p_weekno) ... 108

Time and date components in text form ... 109
Get the day name (p_nmday) ... 109
Get the day name abbreviation (p_nmdaya) ... 110
Get the month name (p_nmmon) ... 110
Get the month name abbreviation (p_nmmona) ... 110
Get the day-in-month suffixes (p_getsuffixes) ... 110
Get the am and pm suffixes (p_getampmtext) ... 110
Get time representation preferences (p_getctd) ... 111

Generating time and/or date strings ... 112
Date and time format strings ... 112
Convert a P_DATE time to a string (p_dt2str) ... 115
Convert a P_DAYSEC time to a string (p_ds2str) ... 115
Convert a system time to a string (p_st2str) ... 115
Convert the current time to a string (p_now2str) ... 115

11 Files ... 117

Files in EPOC ... 117
The file server ... 117
File systems ... 117
SSD drives ... 118
Unattended applications ... 118
RAM SSDs ... 119
Flash SSDs ... 119
File specifications ... 120
Default path ... 121
Channel-based services ... 121
Non-channel-based services ... 121
Asynchronous file operations ... 122

Manipulating file specifications ... 123
Parse a file specification (p_fparse) ... 123
Using p_fparse across filing systems ... 124

Change the directory in a file specification (p_chdir) 125

The default node, device and directory 126
Set the system-wide default path (p_setdefaultpath) 126
Set the default path of this process (p_setpth) 126
Get the default path of this process (p_getpth) 127
Get the default path by process id (p_getpthbyid) 127

Operations on nodes and devices 127
Get a list of nodes (p_open(P_FNODE)) 127
Get node information (p_ninfo) 128
Check if LOC:: has changed (p_locchg) 128
Get a list of devices (p_open(P_FDEVICE)) 128
Get device information (p_dinfo) 129
Format a device (p_open(P_FFORMAT)) 131
Read media information of a local device (p_locdevice) 132
Direct read of local SSD (p_locreadpdd) 133

Operations on directories and files 133
Get a directory list (p_open(P_FDIR)) 133
Return file information (p_finfo) 135
Test for the existence of a directory (p_testpth) 136
Rename a file or directory (p_rename) 136
Delete a file or directory (p_delete) 137
Make a new directory (p_mkdir) 138
Set file attributes or label medium (p_sfstat) 138
Set file creation date (p_fdate) 139

Binary file access 140
Shared access 140
Example of binary file access 140
Open a binary file (p_open(P_FSTREAM)) 142
Close a binary file channel (p_close) 143
Read from a binary file channel (p_read) 144
Write to a binary file channel (p_write) 144
Position a binary file channel (p_seek) 144
Flush internal file buffers (p_iow(P_FFLUSH)) 145
Set end of file (p_iow(P_FSETEOF)) 146
Cancel an asynchronous file channel request (p_iow(P_FCANCEL)) 146

Stream text file access 146
Open a stream text file (p_open(P_FSTREAM_TEXT)) 147

Text file access 147
Open a text file (p_open(P_FTEXT)) 148
Close a text file channel (p_close) 148
Read from a text file channel (p_read) 148
Write to a text file channel (p_write) 149
Position a text file channel (p_seek) 149
Flush internal file buffers (p_iow(P_FFLUSH)) 150
Set end of text file (p_iow(P_FSETEOF)) 150
Cancel an asynchronous file channel request (p_iow(P_FCANCEL)) 150

12 Processes and Inter-Process Messaging 151

Processes 151
System processes 152
Process ID and process control block 152
Process states 154
Process queues 154
Process priorities 154
Preemptive scheduling 155
Process names 155
Reserved statics (magic statics) 156
Shared code segments 158
Image files 158

Process termination 159

Creating a process 160
Load an image (p_execc) 160
Load an image asynchronously (p_execcasync) 162
Create a process (p_pcreate) 162

Operations on the current process 163
Get this process ID (p_getpid) 164
Mark this process as non-active (p_unmarka) 164
Register activity (p_tickle) 164
Mark this process as active (p_marka) 164

Operations on any process 165
Get a process priority (p_getpri) 165
Set a process priority (p_setpri) 165
Resume a process (p_presume) 165
Suspend a process (p_psuspend) 165
Get a process name by ID (p_pname) 165
Rename a process (p_prename) 166
Get a process ID by name (p_pidfind) 166
Find all processes (p_pfind) 166
Determine the owner of a process (p_getowner) 166

Accessing a process data segment 167
Copy data from a process (p_pcpyfr) 167
Indirected string copy from a process (p_piscpyfr) 167
Copy data to a process (p_pcpyto) 167

Inter-process messaging 168
Message slots 168
What the server does 169
What the client does 169
An example of a server 169
Corresponding client code example 170
Asynchronous messaging 171
Message processing order 172

Server functions 172
Initialise for message reception (p_minit) 173
Wait for message reception (p_mreceivew) 173
Asynchronous message reception (p_mreceive) 173
Cancel a message receive request (p_mcancel) 174
Free a message (p_mfree) 174

Client functions 174
Send a message (p_msend) 174
Send a message and wait for a reply (p_msendreceivew) 175
Asynchronous send message and get reply (p_msendreceivea) 175

13 General System Services 177

System information 177
Get the operating system version (p_version) 177
Get the ROM version (p_romversion) 177
Get the cause of the last system shut-down (p_getres) 177
Get operating system data (p_getosd) 178
Get power supply type (p_getpsu) 178

Language and country 178
Get the language code (p_getlanguage) 179
Get operating system text (p_gettext) 179
Get country-dependent data (p_getctd) 180
Set country-dependent data (p_setctd) 180

Switching on and off 180
Switch off (p_off) 181
Get the auto-switch-off period (p_getauto) 181
Set the auto-switch-off period (p_setauto) 181

Get switch-off state when mains is present (p_getautomains) 181
Disable/enable switch-off if mains is present (p_setautomains) 181
Allow auto switch off (p_allowoff) 181
Enable/disable the ON key event (p_setonevent) 182

Power supply 182
Get power supply status (p_supply) 182
Get additional power supply data (p_supplyinfo) 183
Get battery warning and maximum levels (p_wsupply) 184
Get the battery type (p_getbat) 185
Set the battery type (p_setbat) 185

Keyboard 185
Get the state of all keys (p_getscancodes) 185

Display 186
Get the system display type (p_getlcd) 186
Change the LCD contrast (p_lcdcontrastdelta) 186
Get the current LCD contrast (p_getlcdcontrast) 186
Switch the backlight on or off (p_backlight) 186
Set the backlight control value (p_setbacklight) 187
Get the backlight enablement (p_getbacklight) 187

Sound 187
Make a sound with the piezo (p_sound) 187
Get the sound flags (p_getsnd) 188
Set the sound flags (p_setsnd) 188

Sound on the Series 3a 188
Sound files 188
The A-Law encoding scheme 189

Series 3a sound system services 192
Record a sound asynchronously (p_recordsounda) 192
Cancel sound recording (p_recordsoundcancel) 192
Record a sound synchronously (p_recordsoundw) 193

Play back a sound asynchronously (p_playsounda) 193
Cancel sound playback (p_playsoundcancel) 193

Play back a sound synchronously (p_playsounda) 193

Exit the system 194
Exit to DOS (p_hwexit) 194

14 Database Files 195

Overview of database files 195
The file header 196
Records 196
String fields 197
Number of records 198
End of file record 198
Database files and OPL 198

DBF functions 198
Open a database file (DbfOpen) 198
Open a database file (DbfQuickOpen) 200
Close a database file (DbfClose) 200
Flush a database file (DbfFlush) 200
Notify that the DBF buffer has been overwritten (DbfTrash) 200
Copy down a DBF record (DbfCopyDown) 201
Compress a database file (DbfCompress) 201
Copy a database file (DbfCopyFile) 201
Find the size of a database file (DbfFileSize) 202
Read a DBF extended header (DbfExtHeaderRead) 202
Write a DBF extended header (DbfExtHeaderWrite) 203
Read a DBF descriptive record (DbfDescRecordRead) 203
Write a DBF descriptive record (DbfDescRecordWrite) 203
Get the DBF version number (DbfVersion) 203
Read a specific DBF record (DbfAbsRead) 204

Read and sense a specific DBF record (DbfAbsReadSense) 204
Read the next DBF record (DbfNextRead) 204
Read the previous DBF record (DbfBackRead) 204
Read the first DBF record (DbfFirstRead) 205
Read the last DBF record (DbfLastRead) 205
Append a DBF record (DbfAppend) 205
Erase a DBF record (DbfEraseRead) 206
Update a DBF record (DbfUpdate) 206
Find a DBF record (DbfFindReadField) 207
Find a DBF record (DbfFindRead) 208
Sense the current DBF record number (DbfSense) 209
Count the number of DBF records (DbfCount) 209

15 Object Oriented Programming 211

Classes 211
Class descriptor 211
Object instances 212
Object destruction 212

Categories 212
Category handles 213
Category numbers 213
Dynamic linkage 213
Referencing by category handle 214
DYLs 214
The structure of a loaded and linked category 216
The structure of an unlinked category 216
What happens during dynamic linkage 216

Message passing 216
Calling conventions for method functions 217
Performance of message sending 218

DLLs 219

Category functions 220
Load a DYL (p_loadlib) 220
Open a file containing multiple DYLs (p_openlib) 221
Load from a multiple DYL file (p_loadfilelib) 221
Unload a dynamic library (p_unloadlib) 221
Link a loaded category (p_linklib) 221
Find a category handle (p_findlib) 222
Convert a category number to a handle (p_getlibh) 222
Copy data from a category (p_cpycat) 222
Copy data from the local category (p_ccpy) 222

Object functions 223
Create an object by category number (p_new, f_new) 223
Create an object by category handle (p_newlibh, f_newlibh) 223
Send a message to an object (p_send) 224
Send a message to be handled by the superclass (p_supersend) 224
Send a message with an enclosing p_enter (p_entersend) 225
Send a message to a specific class (p_exactsend) 225
Create and initialise an object by category number (f_newsend) 225
Create and initialise an object by category handle (f_newlibhsend) 226
Reclass an object by category number (p_reclass) 226
Reclass an object by category handle (p_reclassbyhandle) 226

CHAPTER 1

INTRODUCTION

PLIB, SIBO and EPOC

PLIB is a library of C functions that are used to access the system services in the ROM of a *SIBO machine* that is running the *EPOC operating system*.

PLIB may be used in isolation or it may be used in conjunction with other libraries, including:

CLIB	the EPOC version of the standard C library
WLIB	the window server library (for graphics output)

CLIB and WLIB are described further in the section *Related Reference Manuals* at the end of this chapter.

The SIBO architecture

A SIBO machine is a battery-powered portable computer that is based on the SIBO architecture. This architecture is designed to minimise the size, weight and power consumption of the computer. The key components of the architecture are:

- A sophisticated power management system that selectively powers subsystems under software control
- Solid State Disks (SSDs) that provide fast low-power silicon-based mass storage with no moving parts
- A synchronous serial interface for peripherals running at high speed (Mega bit rates)
- An 8086 class of processor (or any compatible processor such as an 80286)
- Hardware protection of the system from aberrant processes (address trapping of out-of-range writes and a watch-dog timer on interrupts being disabled)
- Real-time clock
- ROM-resident system software
- Graphics LCD display
- A touch sensitive digitising pad that provides a pointing device (used in some models)
- ISDN combo sound system (used in some models)

The hardware architecture is primarily implemented in custom ICs called ASICs (at the time of writing, there were 7 different SIBO ASICs). The SIBO architecture uses surface-mounted static CMOS ICs throughout.

For further information see the *SIBO Hardware Reference* manual.

The EPOC operating system

The EPOC O/S, designed for the SIBO architecture, has the following features:

- preemptive multi-tasking
- MSDOS-compatible file system
- installable file systems, including remote file access
- asynchronous services
- support for client-server architectures (used to implement system components such as the file server and window server)
- a comprehensive I/O system with many built-in I/O devices
- dynamically loadable device drivers
- re-entrant function library
- multiple processes of the same program share a single copy of the code
- support for object-oriented programming
- code-shared dynamic link libraries

On SIBO machines, the system software resides on an in-built ROM. A version of the EPOC operating system also runs on a PC[1].

The EPOC programming environment

Small programming model

When programming in C for a PC, the C programmer chooses, normally by means of a compiler option, between various coding models (called eg small, compact, medium, large, huge). The choice of the model to be used depends primarily on the size of the program code and program data.

When programming in C for EPOC, you **must** program using the small model, in which the code and the data segment are each limited to 64K bytes. The restriction to the small model allows EPOC to move memory segments, including the process code and data segments, without any cooperation from applications. Being able to move memory segments around in a multi-tasking system (for example, as processes are created and destroyed) is vital for efficient RAM usage. There is a description of system memory usage at the beginning of the *Memory Allocation* chapter.

For reasons described below, program executables tend to be significantly smaller when built in EPOC compared to other environments (such as a PC) and the 64K code segment limit is less likely to be a problem than you might have first thought. It is also true that small model code is in any case more compact because function and variable addresses are 16-bit words.

Notwithstanding the above, if an application does require more than 64K of code, there are the following options:

- break up the application into a main process with multiple transient sub-processes
- implement part of the functionality in a server process where the services are accessed using inter-process messaging
- implement part of the functionality as a device driver, accessed via I/O system calls
- when using object-oriented programming (OOP) or otherwise, break up the program into multiple dynamic libraries containing external classes that are accessed by OOP message sending

The 64K *data* segment limit is normally large enough for the stack and miscellaneous data structures. Dynamic data structures are typically allocated from the "heap". This resides at the high address end of the data segment and can grow as the need arises.

When a program does require more than 64K of data, it is often because of a single data structure that can grow to a large size (such as, for example, a word processor document). Such potentially large data

[1] In this manual, the term PC is used to mean an IBM PC/XT/AT or compatible.

structures may be implemented in external memory segments where a particular segment can grow up to a limit of 512K bytes. However, the access to data in an external segment is not as convenient as it is for data in the process data segment.

Hardware protection

As mentioned earlier, the SIBO architecture provides some protection to the operating system from aberrant processes.

Unless a program takes steps to disable the protection, a process may not write outside its own data segment nor may it use the 8086 I/O instructions `IN` and `OUT`.

The watch-dog timer will terminate a process if it disables interrupts for more than 24 system ticks (three quarters of a second). Switching off interrupts for more than 100 micro-seconds is considered poor design.

None of the above happens ordinarily when programming in C to the small model. Although, (and of course this is the reason for providing the hardware protection), it can happen in a bugged program.

More about memory moving and the 8086 segment registers

When EPOC moves memory, it is the supervisor process (with process name `SYS$MANG.$02`) that actually does the work. (The supervisor and other system processes are described further in the chapter *Processes and Inter-Process Messaging*.)

The supervisor has two essential qualifications for the job of memory moving:

- it runs at a higher priority than any other process (which means that it is not interrupted in its task)
- its own data segment does not move (nor does its code segment since it is in ROM)

After completing the memory move, the supervisor adjusts the 8086 segment registers[2] of each process context to take account of moved memory segments. When a process other than the supervisor subsequently runs, it does so with suitably adjusted segment registers.

A segment register is adjusted (by the amount the memory segment moved) if it contains a value that is greater than or equal to the start of a moved memory segment and less than the end of the memory segment.

If the segment register points to the address of a memory segment that has either not moved or (if the segment is in the ROM) can't move, it is not adjusted. For example, if a process is running ROM code, its CS will not be adjusted. The DS, SS and ES registers of a process normally all point to the beginning of the process data segment. This is likely to be moved at some time or another (for example when another process terminates).

In order to be able to adjust the segment registers appropriately, the supervisor relies on the assumption that programs obey the following rules:

- not to store the segment registers in memory and then to restore them from memory (since the memory might be moved between the store and restore)
- not to store values other than bona fide memory segment values in a segment register (since an arbitrary value might happen to fall within range of a moved segment and get adjusted)

When using the segment registers in 8086 assembly language, it is actually difficult (since a segment register can only be loaded from memory) to avoid saving and restoring a register from memory. In practice, you protect the save and restore by disabling interrupts (which stops a context switch from happening) until after the restore.

When programming in C to the small model, the above rules are automatically obeyed. However, the compiler has to adhere *strictly* to the small model - sometimes called the *pure* small model.

The Clarion TopSpeed C compiler

When writing C for the EPOC operating system you must use the Clarion TopSpeed C compiler.

We chose the TopSpeed C compiler because it supports a pure small model, for which the code generator totally abstains from manipulating the 8086 segment registers. Other C compilers occasionally save and restore the segment registers to memory when generating small model code.

---

[2]The 8086 segment registers are CS (code segment), DS (data segment), SS (stack segment) and ES (extra segment).

As it happens there are other benefits to using the Clarion TopSpeed C compiler:

- in the EPOC environment, we have found that the code generated by the TopSpeed C compiler is typically 20% more compact than that produced by Turbo C or Microsoft C
- it supports register-based (rather than just stack-based) calling conventions which contribute to its compact code generation and also reduce execution times
- we have used its capability to define custom calling conventions (via `#pragma` statements) to interface[3] efficiently with the software interrupts of the system services (in some cases removing entirely the need for interfacing code)

Because the functions in the PLIB library for TopSpeed C use custom calling conventions, **the use of C prototypes is mandatory**.

System services

Just like MSDOS and the BIOS on a PC, the system services are accessed using 8086 software interrupts where the parameters are passed in 8086 processor registers.

The software interrupt interface to the system services are described in the *EPOC O/S System Services* reference manual. You would refer to this manual if you were writing in 8086 assembler or accessing a system service from OPL. You don't need the *EPOC O/S System Services* reference manual when programming in C but knowing about them can be useful when debugging applications.

Given the quality of the TopSpeed C compiler, it is hard to justify writing anything in 8086 assembler except device drivers. Even then, you only need to write an interfacing layer (between the I/O system and the device driver) in assembler - the rest can be written in C.

Because there is a very close correspondence between PLIB and the system services, the vast majority of the functions in the PLIB library are "thin" code shells around one or more software interrupts to system services. Many PLIB functions just call the appropriate interrupt and convert return values.

For example, if your program calls `p_bcpy` (the first function to be described in this manual) and you use the debugger to disassemble from address `p_bcpy` to see what was brought in from the PLIB library, you would get:

```
CD A1       BufferCopy
8B C7       MOV AX,DI
03 C1       ADD AX,CX
C3          RET
```

which adds up to 7 bytes. You don't need to know much about 8086 assembly language to know that 7 bytes is good value for any function. All the work is done by code that is in the ROM and which, in this case, is called by the `BufferCopy` or `INT A1` instruction. In this case, the calling convention for `p_bcpy` has been set to match that of the `BufferCopy` interrupt but some code is required following `BufferCopy` to return the correct value.

For some functions there is no shell at all and, in this case, the C compiler converts a PLIB function call into an in-line software interrupt.

PLIB header files

To get the constants, typedefs and prototypes for using PLIB, the simplest thing to do is to insert:

```
#include <plib.h>
```

at the beginning of your C source file.

The *plib.h* header file collects a default set of header files that are sufficient for the functions described in this manual.

For most source files, *plib.h* will include more than you actually need. Once you are familiar with PLIB and if you can be bothered, you may wish to browse around the header files in the *\sibosdk\include* directory to work out which files you need to include in a particular source file.

In all the structs in the PLIB headers, we have been careful to organise the members so that 16-bit (or wider) variables are on even address boundaries - since the 8086 processor can fetch a 16-bit word in a single cycle rather than two if the word is at an even address.

---

[3]The code, written in 8086 assembler, that provides a C function interface to a ROM-based service is sometimes called a *C shell*.

p_std.h

Because Psion has had ten years of working with C using tens of different C compilers to develop for tens of different target computers, we have of necessity developed a fairly defensive approach to our C sources. Rather than using the C variable types and declarations directly, we define our own to give us the opportunity of re-defining their meaning, depending on the compiler. All these definitions are in *p_std.h* that is included first by *plib.h*.

The following extract from the Clarion version of *p_std.h* is for declaring functions and data:

```
#define GLREF_D extern
#define GLDEF_D
#define LOCAL_D static
#define GLREF_C extern
#define LOCAL_C static
#define GLDEF_C
```

where the `_C` and `_D` refer to code and data respectively and where:

`LOCAL_C` `LOCAL_D`	are used to declare local functions and local static variables
`GLDEF_C` `GLDEF_D`	are used to declare global functions and global variables
`GLREF_C` `GLREF_D`	are used to declare function prototypes and external global variables

The following extract from *p_std.h* declares operand types:

```
#define VOID void

typedef int INT,HANDLE;
typedef unsigned int UINT;
typedef char BYTE;
typedef unsigned char UBYTE;
typedef short int WORD;
typedef unsigned short int UWORD;
typedef long int LONG;
typedef unsigned long int ULONG;
typedef double DOUBLE;
typedef char TEXT;
```

where the meanings of `BYTE`, `UBYTE`, `WORD`, `UWORD`, `INT`, `UINT`, `LONG`, `ULONG` and `DOUBLE` are as suggested by their names (and where a leading `U` means `unsigned`).

By convention, we favour the signed variant in cases where it does not matter whether the signed or the unsigned variant is used.

The `TEXT` type is used to indicate character data as in, for example:

```
TEXT *str;
```

where `str` points to a character string.

The `HANDLE` typedef is used to refer to an instance of something - such as a process or a memory segment. In many cases, a `HANDLE` is actually the offset into the operating system's data space.

This manual uses the above declarations in function descriptions and examples so you do need to know about them to be able to understand this manual.

However, this does not mean that you have to use our declarations. For example, using our declarations, you can write:

```
#include <plib.h>

GLDEF_C INT main(VOID)
    {
    p_printf("Hello world");
    p_getch();
    return(0);
    }
```

or, not using our declarations, you can write:

```
#include <plib.h>

int main(void)
    {
    p_printf("Hello world");
    p_getch();
    return(0);
    }
```

It is entirely up to you.

Calling conventions

The content of this section is quite technical. Provided you:

- include *plib.h*
- declare local functions before calling them
- use prototypes when calling your own global functions

you don't need to be particularly aware of the calling convention used and you don't have to understand this section (although you may feel more comfortable if you do).

The only exception is for functions which take as a parameter the address of (and subsequently call) a second function. In such a case you must declare an explicit calling convention for the second function. In PLIB this occurs when using `p_enter`, used to handle errors, or when using an object-oriented programming message-sending function such as `p_send`. In these two cases the descriptions of the relevant functions include full guidance.

For a greater understanding of calling conventions and the `#pragma call` declaration, refer to the *TopSpeed* documentation.

With the TopSpeed C compiler you can change the calling convention using:

`#pragma save`	to save the current calling convention
`#pragma call`	to set a new current calling convention as defined by parameters that follow `call`
`#pragma restore`	to restore a previously saved calling convention

You can also declare functions to use a stack-based calling convention using `CDECL`.

The C header files containing the prototypes for the PLIB functions (and automatically included when you include *plib.h*) also contain `#pragma` statements that declare calling conventions on a function by function basis.

For example, the prototype for `p_bcpy` is effectively:

```
#pragma save
#pragma call(reg_param =>(di,si,cx),reg_saved =>(bx,cx,dx,si,di,ds,st1,st2))
GLREF_C UBYTE *p_bcpy(VOID *,VOID *,UINT);
#pragma restore
```

where the three parameters to `p_bcpy` are passed in the registers DI SI and CX as required by the `BufferCopy` interrupt (described in the *EPOC O/S System Services* reference manual).

When defining a sequence of prototypes, you only need to bracket the sequence with `#pragma save` and `#pragma restore` - not each individual prototype.

Prototypes for PLIB functions that take a variable number of parameters must use a stack-based calling convention and are declared using `CDECL` as in, for example:

```
GLREF_C INT CDECL p_iow(VOID *,INT,...);
```

In many cases (where there is a limit to the number of parameters), the same function is also offered in fixed parameter versions as in, for example:

```
INT p_iow2(VOID *pcb, UINT func);
INT p_iow3(VOID *pcb, UINT func, VOID *a1);
INT p_iow4(VOID *pcb, UINT func, VOID *a1, VOID *a2);
```

where you can use the appropriate fixed parameter variant to take advantage of a more efficient register calling convention.

When calling your own functions, you don't need to worry about setting a calling convention since the default calling convention will apply. The default (register) calling convention is:

```
#pragma call(reg_saved =>(ax,bx,cx,dx,di,si,ds,st1,st2),reg_param =>(ax,bx,cx,dx),c_conv=>off)
```

When the compiler comes across a function that is declared as taking a variable number of arguments, such as:

```
LOCAL_C VOID PrintToLog(TEXT *str, ...)
```

it ignores the current calling convention and uses a stack-based calling convention.

Small programs

Because the body of nearly all PLIB functions is provided by code in the ROM, C programs built for EPOC are typically significantly smaller than, say, the standard C library on a PC.

The difference is at its most extreme when there is such a small amount of program-specific code that the size of the program is dominated by the code that is brought in from the library. For example, the following program:

```
#include <plib.h>

GLDEF_C INT main(VOID)
    {
    p_printf("Hello world");
    p_getch();
    return(0);
    }
```

when compiled and linked for EPOC produces a program that contains fewer than 500 bytes of code. Depending on the compiler, upwards of 10K is typical on a PC.

The amount of memory required to run an EPOC program (sometimes called the "working set") typically ranges from 10K to 100K bytes. For example, when using the Spreadsheet on the MC400 (a large program by any standard), you can load about 70K of code and open and manipulate a 10K spreadsheet with less than 100K of free system memory.

In practice, it is quite possible to take advantage of the multi-tasking and run several programs at the same time and especially (given that the code is only loaded once) to run more than one process of the same program.

The PLIB C startup modules

When producing an executable using the linker, a C startup module is automatically linked in before the program-specific modules and the libraries.

A number of C startup modules is supplied with PLIB where each module sets set up a different stack size (typically ranging from 2K to 8K). Except for the different stack size, the different PLIB startup modules are identical - they all declare the same basic structure for the process data segment - as described in the chapter *Memory Allocation*.

The code in the PLIB startup modules is minimal - it just connects to the file server (using a `FilConnect` interrupt) before jumping to `main` (most applications need the services of the file server and the overhead to connecting to the file server is modest).

Note that the PLIB startup modules do not set up the standard `argv`, `argc` parameters to `main` because PLIB is not particularly designed for command line user interfaces (although there is a mechanism for passing parameters when starting a process - see the chapter *Processes and Inter-Process Messaging*).

Not all the stack that is declared in the PLIB startup module may be used by the program and you should subtract:

`0x100`	for any program

0x300	for programs that use the floating point emulator (as described in the *Floating Point* chapter)

Related reference manuals

The ROM contains more system code than is directly accessed by the functions described in this manual. In addition, a standard C library (CLIB) is provided.

TopSpeed C library reference

CLIB is a version of the TopSpeed C library for the EPOC operating system. The functions in CLIB are described in the *TopSpeed C Library Reference* manual. Additional notes, including a list of the TopSpeed C library functions that are not implemented, may be found in *\sibosdk\doc\clib.doc*.

The EPOC version of the TopSpeed C library supports the ANSI functions and most of the portable functions that are commonly supported by MSDOS C libraries such as Microsoft C and Borland's Turbo C. The less portable functions such as those that access the BIOS and graphics functions are not included.

The benefits of using CLIB are:

- portability (existing C programs may easily be converted)
- less to learn for programmers already familiar with standard C libraries

Although the EPOC system services (and hence PLIB) has comprehensive support for floating point operations, it does not provide this support in a way that supports the floating point C as generated by the TopSpeed C compiler. This is described more fully in the chapter *Floating Point* in this manual.

As you might expect, using CLIB in place of PLIB makes less efficient use of the SIBO architecture. In particular:

- many of the EPOC system services are not available from CLIB (eg asynchronous I/O, inter-process messaging, the window server graphics functions)
- executables are larger and the process takes a larger data segment

The executables are larger because, although CLIB uses the ROM-based system services wherever possible and fares better than the PC library, it is still a much "thicker" library than PLIB. The data segments also tend to be larger because the various CLIB subsystems typically require large static buffers and tables.

For example, the following CLIB program:

```
#include <stdio.h>

int main(void)
    {
    printf("Hello world");
    getchar();
    return(0);
    }
```

when compiled and linked for EPOC produces a program that contains 6K bytes of code (as compared with 0.5K for the equivalent PLIB program).

Unless you are using the in-built user interface object dynamic libraries (accessed using object-oriented programming) described below, you can freely mix PLIB calls with CLIB. We expect most experienced C programmers to use CLIB and regard PLIB and WLIB (the window server library, described below) as they would regard non-portable components of any C library.

It is worth converting completely to PLIB and WLIB when the desirability of making efficient use of memory outweighs the benefits of portability and familiarity.

Window server reference

The window server is a system process that provides shared access to the screen and keyboard (and also a pointing device, if present).

The PLIB library contains only primitive console functions to input typed lines (with simple backspace editing) and to output lines of mono-spaced characters. The CON: device driver provides row and column positioning and printing of mono-spaced characters.

Although the PLIB console functions and the CON: device driver ultimately call on the window server for both user input and screen drawing, the window server is capable of far more than can be accessed via these interfaces. In particular, the window server can be used to implement graphical user interfaces and to display bitmap images such as maps and diagrams. The WLIB library contains a set of C functions that can access all the services of the window server. These functions are described in the *Window Server Reference* manual.

The window server supports the following features:

- a hierarchical system of overlapping windows where all drawing is clipped to visible areas
- redraw events informing the client of areas of windows that need to be redrawn, with redrawing clipped to the invalid areas
- multi-font (proportional and mono-spaced) pixel addressable text drawing in a variety of text modes and styles
- fast bitmap operations
- drawing of lines, boxes and pattern-filled areas in a variety of modes
- optional double drawing to a background bitmap as well as the window such that the window server automatically redraws windows as necessary

Like PLIB, the WLIB library is a library of thin C shell functions that contain software interrupts to ROM-based code.

I/O devices reference

This *I/O System* chapter in this manual describes the EPOC I/O system in general and the PLIB C functions that are used to access I/O devices.

A device driver may be built into the ROM or it may be loaded from an external source (such as an SSD).

To use a particular device you need to read a description of the device driver.

The files device driver and the asynchronous timer device driver are described in this manual - in the chapters *Files* and *Time, Timers and Dates* respectively. All other device drivers are described in the *I/O Devices Reference* manual.

The *I/O Devices Reference* manual describes device drivers that have been written by Psion - many of which are commonly supplied in the ROM. The device drivers described include the following:

- Parallel port (PAR:)
- Serial port (TTY:)
- Console device (CON:)
- Sound driver (SND:)

Descriptions of additional device drivers will be added to the *I/O Devices Reference* manual from time to time.

EPOC O/S System Services reference manual

The *EPOC O/S System Services* reference manual describes the software interrupt interface to the ROM-based system services.

Nearly all of the services are available from C using the functions in PLIB. The CLIB library also uses the system services where possible (the source for CLIB may be found in *\sibosdk\src*).

Because the *EPOC O/S System Services* reference manual is intended to be used in conjunction with the *PLIB Reference* manual, the descriptions in the *EPOC O/S System Services* reference manual are comparatively brief.

You would refer to the *EPOC O/S System Services* reference manual if you were writing in 8086 assembler or accessing a system service from OPL. In this case, you should refer to the corresponding function in the PLIB manual for a fuller description of the service (there is a list of the corresponding PLIB functions in an appendix of the *EPOC O/S System Services* reference manual). Being able to refer to the description of a software interrupt can be useful when debugging C programs.

The *EPOC O/S System Services* reference manual also contains information on:

- writing device drivers

- writing an installable file system
- hardware interfacing

Object dynamic libraries

The *Object-Oriented Programming* chapter of this manual describes a set of functions that provide run-time support for object-oriented programming where object classes are constructed by:

- using a proprietary tool to define class property structures and to declare methods
- using regular TopSpeed C to implement the declared method functions for each class

Object-oriented programming techniques are well-suited to implementation of graphic user interfaces and multi-threaded application control.

The object dynamic libraries (DYLs) contain classes that may be used and/or subclassed to construct applications with a consistent graphical user interface. The classes supplied in the libraries include the following:

- dynamic variable length arrays and large character buffers for building complex in-memory data structures
- active objects that represent a variety of event sources for controlling multi-threaded programs
- an extensive window class tree supporting such graphics user interface components as menus, dialog boxes and edit boxes

The classes that are used to build user interfaces may vary for different SIBO machines. At the time of writing, a user interface object library had not been constructed for the HC range (since there is no requirement for a consistent user interface on a machine of this type).

The object classes are organised into dynamic libraries (DYLs) in the ROM. For example, the MC400 has:

`OLIB.DYL`	containing classes that are independent of the user interface
`WIMP.DYL`	containing classes that implement the graphical user interface

The object-oriented message passing mechanism also serves as a means of calling far code (such as the code in ROM-based DYLs). Large applications may be split into multiple DYLs to reduce their working set and to overcome the 64K code segment limit.

The object classes are described in a manual per DYL. For example, the *OLIB Reference Manual* describes variable arrays and active objects.

CHAPTER 2

CHARACTERS, STRINGS AND BUFFERS

General string and buffer functions

PLIB contains the following general string and buffer functions:

p_bcpy	to copy a buffer
p_slen	returns the length of a string
p_scpy, p_scpym	to copy a string or multiple strings
p_scat, p_scatm	to concatenate a string or multiple strings
p_brep, p_srep	to fill a buffer or string with a repeated sequence
p_bswap	to swap the contents of two buffers
p_bfil	to fill a buffer with a repeated character
p_jtob	to left, right or centre align a buffer in a (normally wider) buffer with a fill character
p_crc	to generate the CRC number of a buffer

p_bcpy — Copy memory to memory

```
UBYTE *p_bcpy(VOID *target, VOID *source, UINT len);
```

Copy `len` bytes of data from `source` to `target` and return the address following the last byte written (ie `target+len`).

The data is copied correctly when `source` and `target` overlap.

For example:

```
p_bcpy(str+1,str,p_slen(str)+1);
*str='A';
```

inserts 'A' at the beginning of `str`.

p_slen — Return string length

```
UINT p_slen(TEXT *str);
```

Return the length of the zero terminated string `str`, not including the terminating zero.

p_scpy — Copy a string

```
TEXT *p_scpy(TEXT *target, TEXT *source);
```

Copy the zero terminated string `source`, producing a zero terminated string at `target` and return the address of the terminating zero of `target`.

The strings should not overlap.

For example:

```
p_scpy(buf,"hello");
```

writes "hello" to buf.

p_scpym — Copy multiple strings

```
TEXT *p_scpym(TEXT *target, ...);
```

Copy and concatenate a list of zero terminated strings to target creating a zero terminated string at target.

Returns the address of the zero that terminates the string at target.

The first string is copied to target and the following strings are concatenated to it. The list of strings should be terminated by a NULL argument.

For example:

```
p_scpym(buf,"The cat"," jumped",NULL);
```

writes "The cat jumped" to buf.

p_scat — Concatenate two strings

```
TEXT *p_scat(TEXT *lstr, TEXT *rstr);
```

Concatenate the zero terminated string rstr to the zero terminated string lstr and return the address of the zero that terminates the new string at lstr.

For example:

```
p_scpy(buf,"hello");
p_scat(buf," fred");
```

writes "hello fred" to buf.

p_scatm — Concatenate many strings

```
TEXT *p_scatm(TEXT *lstr, ...);
```

Concatenate a list of zero terminated strings to the zero terminated string lstr.

Returns the address of the zero that terminates the new string at lstr.

The list of strings should be terminated by a NULL argument.

For example:

```
p_scpy(buf,"The");
p_scatm(buf," cat"," jumped",NULL);
```

writes "The cat jumped" to buf.

p_brep — Replicate a buffer

```
UBYTE *p_brep(VOID *buf, INT buf_len, VOID *pattern, INT pat_len);
```

Replicate pattern as many times as necessary to exactly fill the buffer buf of length buf_len and return the address of the byte following the last byte written (ie buf+buf_len).

If buf_len<pat_len then only buf_len bytes of pattern are copied to buf (the same principle applies if buf_len is not a multiple of pat_len).

For example:

```
*p_brep(buf,7,"ab",2)=0;
```

writes "abababa" to buf.

p_srep — Replicate a string

```
TEXT *p_srep(TEXT *buf, INT buf_len, TEXT *pattern);
```

Replicate the contents of the zero terminated string pattern as many times as necessary to exactly fill the buffer buf of length buf_len and return the address of the byte following the last byte written (ie buf+buf_len).

The zero terminator from pattern is excluded in the copy. If buf_len<p_slen(pattern) then only buf_len bytes of pattern are copied to buf (the same principle applies if buf_len is not a multiple of p_slen(pattern)).

For example:

```
*p_srep(buf,7,"ab")=0;
```

writes "abababa" to buf.

p_bswap — Swap two buffers

```
VOID p_bswap(VOID *buf1, VOID *buf2, INT len);
```

Swap len bytes from the buffers buf1 and buf2.

For example:

```
p_scpy(buf1,"xxx");
p_scpy(buf2,"yyy");
p_bswap(buf1,buf2,3);
```

leaves "xxx" in buf2 and "yyy" in buf1.

p_bfil — Fill a buffer with a value

```
UBYTE *p_bfil(VOID *buf, UINT buf_len, INT fill_byte);
```

Fill the buffer buf of length buf_len with the fill byte fill_byte and return the address of the byte following the last byte written (ie buf+buf_len).

For example:

```
TEXT buf[32];

p_bfil(&buf[0],sizeof(buf),0);
```

zero fills buf.

p_jtob — Align buffer

```
TEXT *p_jtob(TEXT *tbuf, INT tlen, TEXT *sbuf, INT slen, INT type, INT fill);
```

Align sbuf, slen in tbuf, tlen according to the alignment type type and fill any excess space in tbuf with character code fill where type is one of:

P_JLEFT	to align left
P_JRIGHT	to align right
P_JCENTRE	to align centred

If slen is greater than tlen, the first tlen bytes from sbuf is copied to tbuf.

If tlen is -1, p_jtob just copies slen bytes from sbuf to tbuf.

The function returns the address of the byte following the last byte written to tbuf. That is, tbuf+tlen if tlen is not -1 or tbuf+slen if tlen is -1.

For example:

```
TEXT buf[32];

*p_jtob(&buf[0],8,"FRED",4,P_JCENTRE,'*')=0;
```

writes the zero terminated string "**FRED**" to buf.

p_crc — Generate the CRC number

```
VOID p_crc(UWORD *pcrc, UBYTE *buf, UINT len);
```

Incrementally generate the CRC polynomial checksum *pcrc (X power 16 + X power 12 + X power 5 + 1, as recommended by CCITT) of the len bytes at buf.

If the checksum is being started *pcrc should be initialised to zero. Subsequent calls incrementally modify *pcrc.

Character classification and conversion

The character classification and conversion functions in PLIB are based on four EPOC system tables (which are normally built into the ROM):

- a table that classifies characters
- a table that defines how characters are folded
- a table for converting to upper case
- a table for converting to lower case

The interpretation of character codes depends upon the character fonts that are built into the system. Character fonts on SIBO machines are normally compatible with the IBM code page 850 character set (a superset of the ASCII character set) which is widely supported by PCs and printers. The character classification and conversion tables may be changed to accommodate different character sets.

The two tables that convert to upper and lower case may also be changed to accommodate the requirements of different languages (without having to rebuild applications). For example, the table for converting to upper case may be different on a French machine from that on a German machine (because accented characters may be handled differently).

All the functions described in this chapter are provided as real functions rather than C macros (as is normal practice in standard C libraries) so that their effect can be dependent on the built-in system tables.

All the tables apply to character codes in the range 0 to 255 inclusive. The character classification table contains 256 8-bit bit masks. The other three tables contain 256 byte mapping tables where each byte contains the code of the corresponding converted character.

The character classification table

The character classification table is used by the 11 classification functions: p_isalnum, p_isalpha, p_iscntrl, p_isdigit, p_isgraph, p_islower, p_isprint, p_ispunct, p_isspace, p_isupper, p_isxdigit. Remove the leading p_ and the functions correspond in name and purpose to the macros normally supplied with standard C libraries.

Each byte in the character classification table contains a mask of 8 bits as follows:

```
#define _U      0x1     /* uppercase, p_isupper */
#define _L      0x2     /* lowercase, p_islower */
#define _D      0x4     /* digit, p_isdigit */
#define _S      0x8     /* whitespace, p_isspace */
#define _P      0x10    /* punctuation, p_ispunct */
#define _C      0x20    /* control, p_iscntrl */
#define _X      0x40    /* hex digit, p_isxdigit */
#define _B      0x80    /* blank, used by p_isprint */
```

The _U and _L bits control p_isupper and p_islower respectively - they should not both be set. If either is set, p_isalpha, p_isalnum, p_isgraph and p_isprint return TRUE.

The _D bit is set for characters '0' to '9' and controls p_isdigit. If it is set, p_isalnum, p_isgraph and p_isprint return TRUE.

The _S bit is set for character codes 0x9 to 0xD inclusive and 0x20. It controls p_isspace only.

The _P bit is set for punctuation characters and controls p_ispunct. If it is set, p_isgraph and p_isprint return TRUE.

The _C bit is set for character codes 0x0 to 0x1F inclusive and 0x7f. It controls p_iscntrl only.

The _X bit is set for characters '0' to '9', 'A' to 'F', and 'a' to 'f'. It controls p_isxdigit only.

The _B bit is set only for character code 0x20. If it is set, p_isprint returns TRUE.

The fold table

Folding means the removal of differences between characters that the author of the fold table deems unimportant for the purposes of inexact or "case insensitive" matching. As well as ignoring differences of case, folding ignores any accent on a character. By convention, folding converts lower case characters into upper case and removes any accent. The folding functions should not be used in place of p_toupper to convert to upper case.

The folding table is used by the primitive folding functions p_tofold (which folds a single character) and by p_scpyf and p_sconf (which fold strings), described in this chapter. These primitive functions are in turn used by the PLIB functions that perform case insensitive searching (eg p_sloci) and lexical comparison (eg p_scmpi).

Folding strings before performing case sensitive operations has the same effect (but may be faster) as performing the case insensitive operations (such as p_scmpi).

Examples of situations where folding is used are file names, program language keywords, command parameters and case insensitive searching.

Strictly speaking, folding should not be used for sorting human readable[1] lists on a textual key (where the folding function is applied to the sort key). To do the job properly requires an additional table to define an independent collating sequence for each language and additional language specific logic to ignore certain characters for the purposes of comparison and to expand some characters to two characters for the purposes of comparison. See the IBM publication *Software without Frontiers (Second Edition)*, pages 1-18 to 1-19 for further details and examples. However, some applications do use the folding functions to sort on a textual key because, removing accents does group accented characters correctly (although the order within the group is not correct) and it is much simpler than doing it properly.

The case conversion tables

The tables that convert to upper and lower case are used only by the functions p_scap, p_toupper and p_tolower.

These functions are provided as a service for application programs and, in contrast to the folding functions, are not otherwise used by the system.

Examples of their legitimate use are commands in a word processor that force selected text to upper and lower case and case conversion functions in OPL and the Spreadsheet.

p_isupper — Test for upper case character

```
INT p_isupper(INT c);
```

Returns TRUE if c modulo 256 is an uppercase alphabetic character, accented or otherwise.

p_islower — Test for lower case character

```
INT p_islower(INT c);
```

Returns TRUE if c modulo 256 is a lower case alphabetic character, accented or otherwise.

p_isalpha — Test for alphabetic character

```
INT p_isalpha(INT c);
```

Returns TRUE if c modulo 256 is an alphabetic character of either case. Equivalent to (p_isupper(c) || p_islower(c)).

[1] It is appropriate to use folding when the order is not seen by a human - as in a symbol table.

p_isdigit — Test for numeric digit

```
INT p_isdigit(INT c);
```

Returns TRUE if c modulo 256 is a decimal digit, that is, 0-9.

p_isalnum — Test for alphanumeric character

```
INT p_isalnum(INT c);
```

Returns TRUE if c modulo 256 is an alphanumeric character. Equivalent to `(p_isalpha(c) || p_isdigit(c))`.

p_isxdigit — Test for hexadecimal digit

```
INT p_isxdigit(INT c);
```

Returns TRUE if c modulo 256 is a hex digit. That is, 0-9, A-F or a-f.

p_isspace — Test for whitespace character

```
INT p_isspace(INT c);
```

Returns TRUE if c modulo 256 is a whitespace character, where a whitespace character is a space or space-like control character (HT, NL, VT, FF or CR).

Used by p_skipwh and p_skipch.

p_iscntrl — Test for control character

```
INT p_iscntrl(INT c);
```

Returns TRUE if c modulo 256 is a control character. Control character codes are 0-31 and 127.

p_ispunct — Test for punctuation character

```
INT p_ispunct(INT c);
```

Returns TRUE if c modulo 256 is a punctuation character. A character is a punctuation character if it is a printable graphic that is not alphanumeric or space.

p_isgraph — Test for printable graphic character

```
INT p_isgraph(INT c);
```

Returns TRUE if c modulo 256 is a printable graphic. That is, including p_isalnum or p_ispunct. To put it another way, not p_iscntrl or p_isspace.

p_isprint — Test for printable character

```
INT p_isprint(INT c);
```

Returns TRUE if c modulo 256 is a printable character. This is the same as p_isgraph except that it includes space.

p_skipwh — Skip whitespace characters

```
TEXT *p_skipwh(TEXT *str);
```

Skip over any leading white space characters in the passed string, returning the address of the first non white space character (ie p_isspace returns FALSE) or zero terminator.

For example:

```
str=p_skipwh("   abcd");
```

sets str to point to the character 'a' in the string.

p_skipch — Skip non-whitespace characters

```
TEXT *p_skipch(TEXT *str);
```

Scan the string until either a white space character (ie p_isspace returns TRUE) or a zero terminator is detected.

For example:

```
str=p_skipch("abcd   ef");
```

sets str to point to the character after 'd' in the string.

p_tofold — Fold a character

```
INT p_tofold(INT c);
```

Returns the folded character using the built-in fold table.

Returns the parameter unchanged if c is greater than 255. Fold tables are normally designed such that c is unchanged if p_isalpha(c) is FALSE.

p_scpyf — Copy string with fold

```
TEXT *p_scpyf(TEXT *target, TEXT *source);
```

Places a folded copy (using p_tofold) of the zero terminated string at source into the buffer at target, returning the address of the zero that terminates the string at target (ie target+p_slen(source)).

Behaves and returns like p_scpy except that characters are folded on the way.

Example

```
TEXT buf[32];

p_scpyf(&buf[0],"hello");
```

copies "HELLO" to buf.

p_sconf — Fold string

```
VOID p_sconf(TEXT *str);
```

Fold the characters (using p_tofold) in the zero terminated string *str.

p_toupper — Convert character to upper case

```
INT p_toupper(INT c);
```

Returns the character converted to upper case as defined by the built-in upper case table.

You can't assume that the returned character is less than 128 or whether it has an accent. The function is intended only for use by applications to convert text to upper case (as in, for example, the Upper command in the MC GI Text Processor). It should not be used for case insensitive matching or lexical comparison - see p_tofold.

Returns the parameter unchanged if c is less than 0 or greater than 255. Upper case tables are normally designed such that c is unchanged if p_isalpha(c) is FALSE.

p_tolower — Convert character to lower case

```
INT p_tolower(INT c);
```

Returns the character converted to lower case as defined by the built-in lower case table.

You can't assume that the returned character is less than 128 or whether it has an accent. The function is intended only for use by applications to convert text to lower case (as in, for example, the Lower command in the MC GI Text Processor). It should not be used for case insensitive matching or lexical comparison - see p_tofold.

Returns the parameter unchanged if c is less than 0 or greater than 255. Lower case tables are normally designed such that c is unchanged if p_isalpha(c) is FALSE.

p_scap — Capitalise string

```
VOID p_scap(TEXT *str);
```

This function is only available in EPOC version 2.14 or later.

Capitalises the zero terminated string *str.

Applies p_toupper to the first character, and p_tolower to all subsequent characters.

Example

```
TEXT buf[32];

p_scpy(&buf[0],"hello WORLD");
p_scap(&buf[0];
```

leaves "Hello world" in buf.

String comparison

For comparing strings, PLIB contains:

p_bcmp, p_scmp	for comparing two strings
p_bcmpi, p_scmpi	for case independent comparison

String comparison is based on comparing corresponding bytes in the strings. The functions return zero if the two strings are equal (that is, they have the same length and their corresponding bytes match). If the strings are not equal, the functions return a signed non-zero value derived from the first two bytes to disagree. The sign of the non-zero returns indicates which of the two strings is the greater and can be used to sort strings.

With case independent comparison, the characters in both strings are "folded" using p_tofold before comparing them in the same way as for case dependent comparison.

In many applications, one match string is repeatedly compared in a case independent way against a symbol table of strings. If you are prepared to lose the case (and accent) of the original input string, it is more efficient to fold the characters in both the symbol table (when building it) and the match strings and to use the much faster normal (ie case independent) string comparison functions.

p_bcmp — Compare two buffers

```
INT p_bcmp(VOID *lbuf, INT lbuf_len, VOID *rbuf, INT rbuf_len);
```

Compare two buffers by comparing corresponding unsigned bytes, returning lbuf-rbuf, that is:

If lbuf<rbuf then return is less than 0
If lbuf>rbuf then return is greater than 0
If lbuf==rbuf then return equals 0

The result of the comparison is based on the difference of the first two unsigned bytes to disagree. The strings are equal if they have the same length and content. Where two strings have different lengths and the shorter string matches the first part of the longer string, the shorter string is considered to be less than the longer string.

For example:

p_bcmp("abc",3,"abcd",4) returns less than 0
p_bcmp("abcd",4,"abc",3) returns greater than 0
p_bcmp("abc",3,"abc",3) returns 0.

p_scmp — Compare two strings

```
INT p_scmp(TEXT *lstr, TEXT *rstr);
```

Compare two zero terminated strings by comparing corresponding characters, returning `lstr-rstr`, that is:

If `lstr<rstr` then return is less than 0
If `lstr>rstr` then return is greater than 0
If `lstr==rstr` then return equals 0

The result of the comparison is based on the difference between the first two characters to disagree. The strings are equal if they have the same length and content.

For example:

`p_scmp("abc","abcd")` returns less than 0
`p_scmp("abcd","abc")` returns greater than 0
`p_scmp("abc","abc")` returns 0

p_bcmpi — Case independent buffer compare

```
INT p_bcmpi(TEXT *lbuf, INT lbuf_len, TEXT *rbuf, INT rbuf_len);
```

Performs a case independent comparison of the two buffers by effectively folding the characters in both buffers before comparing them using `p_bcmp`. Returns as for `p_bcmp`, described above.

For example:

`p_bcmpi("abc",3,"abcd",4)` returns less than 0
`p_bcmpi("abcd",4,"abc",3)` returns greater than 0
`p_bcmpi("ABC",3,"abc",3)` returns 0

p_scmpi — Case independent string compare

```
INT p_scmpi(TEXT *lstr, TEXT *rstr);
```

Performs a case independent comparison of the two zero terminated strings `lstr` and `rstr` by effectively folding the characters in each string before comparing them using `p_scmp`. Returns as for `p_scmp`, described above.

For example:

`p_scmpi("abc","abcd")` returns less than 0
`p_scmpi("abcd","abc")` returns greater than 0
`p_scmpi("ABC","abc")` returns 0

String searching

The PLIB string searching functions are:

`p_bloc, p_sloc, p_bloci, p_sloci, p_slocr, p_slocri`	to search for a character
`p_bsub, p_ssub, p_bsubi, p_ssubi`	to search for a sequence of characters
`p_bmatch, p_smatch, p_bmatchi, p_smatchi`	to search for a sequence of characters that matches a wildcard specification

p_bloc — Locate byte in buffer

```
INT p_bloc(VOID *buf, INT buf_len, INT ch);
```

Locate the byte `ch` in the buffer at `buf` of length `buf_len` returning the index of the first matching byte or -1 if `ch` is not in the buffer.

For example:

```
p_bloc("abcde",5,'f') returns -1
p_bloc("abcde",5,'a') returns 0
p_bloc("abcde",5,'c') returns 2
```

p_sloc — Locate character in string

```
INT p_sloc(TEXT *str, INT ch);
```

Locate the first occurrence of the character `ch` in the zero terminated string `str`, returning the index of the first matching character or -1 if `ch` is not in `str`.

For example:

```
p_sloc("abcde",'f') returns -1
p_sloc("abcde",'a') returns 0
p_sloc("abcde",'c') returns 2
```

p_bloci — Case independent locate character in buffer

```
INT p_bloci(TEXT *buf, INT buf_len, INT ch);
```

Perform a case independent locate of `ch` in the buffer `buf` by effectively folding `ch` and the characters in `buf` before using `p_bloc` to locate the folded character. Returns as for `p_bloc`, described above.

For example:

```
p_bloci("abcde",5,'f') returns -1
p_bloci("abcde",5,'A') returns 0
p_bloci("abcde",5,'c') returns 2
```

p_sloci — Case independent locate character in string

```
INT p_sloci(TEXT *str, INT ch);
```

Perform a case independent locate of `ch` in the zero terminated string `str` by effectively folding `ch` and the characters in `str` before using `p_sloc` to locate the folded character. Returns as for `p_sloc`, described above.

For example:

```
p_sloci("abcde",'f') returns -1
p_sloci("abcde",'a') returns 0
p_sloci("abcde",'C') returns 2
```

p_slocr — Locate last matching character in a string

```
INT p_slocr(TEXT *str, INT ch);
```

Locate the last occurrence of the character `ch` in the zero terminated string `str`, returning the index of the matching character or -1 if `ch` is not in `str`.

For example:

```
p_slocr("abcabc",'A') returns -1
p_slocr("abcabc",'a') returns 3
```

p_slocri — Locate last matching folded character in a string

```
INT p_slocri(TEXT *str, INT ch);
```

Locate the last occurrence of `ch` in the zero terminated string `str` by effectively folding `ch` and the characters in `str` before using `p_slocr` to locate the folded character. Returns as for `p_slocr`, described above.

For example:

```
p_slocri("abcde",'f') returns -1
p_slocri("abcabc",'A') returns 3
```

p_bsub — Locate sub-buffer in buffer

```
INT p_bsub(VOID *buf, INT buf_len, VOID *sbuf, INT sbuf_len);
```

Locate sub-buffer sbuf,sbuf_len in buf,buf_len returning the index of the start of sbuf in buf or -1 if sbuf does not exist in buf (which must be the case if buf_len<sbuf_len). Returns zero if sbuf_len is zero.

For example:

p_bsub("abcde",5,"fg",2) returns -1
p_bsub("abcd",4,"ab",2) returns 0
p_bsub("abcde",5,"cd",2) returns 2

p_ssub — Locate substring in string

```
INT p_ssub(TEXT *str, TEXT *substr);
```

Search the zero terminated string str for the first occurrence of the zero terminated string substr and return the index of substr in str or -1 if substr does not exist in str. Returns zero if the length of str is zero.

For example:

p_ssub("abcde","fg") returns -1
p_ssub("abcd","ab") returns 0
p_ssub("abcde","cd") returns 2

p_bsubi — Case independent locate sub-buffer in buffer

```
INT p_bsubi(TEXT *buf, INT buf_len, TEXT *sbuf, INT sbuf_len);
```

Perform a case independent locate of sbuf,sbuf_len in buf,buf_len by effectively folding the characters in both buffers before using p_bsub to locate the sub buffer. Returns as for p_bsub, described above.

For example:

p_bsubi("abcde",5,"fg",2) returns -1
p_bsubi("abcd",4,"ab",2) returns 0
p_bsubi("abcde",5,"CD",2) returns 2

p_ssubi — Case independent locate substring in string

```
INT p_ssubi(TEXT *str, TEXT *substr);
```

Perform a case independent locate of the zero terminated string substr in the zero terminated string str by effectively folding the characters in both strings before using p_ssub to locate the sub string. Returns as for p_ssub, described above.

For example:

p_ssubi("abcde","fg") returns -1
p_ssubi("abcd","ab") returns 0
p_ssubi("abcde","CD") returns 2

p_bmatch — Pattern match a buffer

```
INT p_bmatch(TEXT *buf, INT blen, TEXT *mbuf, INT mlen);
```

Compare the match pattern in mbuf,mlen against buf,blen and return TRUE if they match.

The match buffer mbuf may contain the wildcard characters '*' and '?' where '*' matches zero or more consecutive occurrences of any character and '?' matches a single occurrence of any character.

Note that p_bmatch returns TRUE only if the match pattern in mbuf matches the whole of buf. If you want to test for the existence of a pattern within a string, you must have a '*' at the beginning and end of mbuf. See p_smatch for examples.

p_bmatchi — Pattern match a buffer, case independent

```
INT p_bmatchi(TEXT *buf, INT blen, TEXT *mbuf, INT mlen);
```

Case independent version of p_bmatch. Effectively folds the characters in both buffers before using p_bmatch. Parameters and returns are as for p_bmatch, described above.

p_smatch — Pattern match a string

```
INT p_smatch(TEXT *str, TEXT *mstr);
```

Compare the match pattern in the zero terminated string mstr against the zero terminated string str and return TRUE if they match.

The match buffer mstr may contain the wildcard characters '*' and '?' where '*' matches zero or more consecutive occurrences of any character and '?' matches a single occurrence of any character.

Note that p_smatch returns TRUE only if the match pattern in mstr matches the whole of str. If you want to test for the existence of a pattern within a string, you must have a '*' at the beginning and end of mstr.

For example:

```
LOCAL_D TEXT str[]="abcdefghijklmnopqrstuvwxyz";

p_smatch(str,"*ijk*") returns TRUE
p_smatch(str,"*i?k*") returns TRUE
p_smatch(str,"ijk*") returns FALSE
p_smatch(str,"*i*mn*") returns TRUE
```

p_smatchi — Pattern match a string, case independent

```
INT p_smatchi(TEXT *str, TEXT *mstr);
```

Case independent version of p_smatch. Effectively folds the characters in both buffers before using p_smatch. Parameters and returns are as for p_smatch, described above.

CHAPTER 3

ARRAYS AND QUEUES

Arrays

p_bsrch — Binary search an array of records

```
INT p_bsrch(INT nrec, INT (*compf)(), INT *pmid, UBYTE *pmatch);
```

Use a binary search algorithm to find the record, in an array of `nrec` records, that is adjacent to, if not identical with, the record pointed to by `pmatch`. Writes the record number of the found record to `*pmid` and returns:

0	the found record is equal to `*pmatch`
<0	`*pmatch` belongs before record number `*pmid`
>0	`*pmatch` belongs after record number `*pmid`

If two or more array elements exactly match the record at pmatch, the return value will be zero and `*pmid` will contain the index of any one of the matching elements.

Each time `p_bsrch` needs to compare a record with the record at `pmatch` it calls:

```
compf(n,pmatch);
```

where `n` is the index of the record to be compared. This user-supplied comparison routine should return:

0	`*pmatch` is equal to record number `n`
<0	`*pmatch` is before record number `n`
>0	`*pmatch` is after record number `n`

As with any binary search, the array must be ordered. For `p_bsrch`, it should be ordered with respect to `compf`, the user-supplied comparison routine.

In most cases the set of records will be a fixed array, but any arrangement which allows the routine to identify a record by index (e.g. hashing) will be suitable. Likewise, `pmatch` may be any address suitable for interpretation by `compf`, since `pmatch` is not used inside `p_bsrch`, other than being passed to `compf`. This allows code using `p_bsrch` to be re-entrant.

If the whole table is created before being searched it is, in general, quicker to build the table unordered and then sort it (see `p_qsort`) than to build the table in sequence by insertion.

Example

```
LOCAL_D INT array[]={5,8,13,19,25,30,41,48,51,62,70,76,80,90,98};

LOCAL_C intcompare(INT n,INT *pmatch)
    {
    INT f,m;

    f=array[n];
    m=*pmatch;
    if (f==m)
        return(0);
    return(m>f?1:-1);
    }

VOID find(INT match)
    {
    INT nrec; /* number of INTs in the array */
    INT result;
    INT mid;
    TEXT *pstr;

    nrec=sizeof(array)/sizeof(INT);
    result=p_bsrch(nrec,(INT (*)())intcompare,&mid,&match);
    if (!result)
        p_printf("%d matches record number %d",match,mid);
    else
        {
        pstr=result<0?"before":"after";
        p_printf("%d belongs %s %d",match,pstr,array[mid]);
        }
    }
```

p_qsort — Sort an array of records

```
INT p_qsort(INT nrec, INT (*ordf)(), VOID (*excf)(), UBYTE *base)
```

Sort a set of records into ascending order, using the quicksort algorithm.

There are `nrec` records to sort. Each time the routine needs to compare two of these records it calls:

```
(*ordf)(n,m,base);
```

where `n` and `m` are the indexes of the two records to compare (starting at zero), and `base` is a parameter that may be used or ignored by the ordering function.

The ordering function, `ordf`, should return:

`0`	record number n is equal to record number m
`<0`	record number n is less than record number m
`>0`	record number n is greater than record number m

Each time the routine needs to exchange two records it calls:

```
(*excf)(n,m,base);
```

where `n` and `m` are the indexes (starting at zero) of the two records to exchange. The user-supplied exchange routine should exchange the records indexed by `n` and `m`. Again, the user-supplied exchange function is free to use or ignore `base`, but its use should be consistent between the ordering and exchange functions.

Normally, `base` will represent the address of a fixed length array, but any method of storing records may be used, as long as the records can be accessed using the indices. The value of `base` is not used inside `p_qsort`, but is passed down to both the ordering and exchange functions to enable the writing of re-entrant code that uses `p_qsort`.

The routine uses its own stack, declared locally, to avoid the need to be called recursively. This stack is `150*sizeof(INT)` in length and, as such, could cause the main stack to overflow if the routine is called from too deep inside a program.

The function `p_qsort` returns zero if successful, else a negative error.

An error return value of E_GEN_FAIL is returned if the internal stack overflows. This is, however, unlikely as an experiment to sort 50000 elements used only 80 stack elements. If this error is returned, the routine can be called again, as it is likely that the array has been rearranged enough to permit the successful completion of a second attempt.

Example

```
LOCAL_D INT array[]={98,90,80,76,70,62,51,48,41,30,25,19,13,8,5};

LOCAL_C intcompare(INT first,INT second,INT *array)
    {
    INT f,s;

    f=*(array+first);
    s=*(array+second);
    if (s==f)
    return(0);
    return(f>s?1:-1);
    }

LOCAL_C VOID intexchange(INT first,INT second,INT *array)
    {
    INT r;

    r=*(array+first);
    *(array+first)=*(array+second);
    *(array+second)=r;
    }


LOCAL_C VOID sort(VOID)
    {
    INT n; /* number of INTs in the array */

    n=sizeof(array)/sizeof(INT);
    if (p_qsort(n,intcompare,intexchange,&array[0]))
        p_panic("Too many partitions");
    }
```

Doubly linked queues

This section describes functions for inserting and deleting entries from doubly linked queues. Each entry in the queue contains a P_QUE structure, defined in *p_que.h* as:

```
typedef struct p_que
    {
    struct p_que *next; /* pointer to next item */
    struct p_que *prev; /* pointer to previous item */
    } P_QUE;
```

A special header entry consisting only of a P_QUE data structure provides a single address by which the queue may be accessed. The empty queue consists only of the P_QUE header with both next and prev pointing to itself. A queue with n entries contains n+1 P_QUE structures - one for the header and one for each entry. The queue is built such that the next pointer of the last entry points to the header and the prev pointer of the header points to the last entry such that the n+1 P_QUE structures form a doubly linked circular queue.

The following extract from *p_que.h*:

```
#define P_INITQ(q) (q)->next=(q)->prev=(q)
#define P_DECLAREQ(q) P_QUE q = {&q,&q}
#define P_ISEMPTYQ(q) ((q)==(q)->next)
```

defines 3 macros where:

P_INITQ	may be used to initialise a header that represents an empty queue
P_DECLAREQ	may be used to declare a header that represents an empty queue

P_ISEMPTYQ evaluates to TRUE if the queue header represents an empty queue

Entries are inserted into a queue using p_enque and removed using p_deque. Neither of these functions set aside memory for entries or free memory - they merely make and break the links between entries.

The following example illustrates the use of p_enque and p_deque to set up a queue of zero terminated strings that are allocated and freed from the heap (see the chapter *Memory Allocation* for a description of f_alloc and p_free).

```
#include <p_std.h>
#include <p_que.h>

typedef struct
    {
    P_QUE piq;
    TEXT name[1];
    } QUEUE_ENTRY;

LOCAL_D P_DECLAREQ(headq);

GLDEF_C VOID AddNameToEnd(TEXT *name)
    {
    QUEUE_ENTRY *p;

    p=f_alloc(p_slen(name)+sizeof(QUEUE_ENTRY));
    p_scpy(&p->name[0],name);
    p_enque(&p->piq,&headq);
    }

GLDEF_C TEXT *FirstName(VOID)
    {
    TEXT *name;

    name=&(((QUEUE_ENTRY *)headq.next)->name[0]);
    if (P_ISEMPTYQ(&headq))
        name=NULL;
    return(name);
    }

GLDEF_C VOID DeleteFirstName(VOID)
    {
    QUEUE_ENTRY *p;

    p=(QUEUE_ENTRY *)headq.next;
    p_deque(&p->piq);
    p_free(p);
    }
```

Names are allocated and added to the end of the queue using AddNameToEnd. The code that processes the items in the queue uses FirstName to get the first item in the queue (which returns NULL if the queue is empty). After processing the first name, calling DeleteFirstName removes it from the queue and frees the memory used by it.

p_enque — Add entry to queue

```
VOID p_enque(P_QUE *pNew, P_QUE *pEntry);
```

Insert entry pNew before pEntry (ie between pEntry->prev and pEntry) into the doubly linked queue that contains pEntry.

If P_QUE hdq is the queue header:

p_enque(pNew,&hdq) adds pNew to the end of the queue (since queues are circular the previous entry to the header is the entry at the end of the queue)

p_enque(pNew,hdq.next) adds pNew to the start of the queue

p_deque — Remove entry from queue

```
VOID p_deque(P_QUE *pEntry);
```

Remove queue entry `pEntry` by linking the entries on either side of `pEntry` to each other, excluding `pEntry` from the queue.

If `P_QUE hdq` is the queue header:

`p_deque(hdq.next)`	removes the first entry from the queue
`p_deque(hdq.prev)`	removes the last entry from the queue (since queues are circular the previous entry to the header is the entry at the end of the queue)

If `hdq` is empty then `p_deque(&hdq)` will have no effect. Calling `p_deque(&hdq)` of a non-empty queue is not a good idea as there will then be no way to get into the queue.

Delta queues

A delta queue builds on doubly linked queues, described in the previous section, to store entries ordered on a `LONG` key.

A delta queue consists of a `P_QUE` header and doubly linked entries, each containing a `P_DELTA` structure, defined in *p_que.h* as:

```
typedef struct
    {
    P_QUE q;
    LONG key; /* Delta key */
    } P_DELTA;
```

where (except for the first entry in the queue) `key` contains the offset relative to the previous entry (the delta). The value of an entry's key is determined by accumulating the deltas of its predecessors. For the first entry, the key is just the delta.

The EPOC operating system uses a delta queue to implement timers. Each entry represents a timer where the key is the relative time in system ticks to the expiry of that timer and the delta is then the number of system ticks after its predecessor. This design minimises the system effort to maintain the timers - on each tick the system only has to decrement the head of the queue. See the chapters *Asynchronous Requests and Semaphores* and *Time, Timers and Dates* for more about the time delta queue and timers.

A delta list is set up much as a regular queue. A header is declared and initialised using (`P_INITQ` or `P_DECLAREQ`) giving an empty queue. Entries containing a `P_DELTA` structure are added to the delta queue using `p_enqued` and are removed using `p_dequed`.

Neither function allocates or frees memory for entries - all they do is maintain the links and calculate the deltas.

p_enqued — Add entry to delta queue

```
P_DELTA *p_enqued(P_QUE *pHead, P_DELTA *pEntry, LONG key);
```

Inserts entry `pEntry` into the delta queue headed by `pHead` according to the key `key` and returns the address of the first entry in the queue.

The queue is scanned, accumulating the key from the deltas, until an entry is found with a key that is greater than `key`. The new entry `pEntry` is inserted (with an appropriate delta) before that entry and the delta of that entry is recalculated.

p_dequed — Remove entry from delta queue

```
P_DELTA *p_dequed(P_QUE *pHead, P_DELTA *pEntry);
```

Remove `pEntry` from the delta queue headed by `pHead` and return the address of the first entry or `NULL` if `p_dequed` leaves the queue empty.

The delta of any following entry is updated to keep the accumulated key of each remaining entry constant.

If `P_QUE hdq` is the queue header:

```
p_dequed(&hdq,(P_DELTA *)hdq.next);
```

removes the first entry from the queue.

CHAPTER 4

INTEGER CONVERSION AND RECTANGLE FUNCTIONS

This chapter describes functions for converting integer numbers to a textual representation (eg to print a number) and vice versa (eg to input a number).

The chapter ends with a section that describes a set of functions that operate on rectangle data structures. These are useful, for example, when organizing a screen display.

Converting integers to text

PLIB contains the following functions to convert various types of integers to a textual representation:

`p_itob`	to convert an `INT` to a signed decimal number
`p_ltob`	to convert a `LONG` to a signed decimal number
`p_gtob`	to convert a `UINT` to an unsigned number in any radix
`p_gltob`	to convert a `ULONG` to an unsigned number in any radix
`p_atob, p_atos`	for general purpose conversion and formatting of multiple arguments

The functions that handle "any radix" are typically used to handle a radix of 2 (binary), 8 (octal), 10 (decimal) or 16 (hexadecimal).

Note that it is up to the caller to ensure that there is sufficient space in the target buffers for the output of the conversion.

p_itob — Convert an INT to decimal buffer

```
UINT p_itob(TEXT *buf, INT value);
```

Write a signed decimal representation of `value` to `buf` and return the number of characters written.

If `value` is negative, a leading '-' is written.

For example:

```
buf[p_itob(&buf[0],-24)]='\0';
```

writes "-24" to `buf`.

p_ltob — Convert a LONG to decimal buffer

```
INT p_ltob(TEXT *buf, LONG value);
```

Write a signed decimal representation of the `LONG value` to `buf` and return the number of characters written.

If `value` is negative, a leading '-' is written.

For example:

```
buf[p_ltob(&buf[0],-240000L)]='\0';
```

writes "-240000" to buf.

p_gtob — Convert a UINT to buffer any radix

```
INT p_gtob(TEXT *buf, UINT value, INT radix);
```

Write an unsigned base radix representation of value to buf and return the number of characters written.

For example:

```
buf[p_gtob(buf,0xaa55,16)]='\0';
```

writes "AA55" to buf.

p_gltob — Convert a ULONG to buffer any radix

```
INT p_gltob(TEXT *buf, ULONG value, INT radix);
```

Write an unsigned base radix representation of the ULONG value to buf and return the number of characters written.

For example:

```
buf[p_gltob(buf,0xaa5577,16)]='\0';
```

writes "AA5577" to buf.

p_atob — Convert multiple arguments to buffer

```
INT p_atob(TEXT *buf, TEXT *fstr, VOID *parg);
```

Write formatted text to buf as controlled by the format zero terminated string fstr and the argument list parg and return the number of characters written.

The format string fstr contains literal text, embedded with commands for converting the arguments at parg. The embedded commands are prefixed with the '%' character (two successive '%' characters count as one literal '%'). The literal text is simply copied to buf and the % commands convert successive arguments (which may be integers, longs or strings) at parg.

An embedded command takes one of the following forms:

%<type>	for output (with no padding) of the converted data type <type> that is one of b, c, d, f, m, o, s, u, w or x (as described below). Where appropriate, the type may be widened to a long by preceding the type letter with an l or an L or by providing the type letter in upper case.
%<width><type>	for right-aligned space-filled output in width <width> where <width> is either a positive decimal number or a * to take the width as a UINT from the argument list. If more than <width> characters is generated by the conversion, the output is truncated.
%0<width><type>	for right-aligned zero-filled output in width <width>.
%<a><f><width><type>	for left, right or centre aligned output in width <width> with fill character <f> where <a> is either -, + or =. If <f> is a *, the code of the fill character is taken as a UINT from the argument list. (If you want to fill with *'s, you have to supply it through the argument list).

A common requirement is for space-filled output. It is therefore worth enumerating special cases, using the last of the above four forms of embedded command. *Note that, in all cases, there is a space (the fill character) immediately preceding* <width>.

%- <width><type>	for left-aligned space-filled output in width <width>
%+ <width><type>	for right-aligned space-filled output in width <width>
%= <width><type>	for centre-aligned space-filled output in width <width>

The <type> specifies the type of argument conversion to be performed, as follows:

- b convert the UINT to a binary text representation
- c convert the UINT to a single character corresponding to its code

d convert the INT to a signed decimal text representation

f just output fill characters (does not use up an argument)

m convert the UINT to a two byte binary *numeric* representation, with the most significant byte first (*only available in EPOC version 2.17 or later*)

o convert the UINT to an octal text representation

s copy the TEXT * zero terminated string to the output, excluding the terminating zero.

u convert the UINT to an unsigned decimal text representation

w convert the UINT to a two byte binary *numeric* representation, with the least significant byte first (*only available in EPOC version 2.17 or later*)

x convert the UINT to a hexadecimal text representation

The type may be widened to a long by preceding the type letter with an l or an L or by providing the type letter in upper case (ignored if <type> is s or f). Output for types m and w will then occupy four bytes.

This function is normally used indirectly by the more immediately useful p_atos, described below. However, p_atob is useful for constructing p_printf-like text output functions (p_printf itself is described in the chapter *I/O System*).

For example, if file is static variable that contains the channel of an opened text file, the following function behaves like p_printf.

```
GLDEF_C CDECL VOID PrintToFile(TEXT *fmt,UINT arg,...)
    {
    UINT len;
    UBYTE buf[256];

    len=p_atob(&buf[0],fmt,&arg);
    p_write(file,&buf[0],len);
    }
```

If the INT variable a contains 65:

`PrintToFile("[%b %c %d %o %u %x]",a,a,a,a,a,a)` writes [1000001 A 65 101 65 41]

`PrintToFile("[%04x]",a)` writes [0041]

`PrintToFile("[%*x]",3,a)` writes [41]

`PrintToFile("[%+$4d.00 %s]",a,"over")` writes [$$65.00 over]

`PrintToFile("[%0*s]",10,"fred")` writes [000000fred]

`PrintToFile("[%=*4x]",'*',a)` writes [*41*]

`PrintToFile("[%-**d]",'.',10,a)` writes [65........]

`PrintToFile("[%-A4f]",a)` writes [AAAA] and makes no use of the value of a.

p_atos — Convert multiple arguments to string

```
VOID p_atos(TEXT *str, TEXT *fstr, ...);
```

Convert multiple arguments to a zero terminated string at str under control of the format string fstr.

The content of fstr is described under p_atob, above.

For example, if the variable a is a UINT that contains 65:

`p_atos(str,"%b %c %d %o %u %x",a,a,a,a,a,a)` writes "1000001 A 65 101 65 41" to str

`p_atos(str,"%04x",a)` writes "0041" to str

`p_atos(str,"%*x",2,a)` writes "41" to str

Converting text to integers

PLIB contains the following functions to convert text to various types of numbers:

p_stoi	to convert a signed decimal number string to a WORD
p_stol	to convert a signed decimal number string to a LONG
p_stog	to convert an unsigned number in any radix to a UWORD
p_stogl	to convert an unsigned number in any radix to a ULONG
p_stoa	to convert multiple fields in a string to a series of arguments

p_stoi Convert a signed decimal string to a WORD

```
INT p_stoi(TEXT **pstr, WORD *pval);
```

Attempt to convert the signed decimal string at *pstr to a 16 bit number and, if a valid number is recognised, write the number to pval, update *pstr to point to the terminating character and return zero. Otherwise, neither *pstr nor *pval is written to and the function returns one of the following negative error numbers (defined in *p_gen.h*):

E_GEN_OVER	the number is too large (greater than 32767 or less than -32768)
E_GEN_FAIL	the text could not be recognised as a number

Conversion continues until a non-decimal digit is found in the string, or the number overflows. For a number to be recognised, the string must contain at least one decimal digit.

The string may be preceded by a '-' or a '+' (whitespace is significant and will terminate the conversion but leading zeros are ignored). To convert unsigned data up to 65535, use p_stog.

For example, after:

```
INT ret;
WORD val;
TEXT *ptr="-123abc";
ret=p_stoi(&ptr,&val);
```

val contains -123, ptr points to the 'a' and ret is zero.

p_stol Convert a signed decimal string to a LONG

```
INT p_stol(TEXT **pstr, LONG *pval);
```

Converts the signed decimal string at *pstr to write a 32 bit number to pval. Behaves and returns as for p_stoi above except that it can handle decimal numbers in the range -2147483648 to 2147483647 inclusive. To convert unsigned data up to 4294967295, use p_stogl

For example, after:

```
INT ret;
LONG val;
TEXT *ptr="-123456abc";
ret=p_stol(&ptr,&val);
```

val contains -123456, ptr points to the 'a' and ret is zero.

p_stog Convert an unsigned number in any radix to a UWORD

```
INT p_stog(TEXT **pstr, UWORD *pval, INT radix);
```

Attempt to convert the unsigned number base radix (typically 2, 8, 10 or 16) at *pstr to a 16 bit number and, if a valid number is recognised, write the number to pval, update *pstr to point to the terminating character and return zero. Otherwise, neither *pstr nor *pval is written to and the function returns one of the following negative error numbers (defined in *p_gen.h*):

E_GEN_OVER	the number is too large (greater than 65535)
E_GEN_FAIL	the text could not be recognised as a number

Conversion continues until a character that is invalid for the radix is found in the string or the number overflows. For a number to be recognised, the string must contain at least one digit in the radix. Whitespace is significant and will terminate the conversion but leading zeros are ignored.

For example, given:

```
UWORD val
TEXT *ptr="f123";
```

`p_stog(&ptr,&val,10)` returns `E_GEN_FAIL`

`p_stog(&ptr,&val,16)` returns 0 and writes 0xf123 to `val`

p_stogl Convert an unsigned number in any radix to a ULONG

```
INT p_stogl(TEXT **pstr, ULONG *pval, INT radix);
```

Converts the unsigned decimal number base `radix` at `*pstr` to write a 32 bit number to `pval`. Behaves and returns as for `p_stog` above except that it can handle numbers up to 4294967295.

For example, after:

```
INT ret;
ULONG val;
TEXT *ptr="f123abz";
ret=p_stogl(&ptr,&val,16);
```

`val` contains 0xf123ab, `ptr` points to the 'z' and `ret` is zero.

p_stoa Convert a string to arguments

```
INT p_stoa(TEXT **pstr, TEXT *fstr, ...)
```

Convert multiple fields within the zero terminated string `*pstr` to a series of arguments ... as controlled by the format zero terminated string `fstr`.

Returns zero if successful and the text pointer `*pstr` is updated to point to the terminating character of the last field converted. If an error occurred, `*pstr` points to the start of the field that caused the error and one of the following negative error numbers is returned:

`E_GEN_OVER`	the result is too large
`E_GEN_FAIL`	fails to recognise a number
`E_GEN_ARG`	the supplied buffer does not contain enough items

The format string `fstr` contains literal text, embedded with commands for converting the fields in `*pstr` to the passed arguments. Any excess leading whitespace (as defined by `p_iswhite`) before a field in `*pstr` is automatically skipped.

The embedded commands in `fstr` are prefixed with the `'%'` character (two successive `'%'` characters count as one literal `'%'`). Any non whitespace literal text causes characters in the `*pstr` to be scanned until a match is made or until the end of the string is encountered (any whitespace characters in `fstr` are discarded). If a match is not found in `*pstr`, the function returns.

The general form of an embedded command is:

```
%[*][<width>][<long>]<type>

<width>:=a positive decimal number
<long>:l|L
<type>:=(B|b|C|c|D|d|N|n|O|o|Q|q|S|s|U|u|X|x)
```

where the square brackets indicate optional fields and `'|'` separates choices.

The mandatory `<type>` parameter (optionally qualified by `<long>`) indicates the data type to be converted.

If the asterisk is present, conversion is performed but the result is not be stored. There should be no corresponding value pointer in the argument list for a suppressed conversion.

The parameter `<width>` is a positive decimal number that specifies the maximum input field width when `<type>` is s or q (in either upper or lower case) - if `<type>` is other than s or q, it is ignored. Note that the

corresponding storage buffer must be large enough to hold <width> characters plus 1 more for the terminating zero (eg %19s requires a buffer of 20 bytes).

The <type> specifies the type of argument conversion to be performed, as follows:

- b convert an unsigned binary number to the UWORD
- c convert a character to its code, written to the UWORD
- d convert a signed decimal number (optionally preceded by a '-' or a '+') to the WORD
- n write the number of characters consumed so far to the UWORD
- o convert an unsigned octal number to the UWORD
- q convert quote delimited text, copying the data in between the quotes (the delimiting quotes are discarded) to produce a zero terminated string at the TEXT * buffer. Although often used to convert text delimited by quote (") characters, the first non whitespace character encountered is taken to be the delimiter.
- s convert contiguous non whitespace text to a zero terminated string at the TEXT * buffer. The string is determined from the first non whitespace character to the first whitespace character (or a zero terminator).
- u convert an unsigned decimal number to the UWORD
- x convert an unsigned hexadecimal number to the UWORD

When performing a numeric conversion, for a number to be recognised the string must contain at least one digit in the radix.

If <long> is present, it should be L or l to specify that the corresponding argument address points to either a LONG or a ULONG (ignored if <type> is s, f or q). A long parameter can also be indicated by specifying the conversion type in upper case.

For example, after:

```
INT ret;
WORD d1,d2;
TEXT *ptr="111,-33";
ret=p_stoa(&ptr,"%b,%d",&d1,&d2);
```

d1 is 7 and d2 is -33, ptr points to the terminating zero, ret is zero. After:

```
INT ret;
WORD d1,d2;
TEXT buf[16]
ptr="xxx @a ^def ghi^ #44a";
ret=p_stoa(ptr,"@ %c %15q # %d",&d1,buf,&d2);
```

d1 contains 'a', "def ghi" is written to buf and d2 is 44, ptr points to 'a' and ret is zero.

Rectangle functions

This section describes a set of functions that operate on P_RECT structs. A P_RECT struct describes a rectangle in terms of its:

- top left coordinates (internal)
- bottom right coordinates (external)

where the units of the coordinates depend upon the application. Typically, the coordinates count pixels (for graphics displays) or monospaced character columns and rows (for character-oriented displays).

For example, a character display would normally be mapped to an (x,y) coordinate system as follows:

- (0,0) corresponds to the character in the top left corner
- x increases to the right and counts the character columns
- y increases downwards and counts character rows

Using the above coordinates, the rectangle of As in the following character display:

```
++++++++
++AAA+++
++AAA+++
+++++Z++
```

is described by the coordinates of the top left A (internal) and the bottom right Z (external) - that is, (2,1) and (5,3). Subtracting corresponding coordinates gives the correct dimensions of the rectangle - (3,2).

The P_RECT struct is defined in terms of two P_POINT structs. The definitions, in *p_graf.h*, are:

```
typedef struct
    {
    WORD x; /* Horizontal coordinate */
    WORD y; /* Vertical coordinate */
    } P_POINT;

typedef struct
    {
    P_POINT tl; /* Top left point (internal) */
    P_POINT br; /* Bottom right point (external) */
    } P_RECT;
```

An empty rectangle is a rectangle that has one or both of its sides zero or negative.

The rectangle functions are as follows:

p_offrec	moves a rectangle by an offset
p_insrec	shrinks or expands a rectangle about its centre
p_unirec	calculates the union of two rectangles (the smallest rectangle that encloses both of them)
p_intrec	calculates the intersection of two rectangles
p_pinrec	determines whether a point is inside a rectangle
p_emprec	determines whether a rectangle is empty
p_absrec	converts any negative sides of a rectangle to their positive equivalents

p_offrec — Offset a rectangle

```
VOID p_offrec(P_RECT *rect, INT xoffset, INT yoffset);
```

Displace rect by (xoffset,yoffset), without changing its size.

p_insrec — Inset a rectangle

```
VOID p_insrec(P_RECT *rect, INT xinset, INT yinset);
```

Adjust the width and height of rect by xinset and yinset respectively, in such a way as to produce a rectangle concentric with the original.

A negative inset makes the rectangle bigger.

For example:

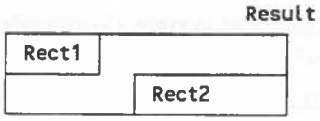
```
p_insrec(&rect,2,-1);
```

decrease the width of rect by 4 and increases the height by 2.

p_unirec — Union of two rectangles

```
VOID p_unirec(P_RECT *rect1, P_RECT *rect2, P_RECT *result);
```

Write the union of the two rectangles rect1 and rect2 (the smallest rectangle that encloses both of them) to result - as illustrated by the following diagram:



The parameter result may point to the same address as either rect1 or rect2.

For example:

```
LOCAL_D P_RECT rect1={{10,20},{30,40}};
LOCAL_D P_RECT rect2={{50,50},{100,120}};
LOCAL_D P_RECT result;

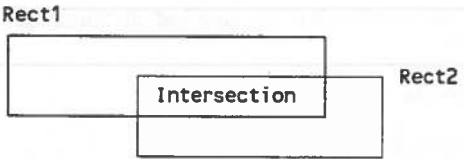
p_unirec(&rect1,&rect2,&result);
```

writes {{10,20},{100,120}} to result.

p_intrec — Intersection of two rectangles

```
INT p_intrec(P_RECT *rect1, P_RECT *rect2, P_RECT *result);
```

Write the intersection of the two rectangles rect1 and rect2 (the largest rectangle that is contained in both of them) to result.



If the rectangles do not intersect or if either rectangle is empty, the result is an empty rectangle.

The parameter result may point to the same address as either rect1 or rect2.

Returns TRUE if the rectangles intersect, FALSE if the rectangles do not intersect (or if either rectangle is empty).

For example, after:

```
LOCAL_D P_RECT rect1={{0,0},{7,20}};
LOCAL_D P_RECT rect2={{4,4},{100,120}};
LOCAL_D P_RECT result;

ret=p_unirec(&rect1,&rect2,&result);
```

ret is TRUE and result contains {{4,4},{7,20}}.

p_pinrec — Test if a point is inside a rectangle

```
INT p_pinrec(P_POINT *point, P_RECT *rect);
```

Return TRUE if the point point is within the rectangle rect or FALSE if point is outside rect (or if rect is empty).

p_emprec — Test if a rectangle is empty

```
INT p_emprec(P_RECT *rect);
```

Return TRUE if the rectangle rect is empty (that is, if it has zero or negative width or height).

p_absrec — Convert to an absolute rectangle

```
VOID p_absrec(P_RECT *rect, P_RECT *result);
```

Write the absolute of the rectangle `rect` (where any negative sides are converted to their positive equivalents) to `result`.

The parameters `rect` and `result` may point to the same address.

For example:

```
LOCAL_D P_RECT rect={{7,20},{4,4}};
LOCAL_D P_RECT result;

p_absrec(&rect,&result);
```

writes `{{4,4},{7,20}}` to `result`.

CHAPTER 5

FLOATING POINT

Floating point C

The 8087 emulator

The code that is generated from floating point C requires that the 8087 floating point coprocessor emulator be present and loaded. The 8087 emulator is implemented as an external logical device driver (LDD), loaded from *sys$8087.ldd*. The PLIB (or CLIB) startup module (that is, the code which calls `main`) automatically loads *sys$8087.ldd* if the application program contains any floating point code.

The startup module searches for *sys$8087.ldd* in the following directories (in order of precedence):

- as specified by the zero terminated string in the environment variable with name `"EM$"` (if such an environment variable exists)
- in the same directory that contained the program being executed

The startup module will fail with panic 80 the search for `sys$8087.ldd` fails. See the chapter on *Error Handling* for an explanation of the panic mechanism.

The `EM$` environment variable may be set using `p_setenv` - as described in the chapter *Memory Allocation*. For example, the following installation program causes the startup module to look for *sys$8087.ldd* in the *a:\sys* directory:

```
#include <plib>

GLDEF_C main(VOID)
    {
    p_setenv("EM$","A:\\SYS\\");
    }
```

Note that environment variables names are case-sensitive - setting up (say) `"em$"` will not have the desired effect.

Only one copy of *sys$8087.ldd* is loaded however many floating point processes are started. The LDD is deleted (freeing the memory) when the last floating point process exits gracefully. If the last process panics or is stopped by another process, the LDD will remain loaded - but it will be deleted by a subsequent normal exit of a floating point process. The LDD has the device name `EM$`, so you can find out if the device is loaded using:

```
LOCAL_C INT Is8087Loaded(VOID)
    {
    TEXT bb[E_MAX_NAME+2];

    return(p_devfnd(0,"EM$",E_LDD,&bb[0])>0);
    }
```

which returns `TRUE` if the LDD is loaded. If required, a clean-up program can delete the loaded LDD with:

```
p_devdel("EM$",E_LDD);
```

which will only successfully delete the LDD if there are no floating point processes using it. The functions `p_devfnd` and `p_devdel` are described along with LDDs in the *I/O System* chapter.

If *sys$8087.ldd* is not present in the ROM, it must be loaded into RAM at a cost of approximately 8K bytes (but this does not detract from the code segment limit of the application).

If the LDD is present in the ROM, it still has to be loaded but at a reduced cost of RAM. You can find out if *sys$8087.ldd* is present in the ROM using:

```
LOCAL_C INT Is8087InROM(VOID)
    {
    P_INFO info;

    return(p_finfo("ROM::SYS$8087.LDD",&info)>=0);
    }
```

which returns `TRUE` if *sys$8087.ldd* is present in the ROM.

Avoiding the 8087 emulator

It is possible to perform floating point operations without using floating point C and without loading the emulator. For example, a program that contains the following function (to evaluate sin(x)/x):

```
LOCAL_C INT Sinc(DOUBLE *pret,DOUBLE *parg)
    {
    if (*parg==0.0)
        {
        *pret=1.0;
        return(0);
        }
    if ((ret=p_sin(pret,parg))<0)
        return(ret);
    *pret=*pret/*parg;
    return(0);
    }
```

will load the emulator because the expressions `*parg==0.0`, `*pret=1.0` and `*pret=*pret/*parg` all generate calls to the emulator.

However, if the function is implemented without using floating point C as in:

```
LOCAL_C INT Sinc(DOUBLE *pret,DOUBLE *parg)
    {
    WORD x;
    DOUBLE zero;

    x=0;
    p_itof(&zero,&x);
    if (!p_fcmp(&zero,parg))
        {
        x=1;
        p_itof(pret,&x);
        return(0);
        }
    if ((ret=p_sin(pret,parg))<0)
        return(ret);
    p_fdiv(pret,parg);
    return(0);
    }
```

then *sys$8087.ldd* is not loaded. Furthermore, the code that is generated is smaller and runs faster.

The benefits of avoiding the emulator are:

- you do not need to worry about the presence of *sys$8087.ldd*
- the program is smaller and runs faster

The benefits of using the emulator are:

- you can use floating point C and use 32-bit float variables (as well as 64-bit doubles)
- the emulator works with 80-bit numbers internally and therefore gives more precise results

Macros

The following macros, defined in *p_math.h*

```
#define ABS(x) ((x)<0 ? -(x) : (x))

#define MAX(a,b) ((a)>(b) ? (a) : (b))

#define MIN(a,b) ((a)<(b) ? (a) : (b))
```

can be used with integer or floating point expressions (although the code that is generated might be quite lengthy). Using any of these macros with a floating point expression will cause the 8087 emulator to be loaded.

The remaining functions in this chapter, with the exception of `p_rand` and `p_randl`, are implemented independently of the emulator. Using them will not cause the emulator to be loaded. If the emulator is loaded, they may still be used.

Converting doubles to and from text

p_dtob — Double to string

```
INT p_dtob(TEXT *pbuf, DOUBLE *pval, P_DTOB *pformat);
```

Convert the double floating point number `*pval` to text at `pbuf` using the `pformat` format specification.

The `P_DTOB` structure is defined in *p_math.h* as:

```
typedef structure
    {
    UBYTE type;    /* conversion type */
    UBYTE width;   /* width of representation in characters */
    UBYTE ndec;    /* number of decimal places */
    TEXT  point;   /* decimal point character */
    TEXT  triad;   /* triad separator character */
    UBYTE trilen;  /* threshold for triad character use */
    } P_DTOB;
```

where:

`type` — specifies the numeric format as being fixed, scientific or general. (Integer format is obtained as a special case of fixed point format where the number of decimal places `ndec` is zero.)

`width` — specifies the maximum number of characters allowed to represent the number (it must be in the range 1 to 255 inclusive). You must reserve `width` bytes at `pbuf`. If the formatted string would be wider than `width` then `E_GEN_FAIL` is returned. (If you want the output aligned and filled, post-process `pbuf` with `p_jtob`).

`ndec` — specifies the number of digits following the decimal point when `type` is `P_DTOB_FIXED` or `P_DTOB_EXPONENT` where `ndec` must be in the range zero to `P_FLT_PREC` (15) inclusive.

`point` — specifies the character used for the decimal point. It would normally be either '.'(dot) or ','(comma).

`triad` — specifies the triad separator character used to delimit groups of 3 digits in the integer part of a fixed point number. It would normally be one of '.'(dot) or ','(comma) or ' '(space).

`trilen` — is either zero to disable triad insertion, or a threshold number of digits above which triad insertion takes place. In practice, `trilen` is set to 1 for normal conventions and 4 to conform to the French convention.

In all cases, negative numbers are represented by the insertion of a leading '-' sign (positive numbers do not have a leading '+' sign). To obtain a bracketed representation of negative numbers you must post-process the output string.

There can never be more than P_FLT_PREC(15) significant digits. Where there are less than P_FLT_PREC significant digits, the number is rounded to the number of significant digits displayed.

The detailed formatting details as a function of type is as follows:

P_DTOB_FIXED	the number is represented with ndec decimal places, where ndec may be zero to represent an integer (in which case no decimal point character is displayed). If the ASCII form exceeds width (usually due to the number being large and having too many digits before the decimal point), E_GEN_FAIL is returned. A zero is displayed in the form "0.000" where there are ndec zeros following the decimal point or as just "0" if ndec is zero.
P_DTOB_EXPONENT	the number is represented in scientific notation with one non-zero digit before the decimal point and ndec digits beyond the decimal point followed by 'E', a sign ('+' or '-') and the exponent as two digits (with leading zero if necessary). If ndec is zero, the number is rounded to one digit of precision and no decimal point is displayed. A zero is displayed in the form "0.000E+00" where there are ndec zeros following the decimal point or as "0E+00" if ndec is zero. Triad separation is not available and triad separation parameters are ignored.
P_DTOB_GENERAL	converts either as fixed format (with no triad separator) or scientific format, making best use of width. Here, "making best use" is defined as showing the greater number of significant digits and preferring fixed format when the number of significant digits shown is the same. The number of decimal places displayed depends on width (ndec is ignored). A zero is displayed as just "0". Triad separation is not available and triad separation parameters are ignored.
P_DTOB_GEN_LIM	as for P_DTOB_GENERAL, except that output is limited to not exceed 12 significant digits.

Returns the number of characters written to pbuf if successful, or one of the following negative error numbers:

E_GEN_UNDER	the number is too small to represent (less than approximately 1E-99). If you would prefer not to fail in this case, you can always write your own zero or recall p_dtob with a zero double.
E_GEN_OVER	the number is too large to represent (greater than approximately 1E99).
E_GEN_FAIL	the representation exceeds width characters.
E_GEN_ARG	either the double is illegal, or type is not one of P_DTOB_FIXED, P_DTOB_EXPONENT or P_DTOB_GENERAL.

p_stod — String to double

```
INT p_stod(TEXT **pstr, DOUBLE *pval, INT point);
```

Scan the zero terminated string *pstr for a floating point number where point is the code of the decimal point character (normally either '.' or ',') and write the value as a double float to *pval.

For a number to be recognised, the string must contain at least one decimal digit.

If p_stod is successful, it updates *pstr to point to the terminating character and returns zero. If it fails, *pstr is not changed and it returns one of the following negative error numbers:

E_GEN_UNDER	the number is too small (less than approximately 1E-99). The value zero is written to *pval.
E_GEN_OVER	the number is too large (greater than approximately 1E99).
E_GEN_FAIL	fails to recognise a number.

The supplied string at *pstr should take the form:

```
[+|-]<int>.<fract>[E|e][+|-]<exp>
```

where:

- the leading '+' sign may be omitted for positive numbers
- <int> and <fract> are optional but at least one should be present

- leading zeros in <int> are legal but have no effect
- trailing zeros in <fract> are legal but have no effect
- there is no reasonable limit to the number of significant digits, but digits that are beyond the precision of the floating point representation will not be reflected in the mantissa of the number which is produced
- the exponent field (which starts with 'E' or 'e') is optional
- the leading '+' sign in the exponent field may be omitted for positive exponents
- the resulting number should be in the range approximately 1E-99 to approximately 1E+99

Example

```
LOCAL_D TEXT buf[]="-134.43735abc";
FAST INT ret;
DOUBLE val;
TEXT *ptr;
....
ptr=&buf[0];
```

After

```
ret=p_stod(&ptr,&val,'.');
```

`val` will be -134.43735, `ptr` will be pointing to 'a' and `ret` will be zero.

p_getctd — Get number representation preferences

```
VOID p_getctd(E_CONFIG *pcfg);
```

Write a copy of the system `E_CONFIG` struct to `pcfg`.

The `E_CONFIG` struct is defined in *p_config.h* as:

```
typedef struct
    {
    UWORD countryCode;
    WORD gmtOffset;
    UBYTE dateType;
    UBYTE timeType;
    UBYTE currencySymbolPosition;
    UBYTE currencySpaceRequired;
    UBYTE currencyDecimalPlaces;
    UBYTE currencyNegativeInBrackets;
    UBYTE currencyTriadsAllowed;
    UBYTE thousandsSeparator;
    UBYTE decimalSeparator;
    UBYTE dateSeparator;
    UBYTE timeSeparator;
    UBYTE currencySymbol[9];
    UBYTE startOfWeek;
    UBYTE summerTime;
    UBYTE clockType;
    UBYTE dayAbbreviation;
    UBYTE monthAbbreviation;
    UBYTE workDays;
    UBYTE units;
    UBYTE spare[9];
    } E_CONFIG;
```

In the context of this chapter we are interested in:

`currencySymbol`	a zero terminated string containing the currency symbol
`currencySymbolPosition`	which should contain either `E_CURRENCY_BEFORE` or `E_CURRENCY_AFTER`
`currencySpaceRequired`	which should contain either `E_NOSPACE_BETWEEN` or `E_SPACE_BETWEEN`
`currencyDecimalPlaces`	the number of decimal places for displaying currency figures

currency-NegativeInBrackets	TRUE if a negative currency should be displayed in brackets rather than with a minus sign
currencyTriadsAllowed	zero to disable triad separator insertion, or a threshold number of digits above which triad separator insertion takes place. In practice a value of 1 is used for normal conventions and 4 for the French convention. Note that, despite the name of this element, triad separators are not restricted to currency fields; they may be inserted in any numeric field.
thousandsSeparator	the character code of the triad (thousands) separator
decimalSeparator	the character code of the decimal separator (normally either ',' or '.')
units	either E_IMPERIAL or E_METRIC to indicate a preference for imperial or metric units (for example, to show page dimensions in inches or centimetres)

Long integer functions

p_randl — Long random number

```
ULONG p_randl(ULONG *pseed);
```

Return the next pseudo random number and updates *pseed.

Used to generate a sequence of pseudo random numbers from an initial value of *pseed. Any given seed will always produce the same sequence of random numbers.

The numbers generated may be any value between 0 and 4294967295 (0xffffffff) or, if considered as a signed result, between -2147483648 (0x80000000) and +2147483647 +(0x7fffffff).

For example, to print reproducibly 100 random longs:

```
ULONG seed;
UINT i;

seed=0l;
for (i=0;i<100;i++)
    p_printf("%ld",p_randl(&seed));
```

To generate a different set of numbers each time, seed the number with the system time, as in:

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

Scientific functions

For all floating point functions that transform a single input parameter it is permissible to use the same address for both parg and pret.

All trigonometric functions assume angles are measured in radians.

p_sin — Sine

```
INT p_sin(DOUBLE *pret, DOUBLE *parg);
```

Write the sine of *parg to *pret.

Returns zero if successful or E_GEN_ARG if *parg was an invalid double.

p_cos — Cosine

```
INT p_cos(DOUBLE *pret, DOUBLE *parg);
```

Write the cosine of *parg to *pret.

Returns zero if successful or E_GEN_ARG if *parg was an invalid double.

p_tan Tangent

```
INT p_tan(DOUBLE *pret, DOUBLE *parg);
```

Write the tangent of a *parg to *pret.

Returns zero if successful or E_GEN_ARG if *parg was an invalid double or if it was greater than 149078413.

p_asin Arcsine

```
INT p_asin(DOUBLE *pret, DOUBLE *parg);
```

Write the angle whose sine is *parg to *pret.

Returns zero if successful or E_GEN_ARG if *parg was an invalid double or ABS(*parg)>1.

p_acos Arccos

```
INT p_acos(DOUBLE *pret, DOUBLE *parg);
```

Write the angle whose cosine is *parg to *pret.

Returns zero if successful or E_GEN_ARG if *parg was an invalid double or ABS(*parg)>1.

p_atan Arctangent

```
INT p_atan(DOUBLE *pret, DOUBLE *parg);
```

Write the angle whose tangent is *parg to *pret.

Returns zero if successful or E_GEN_ARG if *parg was an invalid double.

p_ln Natural logarithm

```
INT p_ln(DOUBLE *pret, DOUBLE *parg);
```

Write the natural (base e) logarithm of *parg to *pret.

Returns zero if successful or E_GEN_ARG if *parg was less than or equal to zero or if *parg was an invalid double.

p_exp Exponential

```
INT p_exp(DOUBLE *pret, DOUBLE *parg);
```

Write the value of the arithmetic constant e (2.71828...) raised to the power of *parg to *pret.

Returns zero if successful or one of the following negative error numbers:

E_GEN_ARG	*parg is not a valid double.
E_GEN_UNDER	underflow has occurred (the magnitude of the result is less than 1E-307), zero is written to *pret.
E_GEN_OVER	overflow has occurred (the magnitude of the result is greater than 1E307).

p_log Logarithm

```
INT p_log(DOUBLE *pret, DOUBLE *parg);
```

Write the base 10 logarithm of *parg to *pret.

Returns zero if successful or E_GEN_ARG if *parg was less than or equal to zero or if *parg was an invalid double.

p_sqrt — Square root

```
INT p_sqrt(DOUBLE *pret, DOUBLE *parg );
```

Write the square root of *parg to *pret.

Returns zero if successful or E_GEN_ARG if *parg was negative or an invalid double.

p_pow — Raise to the power

```
INT p_pow(DOUBLE *pret, DOUBLE *parg1, DOUBLE *parg2);
```

Write *parg1 raised to the power of *parg2 to *pret.

Returns zero if successful or one of the following negative error numbers:

E_GEN_ARG	the arguments are invalid (if *parg1 < 0, *parg2 must be integral), or at least one argument is not a valid double.
E_GEN_UNDER	underflow has occurred (the magnitude of the result is less than 1E-307), zero is written to *pret.
E_GEN_OVER	overflow has occurred (the magnitude of the result is greater than 1E307).

p_rand — Double random number (long seed)

```
DOUBLE p_rand(ULONG *pseed);
```

Return a DOUBLE random number in the range zero (inclusive) to one (exclusive). As in the case of p_randl, the value pointed to by pseed is used to seed the random number generation and is updated for the next call of p_rand.

This function requires the 8087 floating point emulator.

p_frand — Double random number

```
VOID p_frand(DOUBLE *pret, ULONG *pseed);
```

Write a random number in the range zero (inclusive) to one (exclusive) to *pret. As in the case of p_randl, the value pointed to by pseed is used to seed the random number generation and is updated for the next call of p_frand.

Floating point arithmetic without the 8087 emulator

This section describes the PLIB functions that would normally be used to perform floating point arithmetic without having to load the 8087 floating point emulator *sys$8087.ldd*.

For all floating point functions with one parameter it is permissible to use the same address for both parg and pret.

p_fld — Assignment

```
INT p_fld(DOUBLE *pret, DOUBLE *parg);
```

Write the value of *parg to *pret (the 'ld' stands for load).

Returns zero if successful or E_GEN_ARG if *parg was an invalid double.

p_fadd — Add

```
INT p_fadd(DOUBLE *pret, DOUBLE *parg);
```

Write the sum of *parg and *pret to *pret.

Returns zero if successful or one of the following negative error numbers:

E_GEN_ARG	*pret or *parg was not a valid double.

E_GEN_OVER	overflow has occurred (the magnitude of the result is greater than 1E307).

p_fsub Subtract

```
INT p_fsub(DOUBLE *pret, DOUBLE *parg);
```

Write *pret minus *parg to *pret.

Returns zero if successful or one of the following negative error numbers:

E_GEN_ARG	*pret or *parg was not a valid double.
E_GEN_OVER	overflow has occurred (the magnitude of the result is greater than 1E307).

p_fmul Multiply

```
INT p_fmul(DOUBLE *pret, DOUBLE *parg);
```

Write the product of *parg and *pret to *pret.

Returns zero if successful or one of the following negative error numbers:

E_GEN_ARG	*pret or *parg was not a valid double.
E_GEN_UNDER	underflow has occurred (the magnitude of the result is less than 1E-307), zero is written to *pret.
E_GEN_OVER	overflow has occurred (the magnitude of the result is greater than 1E307).

p_fdiv Divide

```
INT p_fdiv(DOUBLE *pret, DOUBLE *parg);
```

Write *pret divided by *parg to *pret.

Returns zero if successful or one of the following negative error numbers:

E_GEN_ARG	*pret or *parg was not a valid double.
E_GEN_UNDER	underflow has occurred (the magnitude of the result is less than 1E-307), zero is written to *pret.
E_GEN_OVER	overflow has occurred (the magnitude of the result is greater than 1E307).

p_fcmp Compare

```
INT p_fcmp(DOUBLE *parg1, DOUBLE *parg2);
```

Compare *parg1 to *parg2, returning:

1	if *parg1 > *parg2
0	if *parg1 == *parg2
-1	if *parg1 < *parg2

The function only returns zero if *parg1 and *parg2 are strictly equal. In some cases it may be more appropriate to test for equality by evaluating the difference using p_fsub and then comparing the difference with a suitably small number (such as, for example, 1E-10).

The return value is undefined if either *parg1 or *parg2 is not a valid double.

Note that it is bad practice to compare floating point numbers for equality, since rounding errors may make the result meaningless.

p_fneg Negate

```
INT p_fneg(DOUBLE *parg);
```

Negate *parg.

Returns zero if successful, or E_GEN_ARG if *parg was an invalid double.

p_mod — Modulus

```
INT p_mod(DOUBLE *pret, DOUBLE *parg1, DOUBLE *parg2);
```

Write the remainder of *parg1 divided by *parg2 to *pret.

Returns zero if successful, or E_GEN_ARG if either *parg1 or *parg2 was an invalid double.

p_int — Integer part

```
INT p_int(DOUBLE *pret, DOUBLE *parg);
```

Write the integer part of *parg to *pret. Negative numbers are rounded towards zero.

Returns zero if successful or E_GEN_ARG if *parg was an invalid double.

p_inti — Convert double to integer

```
INT p_inti(WORD *pret, DOUBLE *parg);
```

Write the integer part of *parg to *pret if *parg is in the range -32768 to +32767 inclusive. Negative numbers are rounded towards zero.

Returns zero if successful or one of the following negative error numbers:

E_GEN_ARG	*parg is not a valid double
E_GEN_OVER	*parg is outside the range -32768 to +32767

p_intl — Convert double to long

```
INT p_intl(LONG *pret, DOUBLE *parg);
```

Write the integer part of *parg to *pret, if it is in the range -2147483648 (0x80000000) to +2147483647 (0x7fffffff) inclusive. Negative numbers are rounded towards zero.

Returns zero if successful, or one of the following negative error numbers:

E_GEN_ARG	*parg is not a valid double
E_GEN_OVER	*parg is outside the range -2147483648 to +2147483647

p_itof — Convert integer to double

```
VOID p_itof(DOUBLE *pret, WORD *parg);
```

Convert *parg to a double and write it to *pret.

p_longtof — Convert long to double

```
VOID p_longtof(DOUBLE *pret, LONG *parg);
```

Convert *parg to a double and write it to *pret.

CHAPTER 6

ERROR HANDLING

Process termination

A process can terminate itself or it can be terminated by another process. By whatever means a process terminates, one or more other processes may be interested in being notified that a process has terminated.

Terminating this process.

A process terminates itself by calling:

`p_exit`	for normal graceful termination
`p_panic`	for abnormal termination, normally as a result of defective code

C programs that fall off the end of `main` effectively call `p_exit` with the return from `main`. That is:

```
GLDEF_C INT main(VOID)
    {
    p_printf("Hello world");
    p_sleep(50l); /* wait 5 seconds */
    return(0);
    }
```

is equivalent to:

```
GLDEF_C VOID main(VOID)
    {
    p_printf("Hello world");
    p_sleep(50l); /* wait 5 seconds */
    p_exit(0);
    }
```

It is poor practice to fall off the end of a `VOID main` since this is equivalent to calling `p_exit` with a random number (whatever happens to be in the `AX` register at the time). If there is a system component (such as the shell) reporting process terminations to the user, it will give a misleading report.

Terminating another process

A process can terminate another process by calling:

`p_pterminate` or `p_pkill`	to terminate another process (typically in response to a user request)
`p_ppanic`	to panic another process (normally following unreasonable behaviour from the process being terminated)

To terminate a process, it is recommended that `p_pterminate` is used in preference to `p_pkill` as the former allows the process being terminated to run any cleanup code before exiting gracefully. Applications wishing to run cleanup code call `p_onterminate` to elect to be sent an inter-process message in response to the `p_pterminate` request.

Finding out when other processes terminate

A process can request to be notified of the termination of another process by calling:

`p_logona`	to be signalled when the specified process terminates
`p_logon`	to receive an inter-process message when the specified process terminates (convenient for server processes to keep track of their clients)
`p_watchall`	to receive an inter-process message when any process terminates (normally used by the Shell to monitor the termination of all processes)

See the chapter *Asynchronous Requests and Semaphores* for the meaning of the term "signalled". See the chapter *Processes and Inter-Process Messaging* for more about inter-process messaging and servers.

The process termination word

Regardless of which one of `p_logona`, `p_logon` or `p_watchall` is used, the system delivers a single 16-bit process termination word giving information on how the process terminated. The most significant byte of the process termination word is one of the following (defined in *epoc.h*)

`E_NORMAL_EXIT`	the process terminated itself by calling `p_exit` or it was terminated by another process calling `p_pterminate` or `p_pkill`. The least significant byte of the process termination word contains the `nReason` parameter to `p_exit`, `p_pterminate` or `p_pkill`. By convention, an `nReason` of zero indicates a normal error-free graceful termination.
`E_PANIC_EXIT`	the process terminated itself by calling `p_panic` or it was terminated by another process calling `p_ppanic`. The least significant byte contains the `nPanic` ("panic number") parameter to `p_panic` or `p_ppanic`.
`E_TASK_PANIC_EXIT`	the process terminated because it owned a task that terminated with `E_PANIC_EXIT` (tasks are lightweight processes and are described in the chapter *Processes and Inter-Process Messaging*).

Panics

When the system detects a condition that it believes could only arise from a bug in a the application program, the system terminates the process with a "panic number" in the range 0 to 255 inclusive (where the system is said to "panic the process"). A panic is a fatal exception that causes the process to terminate immediately. There is no way for applications to avoid being terminated when a panic has been started (cf `p_leave`, described later in this chapter).

Panicking a process is more economical in the use of system code and application code than returning an error, since the latter relies on application code to properly process the error return. As well as protecting the system from defective applications, the panic system enforces a greater discipline on application code by terminating a process as soon as the condition is detected.

This does not mean that the system always prefers to use panics rather than error returns. Error returns are still used where appropriate; the system certainly never panics a process on a condition that could arise from user action. For example, the `p_alloc` function to allocate a memory cell returns `NULL` if there is insufficient free system memory to satisfy the request, but calls `p_panic` if it detects that the heap has been corrupted. In the majority of cases, such as in the above example, it is quite clear where a panic or an error return is appropriate, but occasionally it is not so clear. From the point of view of an application programmer, however, it is clear that panic conditions should be avoided. In consequence the function descriptions in this manual include any conditions that result in the caller being panicked.

If the panic condition is detected within the code of a system function, the function calls `p_panic` as in the example above. If the condition is detected in a system process (for example, the supervisor or the file server) the application process is terminated by the system process calling `p_ppanic`.

Application programmers can include their own calls to `p_panic` to catch conditions indicating a bug in their program which might otherwise go unnoticed (until the software is used by a customer!). A common example would be to put a call to `p_panic` on the `default` case of a `switch` statement, to catch an invalid parameter.

System panic numbers

The following lists the panic numbersthat are used by the operating system:

00 Used by test code when a test fails

01 Invalid function number (semaphore manager)

02 Invalid semaphore handle

03 Semaphore not allocated

04 Initial semaphore count is negative

05 Signal count is negative

06 Invalid function number for process manager

07 Invalid process ID

08 Task tried to create a task

09 Invalid function number for time manager

10 Invalid function number for segment manager

11 Segment size was negative

12 Type was not one of `E_SEGMENT_LOW`, `E_SEGMENT_HIGH`, `E_SEGMENT_DEVICE` or `E_SEGMENT_LOCKED`

13 Invalid segment handle

14 Segment copy is out of range

15 Invalid function number for heap manager

16 Heap not initialised

17 A heap cell is being reduced by more than its size

18 Attempt to set heap granularity greater than `E_MAX_GROWBY`

19 A heap cell address is outside the boundaries of the heap (the heap has probably been corrupted - try calling `p_allchk` to catch the corruption sooner)

20 Invalid function number for inter-process message manager

21 Inter-process messaging has already been initialised (ie `p_minit` has been called twice)

22 Inter-process messaging has not been initialised (ie `p_minit` has not been called)

23 Cannot initialise with zero messages in the queue

24 Invalid function number for I/O manager

25 Invalid I/O channel (possibly because you did not test that the previous `p_open` succeeded or you have closed the channel or you overwrote the variable containing the channel)

26 Device requested panic

27 Invalid wait handler handle (possibly nothing to do with wait handlers and just indicative of a low address overwrite of the 4 bytes at address 2, as a result of an uninitialised pointer)

28 Key and pointing device already hooked

29 Key and pointing device requesting process is not a task

30 Invalid function number for device manager

31 Invalid device handle

32 Invalid function number for file manager

33 Process already connected to file server

34 Reserved for future use

35 Invalid function number for library manager

36 Invalid library handle

37 Invalid function number for library

38 Invalid LIB file channel

39 Invalid DYL index number

40 Invalid message to file server

41 Process has not connected to file server

42 Invalid function number for conversion manager

43 Invalid function number for general manager

44 Attempt to unhook from notify when not already hooked

45 Invalid revector address

46 Invalid function number for conversion manager

47 Leave called before a call to enter (possibly because you were unaware that the function you were calling could call `p_leave`)

48 No method available to handle message (OOP)

49 Invalid reclass attempted (OOP)

50 Unknown category in LibHandle (OOP)

51 Unknown class in LibCreate (OOP)

52 Supersend called from outside a method (OOP)

53 Attempt to get a handle before being linked (OOP)

54 Missing external categories in LibLink (OOP)

55 Object does not point to a valid class (OOP)

56 Invalid link layer completion code

57 Invalid function number for window server

58 Invalid function number for hardware manager

59 Unexpected interrupt

60 Attempted to write outside of process data segment (possibly because of an uninitialised pointer or a corrupted data structure)

61 Interrupts have been disabled for too long

62 Reserved for future use

63 Divide by zero interrupt

64 Overflow interrupt

65 Invalid function number for Dbf manager

66 Invalid DBF I/O channel

67 Invalid parameter for DBF function

68 Address zero overwrite (possibly because of an uninitialised pointer)

69 The operating system detected less than 0x100 bytes of remaining stack (this amount is reserved for hardware interrupts to run). You probably have declared large data structures as automatics. Consider making them static variables or allocate them from the heap.

70 Environment name size > EnvMaxNameSize

71 Single step interrupt (INT 1)

72 Break point interrupt (INT 3)

73 A request was made while an asynchronous request of the same type and on the same channel was already pending

74 Invalid function number for serial I/O manager

75 Call to an ASIC1 function on an ASIC9 machine

76 Attempt to find a DYL not in a visible bank

77 Floating point emulator exception

78 Semaphore count exceeds 0x7fff

80 Library fatal error, preceded by a notification of the specific error

255 The function p_allchk detected a corrupted heap

Some of the panics, especially those described by "Invalid function number for ...", are unlikely to indicate a specific bug; it is almost impossible to create code that would produce such a panic by a coding error. This type of panic could, however, easily arise from trashing a return address on the stack, with the instruction pointer wandering into arbitrary code. In the above list, those panics that are likely to indicate a specific coding problem are described more fully.

If you get a panic 79, or a panic in the range 81 to 254, it may be due to some other system component. See the *Window Server* manual for panics in the range 81-110 and, when using object oriented programming, see the *OLIB* manual for panics in the range 130-158.

p_exit — Terminate this process

```
VOID p_exit(INT nReason);
```

Terminate this process with reason nReason in the range -127 to 128 without returning to the caller.

Applications that exit normally should terminate with nReason equal to zero.

Applications that fail during their initialisation may wish to pass on the error number (in the range -127 to -1 inclusive) that caused the initialisation to fail.

Programs that are run as subprocesses can use a positive nReason to pass back an exit status to their parent process.

p_panic — Terminate after an unrecoverable error

```
VOID p_panic(INT nPanic);
```

Terminate this process with panic number nPanic in the range 0 to 255 without returning to the caller.

To avoid system panic numbers, application specific panic numbers should start from 254 downwards.

p_pkill — Unilaterally terminate a process

```
INT p_pkill(HANDLE pId, INT nReason);
```

Terminate process pId for reason nReason (in the range -127 to 128 inclusive) without giving the process being terminated an opportunity to run any cleanup code before exiting.

It is recommended that p_pterminate (described below) is used in preference to p_pkill as it gives the process being terminated a chance to run any cleanup code.

Returns zero if successful or one of the following negative error numbers:

E_FILE_NXIST	the process does not exist
E_GEN_FAIL	pId is the null process, the supervisor or the file server

p_pterminate — Terminate a process

```
INT p_pterminate(HANDLE pId, INT nReason);
```

Terminate process pId for reason nReason in the range -127 to 128 inclusive.

If process pId has called p_onterminate (described next) the call to p_pterminate will simply send the specified message number to process pId. Otherwise the effect is the same as with p_pkill. It is recommended that p_pterminate is used in preference to p_pkill.

Returns zero if successful or one of the following negative error numbers:

E_FILE_NXIST	the process does not exist
E_GEN_FAIL	pId is the null process, the supervisor or the file server

p_onterminate — Elect to receive termination message

```
VOID p_onterminate(INT nMessage);
```

Elect to receive (non-zero) message number nMessage, rather than being summarily terminated, when another process requests termination by calling p_pterminate.

On receipt of the termination message a process should execute its cleanup code and must then terminate itself, normally by calling p_exit.

The process calling p_onterminate must guarantee to respond promptly when the termination message is sent to it. It should not perform any lengthy, uninterrupted, processing. A process which calls p_onterminate and then (presumably in error) enters an infinite loop will not be terminated by a call to p_terminate. It is recommended that a process should not call p_onterminate unless there is an explicit reason for it to do so.

The election may be cancelled by calling p_onterminate(0).

Messaging must have been initialised with p_minit prior to the call to p_onterminate, otherwise p_panic will be called.

p_ppanic — Panic a process by id

```
INT p_ppanic(HANDLE pId, INT nPanic);
```

Terminate process pId with panic number nPanic in the range 0 to 255 (used for example by server processes that receive a garbage message from a client process to panic the client).

Returns zero if successful or one of the following negative error numbers:

E_FILE_NXIST	the process does not exist
E_GEN_FAIL	pId is the null process, the supervisor or the file server

p_logona — Request notification of process termination

```
INT p_logona(HANDLE pId, WORD *pStatus);
```

Make an asynchronous request to be notified of the termination of process pId by having this process I/O semaphore signalled when pId terminates. See the chapter *Asynchronous Requests and Semaphores* for a description of asynchronous requests and the I/O semaphore.

Returns zero if successful or E_FILE_NXIST if pId does not exist.

After a successful request and before pId has terminated, *pStatus contains E_FILE_PENDING. When pId has terminated, *pStatus contains the (non-negative) process termination word (as described under the heading *The Process Termination Word* at the beginning of this chapter).

The caller can cancel the asynchronous request by calling p_logoffa.

A server process that responds to inter-process messages from client processes will probably find it more convenient to use p_logon, described below.

For example, the following function:

```
GLDEF_C INT RunSubProcessWait(TEXT *name, BYTE *pReason)
    {
    HANDLE pid;
    WORD stat;

    if ((pid=p_execc(name,NULL,0))<0)
        return(pid);
    p_logona(pid,&stat);
    p_presume(pid);
    p_waitstat(&stat);
    *pReason=(BYTE)stat;
    return(stat>>8);
    }
```

loads the executable name, resumes the process, waits for it to terminate, writes the p_exit to *pReason and returns E_NORMAL_EXIT. It returns a negative error number if it fails to load name and E_PANIC_EXIT if the sub-process panics.

p_logoffa Cancel notification of process termination

```
INT p_logoffa(HANDLE pId);
```

Cancel a previously requested notification of the termination of process pId (as passed to the p_logona being cancelled).

Returns zero if successful or E_FILE_NXIST if no request is pending.

If the cancel gets through before pId terminates, *pStatus will contain E_FILE_CANCEL. In either case, the process I/O semaphore is signalled and p_logoffa would normally be followed by a call to p_waitstat(pStatus).

p_logon Request message on process termination

```
INT p_logon(HANDLE pId, INT mType);
```

Request to be notified of the termination of process pId by receiving an inter-process message of type mType when pId terminates. See the chapter *Processes and Inter-Process Messaging* for a description of inter-process messaging.

Returns zero if successful or E_FILE_NXIST if pId does not exist.

When process pId terminates the Supervisor process sends the caller a message of type mType and whose first word in the message buffer is the pId of the terminating process and whose second word is the process termination word giving information on how that process terminated (as described under the heading *The Process Termination Word* at the beginning of this chapter).

The caller can cancel the request by calling p_logoff or p_logoffx.

The process must have messages initialised by calling p_minit - the function calls p_panic if messages have not been initialised.

The p_logon, p_logoff and p_logoffx services were designed for server processes that respond to inter-process messages from client processes to clean up client specific resources should a client process terminate without disconnecting from the server. Processes that are not server process and that do not normally respond to inter-process messages will probably find it more convenient to use p_logona, described above.

p_logoff Cancel message on process termination

```
INT p_logoff(HANDLE pId, INT mType);
```

Cancel a previous p_logon request to be sent an inter-process message when process pId terminates.

The value of pId should be as passed to p_logon, and mType is ignored. This form is suitable for applications that do not make no more than one p_logon request to any particular process.

Applications that make two or more p_logon requests with the same value of pId (but, presumably, different values of mType) should cancel them by means of p_logoffx, described below.

Returns zero if successful or E_FILE_NXIST if pId does not exist.

The function calls p_panic if messages have not been initialised.

p_logoffx Cancel message of specific type on process termination

```
INT p_logoffx(HANDLE pId, INT mType);
```

This function is only available in EPOC version 3.18 or later.

Cancel a previous p_logon request to be sent an inter-process message of type mType when pId terminates (pId and mType should be the values that were passed to the p_logon request that is being cancelled).

This function must be used in preference to p_logoff in cases where two or more p_logon requests are made with the same value of pId.

Returns zero if successful or E_FILE_NXIST if pId does not exist.

The function calls p_panic if messages have not been initialised.

p_watchall — Watching all exits

```
INT p_watchall(UINT mType);
```

Request to be notified of the termination of any process by receiving an inter-process message of type `mType` when a process terminates. See the chapter *Processes and Inter-Process Messaging* for a description of inter-process messaging.

Only one process at a time can request this service and it is usually reserved for use by a system process (normally the Shell process) to monitor the termination of all processes.

The function returns zero if successful or `E_GEN_FAIL` if a watch is already active.

The format of the received message is as for `p_logon`, described above.

Calling `p_watchall` with an `mType` of zero cancels the request.

The process must have messages initialised by calling `p_minit` - the function calls `p_panic` if messages have not been initialised.

Error returns

System functions that can fail must somehow indicate success or failure and, where appropriate, elaborate on the failure.

Where there is no elaboration of the error:

- Functions that return an address typically return a `NULL` (zero) address to indicate failure (this is often used when a function can fail to allocate memory).
- Otherwise functions return zero or positive to indicate success and -1 to indicate failure (the constant `E_GEN_FAIL` is defined as -1 although you can just test for the sign of the returned value).

Where the error is elaborated, the system function returns a system error number in the range -1 to -128, allocated as follows:

-1 to -31	Reserved for general errors of the form `E_GEN_xxx`; (defined in *p_gen.h*)
-32 to -63	Reserved for I/O device errors of the form `E_FILE_xxx` (defined in *p_file.h*)
-64 to -95	Reserved for future use
-96 to -128	Reserved for OPL run-time errors

The system errors include both generic error numbers (eg `E_GEN_NOMEMORY`) and specific error numbers (eg `E_FILE_PARITY` indicating a parity error in a byte received via a serial port). Application programmers may wish to use the generic error numbers in their own code - see the contents of *p_gen.h* and *p_file.h*.

The system stores a language dependent description for each of the system error numbers that may be retrieved by calling `p_errs`.

p_errs — Convert error number to string

```
VOID p_errs(TEXT *str, INT errno);
```

Convert a system error number to a language dependent zero terminated text string in `str`. There should be at least `E_MAX_ERROR_TEXT_SIZE` (64) bytes at address `str`.

If the error is an unknown error the string "Unknown error [xx]" (or a suitable translation if not an English ROM) is returned, where xx is the value of `errno`.

Notifier services

The notifier services are used to inform the user of a condition - particularly an error condition - and to present the user with up to three options on how to proceed. There are two variants of the notify service:

p_notifyerr	which presents the user with a system error message (converted to text from the error number using p_errs) and a contextual message
p_notify	which presents the user with two messages

These services can be used by any application and are particularly useful for processes that do not otherwise have a user interface (for example, the file server). Investigative calls to p_notify may be temporarily inserted into code when debugging programs.

At the level of the services described in this manual and except for the simple console functions such as p_printf, the operating system does not define any user interface components and the notifier services rely on a higher level system process[1] (the "notifier") taking responsibility for implementing a user interface. The notifier "hooks" the user interface (normally at system start up) by calling p_notifyhook. The notifier should pre-allocate any memory it requires so that the call will not fail.

The notifier service is used by the file server process to give the user the opportunity to rectify a problem that would otherwise result in a file service request failing (eg to replace an SSD pack that has inadvertently been removed). This scheme works well when the requesting process is an interactive application but poorly if the requesting process is designed to run unattended (eg a communications program) or is itself a server process (eg the window server trying to read a font file). Such processes can use p_setnotify(FALSE) to stop servers such as the file server from using the notify service to give the user the opportunity to rectify an error condition.

p_notify Present the user with a message and get response

```
INT p_notify(TEXT *pT1, TEXT *pT2, TEXT *pO1, TEXT *pO2, TEXT *pO3);
```

Present the two zero terminated messages pT1 and pT2 to the user, where pO1, pO2 and pO3 are either NULL or zero terminated strings, offering up to three options for the user to select. The function waits for the user to select an option and returns zero if the pO1 option was chosen, 1 if the pO2 option was chosen and 2 if the pO3 option was chosen.

The message pT1 is presented before pT2. So pT1 would typically contain a contextual message (eg "Failed to save notes.tpd") with pT2 containing a more detailed message (eg "Disk full").

If pT2 is NULL the second message line is blank.

To offer the user 2 options rather than 3, pass pO3 as NULL.

To offer the user no option (that is, just to wait until the user has acknowledged the message) pass both pO2 and pO3 as NULL. Alternatively, pass all 3 as NULL as in:

```
p_notify(msg1,msg2,NULL,NULL,NULL);
```

which is equivalent (on an English machine) to:

```
p_notify(msg1,msg2,"CONTINUE",NULL,NULL);
```

Each message string pT1 and pT2 can be up to E_MAX_NOTIFY_TEXT_SIZE (64) in length including the zero terminator. Each option string pO1, pO2 and pO3 can be up to E_MAX_OPTION_TEXT_SIZE (16) in length including the zero terminator.

This function sends an inter-process message to the process that has "hooked" the notify interface and thereby has taken on the responsibility of presenting the error notification user interface. If no process has hooked the notifier, or if the notifier process has terminated, p_notify returns E_GEN_FAIL.

The notifier process may, in some environments, take special action if any of the strings passed to p_notify contain a leading zero. Programmers should therefore ensure that such a string is not passed as a parameter to p_notify. Either pass an explicit NULL (as in the above examples) or intercept the string, as in the following example, where it is assumed that only msg2 may contain a leading zero:

```
LOCAL_C INT NotifyError(TEXT *msg1,TEXT *msg2)
    {
    if (msg2 && !*msg2)
            msg2=NULL;
return(p_notify(msg1,msg2,NULL,NULL,NULL));
    }
```

[1]When a specialised process hooks the notifier, it is, by convention, called SYS$NTFY.

p_notifyerr — Notify user of error and get response

```
INT p_notifyerr(INT nError, TEXT *pT2, TEXT *p01, TEXT *p02, TEXT *p03);
```

Equivalent to calling p_errs(nError) followed by calling p_notify with the resulting error string as the second parameter and with pT2 as the first parameter (ie the first two parameters are the other way around compared to p_notify) with the three option parameters being passed on to p_notify.

The message pT2 is presented before the error text corresponding to nError and pT2 would typically contain a contextual message.

Only system error numbers (which include all errors returned by the functions described in this manual) should be notified using this service.

p_setnotify — Set notify state

```
VOID p_setnotify(UINT nState);
```

Set the notify state of this process.

If nState is FALSE, the system will not automatically present the notifier as a result of an error in a service requested by this process (the error will be returned directly).

If nState is TRUE (the default state after process creation) the notifier may be called.

p_getnotify — Get notify state

```
INT p_getnotify(VOID);
```

Return the notify state for this process.

p_notifyhook — Hook the notifier interface

```
INT p_notifyhook(INT mType);
```

Hook the notifier interface such that this process will get an inter-process message from the notifying process of type mType as a result of the notifying process calling p_notify or p_notifyerr.

Returns zero if successful or E_GEN_FAIL if the notify interface has already been hooked.

The mType message contains an array of 5 string pointers into the notifying process data space corresponding to the parameters to p_notify in the order pT1, pT2, p01, p02 and p03. A process that hooks the notify interface should initialise messaging using p_minit with a value of at least 10 for the message size.

The text strings pointed to by the 5 parameters can be fetched from the notifying process using p_pcpyfr (say using the maximum sizes E_MAX_NOTIFY_TEXT_SIZE and E_MAX_OPTION_TEXT_SIZE). The parameter to p_mfree gives the result of the notification.

A process that has hooked the notify interface should not call either p_notify or p_notifyerr as it would then try and send itself a message, resulting in deadlock.

If the process that has hooked the notify interface terminates, the system automatically frees the notify interface so that another process can hook the interface.

p_notifyunhook — Unhook the notifier interface

```
VOID p_notifyunhook(VOID);
```

Release the notify interface.

Calls p_panic if the caller does not have the notifier interface hooked.

Enter and leave

The functions p_enter and p_leave work together. You use p_enter to call or "enter" a function. If p_leave(err) is called before the entered function returns, the stack is unwound and the call to p_enter

returns err. By convention, a negative value of err indicates an error and a zero (or positive) value signifies an error-free exit.

The call to p_leave (or to a variant such as f_leave) may occur in the entered function or in a sub-function and so on.

The p_leave performs what is sometimes called a "non-local goto" where the address of the goto is defined by the last call to p_enter.

The enter and leave mechanism is commonly used in medium to large interactive applications to implement structured error recovery. When an error occurs, the application has to do the following to recover:

- free any dangling resources (eg free memory cells, close open channels, close screen windows)
- inform the user of the error
- continue

Handling all this with ad hoc conditionals in the code can double the size of a program and makes the code hard to follow. An alternative is to set error state variables, driving centralised clean-up code that is invoked by a negative return from a call to p_enter on an error condition.

The call to p_enter is placed at an appropriate place to continue after the error. The return value indicates the nature of the error (eg no system memory) and a state variable could give a contextual message (eg "while attempting to open file xxx"). Other error state variables would give the handles of resources that should be freed.

p_enter — Enter a function

```
INT p_enter(VOID *pfunc,...);
INT p_enter1(VOID *pfunc);
INT p_enter2(VOID *pfunc, VOID *a1);
INT p_enter3(VOID *pfunc, VOID *a1, VOID *a2);
INT p_enter4(VOID *pfunc, VOID *a1, VOID *a2, VOID *a3);
INT p_enter5(VOID *pfunc, VOID *a1, VOID *a2, VOID *a3, VOID *a4);
INT p_enter6(VOID *pfunc, VOID *a1, VOID *a2, VOID *a3, VOID *a4, VOID *a6);
```

Call function *pfunc where the remaining (up to 5) arguments to p_enter are passed as arguments to *pfunc.

You can either use p_enter, which presents the CDECL calling convention, or one of the p_enter? variants, which use a more efficient register calling convention.

Once a function has been called using p_enter, that function (or any function that is called before *pfunc returns) may call p_leave(ret) to unwind the stack and return prematurely from the call to p_enter with the return value of ret. If p_leave is not called, p_enter returns the value returned by *pfunc.

Functions called with p_enter should return an INT or a UINT (since this is assumed by p_leave). A zero or positive return value is normally taken to indicate that the function completed successfully.

Calls to p_enter may be nested, in which case p_leave returns from the last active p_enter on the stack.

When p_enter calls *pfunc, it passes parameters to it both on the stack and in registers. Neither calling convention[2] is compatible with the default calling convention (which passes parameters in registers) as specified in *p_std.def*.

Because of the above, the target of a p_enter must declare the function as using one of the following calling conventions:

CDECL	where the generated code will take the parameters off the stack
ENTER_CALL	where the generated code will take the parameters from the registers (which is more efficient)

If you also call the entered function directly (without a p_enter) you must ensure that the prototype for the function declares the calling convention consistently.

[2]Note that the calling convention applied to the function called by p_enter has nothing to do with the calling convention of p_enter (or p_enter1 etc) as discussed above.

For example, to declare the target for a p_enter using CDECL:

```
LOCAL_C INT CDECL RunTestProgram(TEXT *name)
    {
    ...
....return(0);
    }
```

or, more efficiently, using ENTER_CALL:

```
#pragma save, ENTER_CALL

LOCAL_C INT RunTestProgram(TEXT *name)
    {
    ...
....return(0);
    }

#pragma restore
```

where, in either case, you would enter the function using, say:

```
    ret=p_enter((VOID *)RunTestProgram,"fred");
```

or, more efficiently:

```
    ret=p_enter2(RunTestProgram,"fred");
```

where ret contains zero if RunTestProgram returned or the parameter passed to p_leave, if p_leave was called.

Since it is impossible to construct a general prototype to cover all cases, pfunc is prototyped as a VOID *. As in the above example, you have to cast the first parameter to p_enter to a (VOID *) to avoid compilation warnings. The header files are organised in such a way that the cast is automatically done when you use one of the fixed parameter p_enter? variants.

Note that the requirement for the target of a p_enter to have one of the two special calling conventions described above means that no PLIB or WLIB function may be the direct target of a p_enter.

p_leave — Unwind stack and return from last p_enter

```
VOID p_leave(INT err);
```

Unwind the processor stack and return from the most recently called p_enter, returning the value err. The function does not return to the caller.

Although in practice err is often a negative error number it need not be and p_leave can legitimately be used to unwind the stack on any kind of condition.

The function calls p_panic if there is no call to p_enter on the stack.

f_leave — Unwind stack and return from last p_enter if error

```
INT f_leave(INT err);
```

Similar to p_leave except that it simply returns err if err>=0. That is, it is equivalent to:

```
GLDEF_C INT f_leave(INT err)
    {
    if (err<0)
        p_leave(err);
    return(err);
    }
```

Although modest in its function, using f_leave rather than p_leave produces smaller executables and makes code more readable. For example, compare:

```
pid=f_leave(p_execc(name,NULL,0));
```

with:

```
pid=p_execc(name,NULL,0);
if (pid<0)
    p_leave(pid);
```

You cannot use f_leave on functions that return an address and fail by returning NULL. However, the more commonly used functions of this type have corresponding f_ variants. For example, p_alloc and p_realloc have the corresponding f_alloc and f_realloc which internally call p_leave(E_GEN_NOMEMORY) rather than return NULL.

Example

```
LOCAL_C INT RunSubProcessWait(TEXT *name)
    {
    HANDLE pid;
    WORD stat;

    pid=f_leave(p_execc(name,NULL,0));
    p_logona(pid,&stat)
    p_presume(pid);
    p_waitstat(&stat);
    if ((stat>>8)!=E_NORMAL_EXIT)
        p_panic(stat);
    return((INT)((BYTE)(stat&0xff)));
    }

LOCAL_C INT CDECL RunTestProgram(TEXT *name)
    {
    ret=RunSubProcessWait(name);
    if (ret)
        p_printf("Program %s failed with reason %d",name,ret);
    return(0);
    }

LOCAL_C VOID RunTestPrograms(TEXT *list)
    {
    TEXT *p;
    INT ret;
    TEXT name[32];

    p=p_scpy(&name[32],"testx")-1;
    while (*p=*list++)
        {
        ret=p_enter((VOID *)RunTestProgram,&name[0]);
        if (ret<0)
            {
            p_atos(&msg[0],"Failed to run %s",&name[0]);
            ret=p_notifyerr(ret,&msg[0],"CONTINUE","ABANDON",NULL);
            if (ret==2)
                p_exit(0);
            }
        }
    }

GLDEF_C INT main(VOID)
    {
    RunTestPrograms("abcd"); /* runs testa.img, testb.img, ... */
    return(0);
    }
```

Note the use of CDECL in the declaration of RunTestProgram.

There are more examples of the use of p_enter and p_leave in the *Files* chapter.

CHAPTER 7

MEMORY ALLOCATION

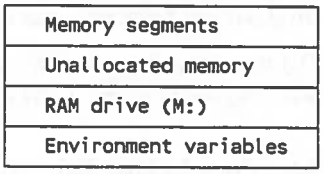
Overview of system memory usage

The EPOC operating system runs on the 8086 processor (and also the 80286, 80386 or 80486) where up to 1Mb of memory may be addressed. On SIBO machines and depending on the model, all or part of this address range may be used where the available memory is allocated (from address zero to `0xffffff`) as follows:

- 1K bytes of interrupt vectors (required by the 8086 architecture)
- the screen bit-map (small display models)
- the operating system data space
- allocated memory segments (including application code segments, process data segments and device driver segments)
- unallocated memory
- the internal RAM drive (LOC::M:)
- environment variables (up to 4K bytes)
- any portion of the 1Mb that is not used
- the screen bit-map (large display models)
- the system ROM (typically 256K bytes)

The contents of the internal RAM drive and the environment variables survive a system reset (unless the ESC key is held down) and most system crashes. See `p_getres` in the chapter *General System Services* for more on system resets.

If we consider only that (greater) part of memory that is dynamically allocated, it is organised into four main sections:



The system maintains all the unallocated memory in a single chunk - between the allocated memory segments and the memory used by LOC::M:. This means that memory segments have to be moved as a result of other segments being created, deleted or having their size changed.

Although application code segments and process data segments can and do move at any time while an application process is running, the operating system automatically adjusts the 8086 segment registers (CS, DS, SS and ES) to follow any movement without explicit support from the application - as discussed in the first chapter of this manual.

As well as giving some background on memory usage, this chapter describes functions that allocate and access:

- memory cells from the heap in the process data segment (which is a dynamic segment)
- memory segments (either device or dynamic)
- environment variables

Memory segments

The allocated memory segments contain both device and dynamic memory segments.

Device segments are created when an external device is installed and are deleted when the device is removed. Once created, a device segment does not normally change its size. The first two device segments are special and are created at system startup for the process data segments of the first two processes to be created - the null process (SYS$NULL) and the supervisor (SYS$MANG). Neither of these two process data segments changes size.

Dynamic segments are much more volatile. Code and data segments are created and deleted as processes are created and terminated. Process data segments change their size to accommodate heap allocations.

Device segments are allocated with a lower address than the dynamic segments so that they are not disturbed by any activity with respect to the more volatile dynamic segments. A device driver normally has to stop working while its segment is moving - which could lead to loss of data (say when receiving data via the serial port).

Allocated memory segments are described by:

- a segment name
- a segment handle
- a segment size
- a segment address

Segment names

Segment names are zero terminated strings of up to eight characters followed by an optional period and up to three further characters (the same rules as for file names). Examples of valid names are as follows:

```
NOTES
NUMBERS.DAT
DATASEG.01
```

As with file names, the segment name extension normally indicates the usage of the segment. The system has the following conventions:

.LDD and .PDD	indicate device segments that contain a Logical Device Driver and a Physical Device Driver respectively. Device drivers are normally written in 8086 assembler. See the *EPOC O/S System Services* reference manual for more about device drivers.
.$SC	indicates the primary shared code segment that is associated with one or more processes. Running say *myprog.img* will cause the code to be loaded into a segment called *myprog.$sc* (provided that *myprog.$sc* isn't already loaded).
.DYL	indicates a dynamic library code segment (DYL) that is associated with one or more processes. See the chapter *Object-Oriented Programming* for more about DYLs.
.$nn	indicate process data segments where nn consists of two decimal digits (01, 02, ...) according to the process number.

The name of a process data segment is normally the same as the process name although they are in fact independently held (the process name can be changed using `p_prename`). The process data segment is described further below.

Segment handle, address and size

The segment handle is actually the relative address of the segment table index entry in the operating system data space. This 16-byte entry contains the address of the segment, the segment usage count and the segment name.

The usage count normally indicates the number of processes that are using the segment - for example, when there are 2 processes of the same application, the usage count of the application code segment is 2 whereas the usage count of each process data segment is 1. When the usage count drops to zero, the memory segment may be (and normally is) deleted.

The order of the segments follows the order of the segment table index entries and the segment size is calculated by subtracting the start addresses of adjacent segments.

The segment index table has a fixed total capacity with fixed sub-capacities for device and dynamic segments (typical limits are 96 total segments as 32 device segments and 64 dynamic segments). A segment allocation will fail if one of these limits is reached (which is unlikely unless there is a bugged application that fails to free segments).

Although the allocated segments themselves are contiguous, the segment index table may have "holes" in it as a result of segment deletion. When a new segment is allocated, it will tend to fill any holes in the index table first and, in this case, the created segment will be inserted before other segments causing them to be moved.

The address and size of memory segments is expressed in 16-byte paragraphs (as used in setting the value of the 8086 segment registers CS, DS, ES and SS). It follows that segments start on 16-byte boundaries and are a multiple of 16 bytes long. From the point of view of the segment allocator, the maximum segment size is 512K which makes it possible to specify a segment size (in 16-byte paragraphs) within a signed 16-bit word.

Process data segments

When a process is created (normally by loading an image using `p_execc` or `p_execcasync`), a process data segment is also created. When that process is running application code, the 8086 DS, SS and ES registers point to the start of the data segment which can not be greater than `0xffe0`[1] bytes long (this is called the *small model* on PCs).

The data segment contains (from low to high address):

- the reserved static variables (`0x40` bytes)
- the floating point emulator data space (`0x200` bytes from offset `0x100`)
- the processor stack
- initialised static variables
- uninitialised static variables
- the process heap

The size of the processor stack depends upon the C startup module that is being used but is typically in the range 2K to 8K. Note that the stack size declared by the startup module includes the 64 bytes of reserved static variables and the floating point emulator data space.

The word at address zero is initialised to `0xDEAD` and is otherwise unused. If it is not `0xDEAD`, it is probably because of a write using an uninitialised pointer that happened to have a zero in it.

In the absence of any further initialisation (such as the `0xDEAD` above), the reserved statics variables are initialised to zero.

The bytes in the stack area are initialised to `0xff`. The number of `0xff` bytes from address `0x40` in the process data segment measures the number of spare bytes on the stack provided the program is not using the floating point emulator. If the program is using the floating point emulator, the lowest point the stack should legitimately reach is `0x300`.

Despite their name, the uninitialised static variables are all initialised to zero.

The process heap contains dynamic data structures that are created and destroyed within the lifetime of the process. The heap is placed at the end of the process data segment so that it can be expanded by expanding the process data segment. Since the data segment is limited to `0xffe0` bytes, the maximum heap size is somewhat smaller than 64K, depending on the size of the stack and the space taken by the static variables. The process heap and the allocator are described in detail next

[1]The segment size is limited to 32 bytes less than the maximum 64K so that a stack underflow will always cause an address trap.

The heap allocator

The allocator is used to allocate, resize and free variable length memory cells (which typically range from 10s of bytes to a few kilobytes in length) from the process heap. Allocated cells are referenced directly by their address; they do not move to compact free space left by freed cells.

The allocator functions are:

`p_alloc, f_alloc`	allocates a cell, returning its address.
`p_free`	frees a cell, which is returned to the heap.
`p_realloc, f_realloc`	changes the size of the cell, returning its new address.
`p_adjust`	opens or closes a gap in the middle of the cell (useful for deletion and insertion of cell content), changing the size of the cell as appropriate.
`p_alen`	returns the size of the cell.
`p_hgran`	sets the heap granularity.
`p_allwalk`	walks all cells in the heap calling a supplied function (used for heap diagnosis).
`p_allchk`	uses `p_allwalk` to check the integrity of the heap.
`p_allspc`	gets the start address and free space in the heap.

The functions `f_alloc` and `f_realloc` are identical to `p_alloc` and `p_realloc` respectively except that they call `p_leave(E_GEN_NOMEMORY)` if the memory could not be found rather than returning a `NULL` address. See the function `p_leave` for more details.

The allocator functions are used internally by many other PLIB functions.

Heap structure

After a number of allocate and free calls the heap typically consists of ranges of adjacent allocated cells separated by single free cells (which are linked). Each cell (whether free or allocated) is prefixed by a hidden 16-bit word that gives the size of the cell in bytes. This leading length word is hidden since the address returned by `p_alloc`, `p_realloc` and `p_adjust` skips this header and the length returned by `p_alen` does not include it.

Writing beyond a cell's limits will corrupt a cell length word (and possibly also a free space cell pointer) which destroys the heap's integrity. Such errors are difficult to debug because there is no immediate effect - the corruption is a "time bomb". It will eventually be detected (resulting in a call to `p_panic`) by a subsequent allocator call (such as `p_free`). The `p_allchk` function is provided as a debugging tool to force a call to `p_panic` sooner rather than later when it is suspected that the heap's integrity has been damaged.

Growing and shrinking the heap

In EPOC, a process heap is not fixed in size - the system can grow and shrink the heap (and the process data segment that contains it). The heap is grown to satisfy allocation requests that would otherwise fail, up to the `0xffe0` process data segment limit. The heap may be shrunk to release memory to the system when it is required. When the heap is grown, it is normally grown by 2K bytes more than is strictly needed to satisfy the request - see `p_hgran` for more details. When an application is executed to create a process, the initial size of the heap is taken from a value that is stored in the executable (by default, 2K bytes). This same value also specifies the minimum size of the heap.

Allocation is based on "walking" the free space list to find a free cell that is big enough to satisfy the request (using the "first fit" algorithm). If no free cell is big enough, the system will attempt to grow the data segment to add more free space at the end of the heap.

If there is no memory in the system to accommodate growth or if the data segment has reached its maximum `0xffe0` byte value, the allocate request fails. There are few circumstances when an allocate request can be assumed to succeed and calls to `p_alloc`, `p_realloc` and `p_adjust` should have recovery code to handle a failure to allocate.

From the point of view of the user of an application, the two causes of an allocation failure produce quite different situations[2]. If there is no memory left in the system, this can normally be remedied by the user taking some action to release memory (such as exiting a task). If the 64K limit has been reached, this is

[2]Unfortunately, the system does not distinguish between the two causes of allocation failure.

presumably as a result of a heap-based data structure reaching its design limit (rather than a bug as in, for example, "alloc heaven", described below).

Applications that contain indefinitely growing heap-based data structures (as in, say, a spreadsheet) should not allow the data structure to grow until the allocation fails when the `0xffe0` data segment limit is reached because, at this limit, there may not be sufficient memory in the free space list to perform other tasks (such as saving the data to file!). The growth of such data structures should be monitored by the application and limited to leave sufficient heap capacity for tasks that the user would reasonably expect to be able to perform.

The segment allocator can reclaim excess space from a heap if the last cell in the heap is a free cell, reducing the size of the last free cell to a small value. However, the heap is never shrunk below its initial size (as specified by the value in the executable).

Alloc heaven

There are cases in which programs allocate a sequence of cells which must either exist as a whole or not at all. If during the allocate sequence one of the later allocations fail, the previously allocated cells must be freed. If this is not done, the heap will contain unreferenced cells that consume memory to no purpose. At Psion we say that these cells have gone to "alloc heaven".

When designing how to physically organise data structures into alloc cells you should be mindful of the recovery code that must be written to free partially built multi-cell structures. The fewer the cells in a structure, the easier the recovery code.

Internal fragmentation

The free space in a heap is normally fragmented where the largest cell that may be allocated is substantially smaller than the total free space. Excessive fragmentation, where the free space is distributed over a large number of cells (and where by implication many of the free cells are small) should be avoided because it results in an inefficient use of memory and reduces the speed with which cells are allocated and freed. Practical design hints for limiting internal fragmentation are:

- Avoid using the heap for small highly transient data structures that can be placed on the stack (as an automatic). High frequency cycling through allocate and free pairs "churns" the heap and leads to a long free space list.
- When you have a large number of variable length data structures (particularly when they are frequently resized), "granularise" them (i.e. round the allocate up to a multiple of some reasonable value) so that you decrease the chance of leaving small unusable free space cells.
- Use heap analysis tools to give you a feel for what is going on. You may find that by changing the way you do something you get a better heap and you may even discover some alloc heaven. Don't go too far - there are diminishing returns to heap usage tuning.

p_alloc (or f_alloc) — Allocate a memory cell

```
VOID *p_alloc(UINT size);
VOID *f_alloc(UINT size);
```

Allocate a memory cell of at least `size` bytes long from the heap and return the address of the allocated cell or `NULL` if there is insufficient memory.

You should always test the result for `NULL` and take recovery action. The size actually allocated may be a few bytes more than that requested (see `p_alen`). The maximum size is 64K minus the combined size of the machine stack and the space taken by static variables.

If the heap is corrupt, calling `p_alloc` may or may not detect it - but if it does it will call `p_panic`. Use `p_allchk` to check the integrity of the heap thoroughly.

The function `f_alloc` is identical except that it calls `p_leave(E_GEN_NOMEMORY)` rather than return `NULL`.

Example

```
GLDEF_C TEXT *AllocString(TEXT *str)
/*
Allocate and copy in a zero terminated string.
*/
    {
    TEXT *p;

    if (p=p_alloc(p_slen(str)+1))
        p_scpy(p,str); /* Copy in the string */
    return(p);
    }
```

This example assumes that any more specific error recovery is handled by the caller.

p_free — Free an allocated cell

```
VOID p_free(VOID *pcell);
```

Free the allocated memory cell at address `pcell`, returning the cell to the free memory list. Does nothing if `pcell` is zero - this is sometimes useful in error clean-up situations.

If pcell is non-zero, it should contain the address of a cell as returned by, for example, `p_alloc`. Passing a value that is not the address of an allocated cell (eg by freeing a cell twice) will corrupt the heap. There is a chance that `p_free` will detect a bad address and call `p_panic`.

p_realloc (or f_realloc) — Change cell size

```
VOID *p_realloc(VOID *pcell, UINT size);
VOID *f_realloc(VOID *pcell, UINT size);
```

Change the size of the allocated cell `pcell` to be `size` bytes and return the address of the new cell or `NULL` if there was insufficient space for the size change.

If `NULL` is returned, the original cell is unaffected. The limits on size are as for `p_alloc`. Calling `p_realloc(pcell,size)` when pcell is zero is equivalent to calling `p_alloc(size)` - this can be useful in start up situations.

The cell retains its original content which is truncated if the cell size is reduced. The cell start address does not change when the cell size is reduced or stays the same. If the cell size is increased, `p_realloc` will use any trailing free space of sufficient size but, more likely, it will allocate a new cell, copy the data across and free the old cell where, in this case, the returned address is different from `pcell`. If `pcell` is neither zero nor the address of an allocated cell, the heap will either be corrupted or `p_panic` will be called.

The function `f_realloc` is identical except that it calls `p_leave(E_GEN_NOMEMORY)` rather than return `NULL`.

p_adjust — Insert or delete data in cell

```
VOID *p_adjust(VOID *pcell, UINT offset, INT amount);
```

Open or close a gap at offset `offset` within the allocated cell `pcell`, using `p_realloc` to make the appropriate change to the cell size. As for `p_realloc`, `p_adjust` returns the address of the new cell or `NULL` if there was insufficient memory. If `amount` is positive, a gap of `amount` bytes is opened. If `amount` is negative `-amount` bytes are deleted. If amount is zero, the function has no effect and returns `pcell`.

Unlike `p_realloc`, `pcell` may not be passed as `NULL`. If `pcell` is not the address of an existing allocated cell, the heap will either be corrupted or `p_panic` will be called.

If `amount` is negative, `-amount` bytes is deleted by shifting the trailing contents left, closing the gap.

```
old cell    xxxxxxxxxxzzzzzzzzzzyyyyyyyyyyyy
            < offset >< amount >

new cell    xxxxxxxxxxyyyyyyyyyyyy
            <  old size - amount  >
```

The cell size is decreased by the same amount. There is no data deletion if the offset is greater than or equal to the original cell size. The minimum cell size actually allocated is as for p_alloc. If the new size would be negative, p_panic is called.

If amount is positive, the cell size is increased and a gap is then inserted starting from the specified offset by shifting the trailing contents right.

```
old cell    xxxxxxxxxxyyyyyyyyyyyy
            < offset >< amount >

new cell    xxxxxxxxxxzzzzzzzzzzyyyyyyyyyyyy
            <     old size + amount         >
```

If offset is greater than or equal to the original cell size, no shifting takes place.

A typical use is to insert or delete a record (which may be of fixed or variable length) in a contiguous sequence of records contained in an alloc cell.

p_alen — Get cell length

```
UINT p_alen(VOID *pcell);
```

Return the length in bytes of the allocated cell pcell.

The returned cell length will be equal to or slightly larger than that requested using p_alloc or p_realloc because (1) sizes are rounded up to an even size, (2) there is a minimum cell size and (3) the allocation can't leave a trailing free space cell below the minimum size.

If the passed address is not the address of a cell, there is a chance that p_alen will detect it and call p_panic.

p_hgran — Set heap granularity

```
VOID p_hgran(UINT nparas);
```

Set the heap granularity to nparas (measured in paragraphs where a paragraph is 16 bytes).

The maximum value for nparas is E_MAX_GROWBY (16K bytes) - p_hgran calls p_panic if this is violated. Processes are created with heap granularity E_GROWBY_DEFAULT (2K bytes). E_MAX_GROWBY and E_GROWBY_DEFAULT are defined in *epoc.h*.

The heap granularity controls the increment by which the heap grows to satisfy an allocation request that cannot be met from the existing free space list. The granularity is added to the amount that is just sufficient to satisfy the request. Growing the heap can be computationally expensive because many other segments may have to be shifted by the growth of the process data segment. Applications that can rapidly create large data structures (such as when loading a large file in the text processor) can improve their performance by setting a larger heap granularity. Setting the heap granularity has no affect on the way the system shrinks the heap to release memory.

p_allwalk — Visit all cells

```
VOID p_allwalk(VOID (*fptr)(VOID *fpar,INT isalloc,UINT len), VOID *fpar);
```

Walk through every cell in the heap (whether allocated or free) in sequence from low to high address and, if fptr is not NULL, call fptr for each cell. The heap is checked for consistency and p_panic is called if an inconsistency (for example an overlap between a free cell and an allocated cell) is detected.

In the call to fptr, the parameter isalloc is TRUE if the cell is an allocated cell and FALSE if it is a free cell. The parameter len is the total length of the cell in bytes (len includes the size of cell header information and is 2 greater than that returned by p_alen).

Cell addresses can be calculated from the cell lengths and the heap start address. The heap start address may be found by calling p_allspc.

This function is provided for heap diagnosis and is called, for example, by p_allchk.

In the following example, NumAllocCells returns the number of cells allocated:

```
LOCAL_C VOID CountIfAlloc(UINT *pn,INT isalloc,UINT len)
    {
    if (isalloc)
        *pn+=1;
    }

GLDEF_C UINT NumAllocCells()
    {
    UINT n;

    n=0;
    p_allwalk((VOID (*)(VOID *,INT,UINT))CountIfAlloc,&n);
    return(n);
    }
```

The call to p_allwalk calls back CountIfAlloc (which simply increments the allocated cell count if the cell is an allocated cell) for each cell in the heap.

p_allchk — Check heap integrity

```
VOID p_allchk(INT num);
```

Walk through all the allocated cells checking for consistency with the free space list. Call p_panic(0xff) if there is something wrong.

Used as a debugging aid to detect errors from freeing a cell twice or overwriting the boundaries of an allocated cell. Otherwise, such errors can remain undetected for some time.

If it does discover something wrong, and before calling p_panic, it writes 3 words of diagnostic information to reserved statics, as follows:

DatApp1 (0x28)	a reason code, described below
DatApp2 (0x2a)	the address at which the corruption was discovered
DatApp3 (0x2c)	the passed parameter num, to enable identification of the offending call

The reason code is one of:

5	a free cell pointer was probably overwritten
6	a cell length is too small or odd (possibly because it was overwritten)
7	an allocated cell length is too large (possibly because it was overwritten)

This information is only useful in conjunction with a debugging tool that allows the process data segment to be examined after a call to p_panic.

p_allspc — Get heap address and potential free space

```
UINT p_allspc(VOID **pheap);
```

Return the potential free space in the heap in bytes and writes the start address of the heap to *pheap.

The potential free space is calculated as the sum of the free cells in a process data segment that has been expanded to its full size of 0xffe0 bytes. In practice, the amount that can be allocated will certainly be less than this and depends upon how fragmented the heap is and on the amount of free memory in the system. This function should not be used to predict a successful allocation but it can be used to predict an unsuccessful one.

The heap start address *pheap can be used to turn the cell lengths produced by p_allwalk into addresses.

System memory usage

p_getram — Get addressable system RAM size

```
UINT p_getram(VOID);
```

Return the size of the addressable system RAM in paragraph (16 byte) units.

Since the memory in machines containing more than 512 kilobytes of RAM is bank-switched, this function will never return a value larger than 32768 (corresponding to 512 kilobytes of RAM).

p_totalK — Get total system RAM size

```
UINT p_totalK(VOID);
```

This function is only available in EPOC version 3.50 or later.

Return the total amount of memory in the machine in kilobytes.

This function reports the total amount of RAM present in the machine, irrespective of bank-switching. Thus, on a machine containing 1 megabyte of RAM, a call to `p_totalK` will return the value 1024, whereas a call to `p_getram` on the same machine will return 32768 (corresponding to 512 kilobytes).

p_sgfree — Get size of available segmented memory

```
UINT p_sgfree(VOID);
```

Returns the amount of available (i.e. currently unused) addressable segmented memory in paragraph (16 byte) units.

Since the memory in machines containing more than 512 kilobytes of RAM is bank-switched, this function will never return a value larger than 32768 (corresponding to 512 kilobytes of RAM).

The value returned should be treated with some caution as the amount of available memory in a multi-tasking environment is a dynamic function of the memory requests of all the currently running processes.

p_sgramdisk — Get memory used by internal RAM disk

```
UINT p_sgramdisk(VOID);
```

Returns the number of 16-byte paragraphs in the addressable RAM that are currently used by the internal RAM disk (LOC::M:).

On machines containing more than 512 kilobytes of RAM, the RAM disk will be created in an upper bank. In this case, unless the RAM disk overflows into the addressable RAM, `p_sgramdisk` will generally return zero.

On machines containing not more than 512 kilobytes of RAM, the return value will be larger than that obtained from `p_dinfo`, which measures storage capacity rather than the total number of bytes used.

Memory segments

Applications that need more memory to store data, can allocate one or more external memory segments. Each segment can be up to 512K bytes long, subject to the availability of free system memory.

The most common use of the functions described in this section is to implement a potentially large data structure without being constrained by the 64K (or less) limit of the heap. To create and access a data structure in an external data segment, you would use:

`p_sgcreate`	to create an external data segment of a specified initial size
`p_sgcopyto`	to write to the external segment
`p_sgcopyfr`	to read from the external segment

p_sgadjust	to adjust the size of the data segment (say to increase its size to accommodate additional content)
p_sgdelete	to delete the data segment after it is no longer required

Memory segments can be used to implement a data structure that is accessed by more than one process. For example, a "pipe" in which one process creates a data segment and writes to it and where a second process reads from it. In this case, the second process would use:

p_sgfind	to locate the pipe by its memory segment name
p_sgopen	to open the segment
p_sgcopyfr	to read from the segment
p_sgclose	after finishing with the segment

You would need to use an associated semaphore to synchronise access to the segment (as described in the chapter *Asynchronous Requests and Semaphores*).

If a memory segment is locked by a process calling p_sglock, the segment will survive the demise of the creating process

p_sgcreate — Create memory segment

```
HANDLE p_sgcreate(TEXT *pName, INT nParas, INT uMode);
```

Creates a memory segment with the zero terminated name pName and size nParas (in 16-byte paragraphs) and, if successful, return the positive handle to the created segment. Otherwise, it returns one of the following negative error numbers:

E_GEN_NOMEMORY	Not enough memory to satisfy the request
E_GEN_NOSEGMENTS	No memory segment handles are available
E_FILE_EXIST	A memory segment of the requested name already exists
E_FILE_NAME	The requested name is invalid

The memory segment created is not initialized in any way and will contain random data. The returned handle allows access to the contents of the segment using p_sgcopyto and p_sgcopyfr.

After being created, the memory segment is automatically opened and given a usage count of 1. It should be closed when no longer required by calling p_sgclose.

When a process terminates, any open memory segment is automatically closed, decrementing the segment usage count. If the access count become zero or negative, the segment is deleted.

The initial size nParas (in 16-byte paragraphs) must be positive (so the maximum segment size is 512K bytes).

The parameter uMode should be one of the following:

E_SEGMENT_HIGH	to create a dynamic memory segment.
E_SEGMENT_LOW	is provided for future expansion and currently has the same effect as E_SEGMENT_HIGH.
E_SEGMENT_DEVICE	to create a device segment. This will result in all devices being held while memory is moved and then resumed. All dynamic segments will be moved up in memory to make room. This service is called, for example, by the File Server when loading external device drivers and should not be used by applications.
E_SEGMENT_LOCKED	is the same as E_SEGMENT_HIGH except that no process owns the created segment. Like E_SEGMENT_DEVICE this mode is used internally by the operating system and should not be used by applications.

The function calls p_panic if the requested size was negative or if uMode was not one of E_SEGMENT_LOW, E_SEGMENT_HIGH, E_SEGMENT_DEVICE or E_SEGMENT_LOCKED.

p_sgdelete — Delete memory segment

```
INT p_sgdelete(TEXT *pName);
```

Delete the memory segment identified by the zero terminated name `pName`. Returns zero if successful or one of the following negative error numbers:

`E_GEN_INUSE`	the segment usage count is greater than zero (eg because it is opened by another process)
`E_FILE_NXIST`	the memory segment does not exist
`E_FILE_NAME`	the memory segment name is invalid

p_sgopen — Open memory segment

```
HANDLE p_sgopen(TEXT *pName);
```

Open the memory segment identified by the zero terminated name `pName` and, if successful, return the handle to the opened memory segment. Otherwise, it returns one of the following error numbers:

`E_FILE_NXIST`	the memory segment does not exist
`E_GEN_OPEN`	the memory segment is already open to this process
`E_FILE_NAME`	the memory segment name is invalid

The returned handle allows access to the contents of the segment using `p_sgcpto` and `p_sgcpfr`.

Opening a segment increments the segment usage count.

When a process terminates, any open memory segment is automatically closed, decrementing the segment usage count. If the access count becomes zero or negative, the segment is deleted.

There is no limit on the number of memory segments that may be opened by a process.

p_sgcopyto — Copy to memory segment

```
INT p_sgcopyto(HANDLE nHandle, LONG pos, VOID *source, UINT len);
```

Copy `len` bytes from `source` in the current process data segment to offset `pos` in the open memory segment `nHandle` (as returned from `p_sgcreate` or `p_sgopen`).

The system ensures that the copy is not interrupted by another process. Address trapping is automatically switched off for the duration of the copy.

The return value has no significance.

The function calls `p_panic` if `pos+len` is greater than the size of the memory segment or if `nHandle` is not a valid memory segment handle.

A program can check that the segment is still in existence before a call to `p_sgcopyto` by calling `p_sgopen`.

If you want to write to a process data segment, it is more convenient to use `p_pcpyto`, which takes a process ID rather than a segment handle.

p_sgcopyfr — Copy from a memory segment

```
INT p_sgcopyfr(HANDLE nHandle, LONG pos, VOID *target, UINT len);
```

Copy `len` bytes from offset `pos` in the open memory segment `nHandle` (as returned from `p_sgcreate` or `p_sgopen`) to `target` in the current process data segment.

The system ensures that the copy is not interrupted by another process.

The return value has no significance.

The function calls `p_panic` if `pos+len` is greater than the size of the memory segment or if `nHandle` is not a valid memory segment handle.

A program can check that the segment is still in existence before a call to `p_sgcopyfr` by calling `p_sgopen`.

If you want to copy from a process data segment, it is more convenient to use p_pcpyfr, which takes a process ID rather than a segment handle.

p_sgsize Get size of memory segment

```
UINT p_sgsize(HANDLE nHandle);
```

Returns the size (in 16-byte paragraphs) of the open memory segment nHandle (as returned from p_sgcreate or p_sgopen).

The function calls p_panic if nHandle is not a valid memory segment handle.

p_sgadjust Adjust the size of a memory segment

```
INT p_sgadjust(HANDLE nHandle, INT nParas);
```

Adjust the size of the open memory segment nHandle (as returned from p_sgcreate or p_sgopen) to nParas 16-byte paragraphs. Return zero if successful or the negative E_GEN_NOMEMORY if there is not enough memory to satisfy the request.

The new segment size nParas must be positive (it follows that the maximum segment size is 512K bytes). Setting nParas to zero discards all the memory allocated to the memory segment but does not delete the segment.

The function calls p_panic if nParas is negative or if nHandle is not a valid memory segment handle.

p_sgfind Find segments by name

```
HANDLE p_sgfind(HANDLE fHandle, TEXT *pMatch, TEXT *pName);
```

Write the next segment name that matches the zero terminated match string pMatch as a zero terminated string to pName where fHandle is NULL for the first call and is subsequently the positive return value from the previous call. When there are no further segments matching pMatch, it returns E_FILE_NXIST.

Used repeatedly to find all the segments that match the wild card string pointed to by pMatch. The buffer at pName should be big enough to receive E_MAX_NAME+2 bytes. The wild card string pMatch should remain the same between successive calls.

No memory is used by this service and it can be abandoned at any time without taking any further action.

The function calls p_panic if fHandle is not a valid memory segment handle.

Example

```
GLDEF_C VOID ListCodeSegments(VOID)
    {
    HANDLE fH;
    TEXT buf[
E_MAX_NAME+2];

    fH=NULL;
    while ((fH=p_sgfind(fH,"*.$SC",&buf[0]))>0)
        p_puts(&buf[0]);
    }
```

p_sgclose Close memory segment

```
INT p_sgclose(HANDLE nHandle);
```

Close the open memory segment nHandle (as returned from p_sgcreate or p_sgopen) decrementing the usage count.

Returns zero if successful or the negative E_GEN_NOTOPEN if the memory segment is not open to this process.

Should the access count become zero or negative, the segment is deleted.

The function calls p_panic if nHandle is not a valid memory segment handle.

p_sglock — Increment segment usage count

```
VOID p_sglock(HANDLE nHandle);
```

Lock the open memory segment nHandle (as returned from p_sgcreate or p_sgopen) by incrementing the segment usage count.

If p_sglock is called after creating a segment, the segment will not be deleted when the process terminates.

The function calls p_panic if nHandle is not a valid memory segment handle.

p_sgunlock — Decrement segment usage count

```
VOID p_sgunlock(HANDLE nHandle);
```

Unlock the open memory segment nHandle (as returned from p_sgcreate or p_sgopen) by decrementing the segment usage count.

There is no harm in unlocking a segment that is already unlocked, although if this done inadvertently the segment could be deleted by another process.

The function calls p_panic if nHandle is not a valid memory segment handle.

Environment variables

The system allocates up to 4K bytes for environment variables at the high address end of the system RAM. Environment variables are a scarce resource and should be used sparingly.

An environment variable consists of:

a name	of up to E_MAX_ENV_SIZE (16) bytes containing any byte except '*' or '?'
a value	of up to P_ENVMAX-1 (256) bytes with no restriction on the content

Each environment variable is stored as two successive leading byte count strings (see the example in the description of p_findenviron, below).

If you are dealing with environment variables where both the names and the values are character strings you can use:

p_getenv	to get the value of an environment variable
p_setenv	to replace the value of an existing environment variable or to create one if necessary
p_delenv	to delete an environment variable
p_fndenv	to get a list of environment variables and their values

A more general but less convenient set of environment variable functions (which can be used, for example, when the values are binary) are:

p_getenviron	to get the value of an environment variable
p_setenviron	to replace the value of an existing environment variable or to create one if necessary
p_delenviron	to delete an environment variable
p_findenviron	to get a list of environment variables and their values

p_getenv — Get environment variable value

```
INT p_getenv(TEXT *pMatch, TEXT *pValue);
```

Copy the value of the environment variable that matches the zero terminated name pMatch to pValue and add a zero terminator to the end of the copied value. The name pMatch may include the wild card characters '?' and '*' in which case the value of the first matching name is copied.

Returns zero if successful or the negative E_FILE_NXIST if no matching environment variable exists.

The maximum length of an environment variable value is P_ENVMAX-1 bytes so up to P_ENVMAX bytes can be written to pValue (including the zero terminator).

p_getenviron — Get environment variable value

```
INT p_getenviron(TEXT *pMatch, INT mLength, VOID *pValue);
```

Copy the value of the environment variable that matches the name pMatch of length mLength to pValue and return the number of bytes copied.

Return the negative E_FILE_NXIST if no matching environment variable exists.

The name pMatch may include the wild card characters '?' and '*' in which case the value of the first matching name is copied.

The maximum length of an environment variable value is P_ENVMAX-1 bytes.

p_setenv — Set environment variable value

```
INT p_setenv(TEXT *pName, TEXT *pValue);
```

Copy the content of the zero terminated string pValue (excluding the terminating zero) into the value of the environment variable with the zero terminated name pName, replacing any previous value. If the environment variable does not exist it is created with the specified name and value.

The name pName may not include wild cards and must not exceed E_MAX_ENV_SIZE in length. The length of pValue should not exceed P_ENVMAX-1 (255) - any non-zero value in the high byte of the length of pValue is ignored.

Returns zero if successful or one of the following negative error numbers:

E_GEN_NOMEMORY	the system was unable to allocate room to store the environment variable and its value either because there is not enough free system memory or because of the 4K limit on the environment variable space
E_GEN_FAIL	if pName contains a wild card character

The function calls p_panic if the length of pName exceeds E_MAX_ENV_SIZE.

Note that you can't delete an environment variable by giving it a null value - use either p_delenv or p_delenviron.

p_setenviron — Set environment variable value

```
INT p_setenviron(TEXT *pName, INT nLength, VOID *pValue, INT vLength);
```

Copy vLength bytes from pValue into the value of the environment variable with name pName of length nLength, replacing any previous value. If the environment variable does not exist it is created with the specified name and value.

The name pName may not include wild cards, nLength must not exceed E_MAX_ENV_SIZE and vLength should not exceed P_ENVMAX-1 (255) - any non-zero value in the high byte of vLength is ignored.

Returns zero if successful or one of the following negative error numbers:

E_GEN_NOMEMORY	the system was unable to allocate room to store the environment variable and its value either because there is not enough free system memory or because of the 4K limit on the environment variable space
E_GEN_FAIL	if pName contains a wild card character

The function calls p_panic if nLength exceeds E_MAX_ENV_SIZE.

p_delenv — Delete environment variable

```
INT p_delenv(TEXT *pMatch);
```

Delete the environment variable that matches the zero terminated name pMatch. The name pMatch may include the wild card characters '?' and '*' in which case the first matching environment variable is deleted.

Returns zero if successful or the negative E_FILE_NXIST if no matching environment variable exists.

p_delenviron — Delete environment variable

```
INT p_delenviron(TEXT *pMatch, INT mLength);
```

Delete the environment variable that matches name pMatch of length mLength. The name pMatch may include the wild card characters '?' and '*' in which case the first matching environment variable is deleted.

Returns zero if successful or the negative E_FILE_NXIST if no matching environment variable exists.

p_fndenv — Find environment variables

```
INT p_fndenv(TEXT *pMatch, TEXT *pName, TEXT *pValue, HANDLE *pHandle);
```

Called repeatedly to find the name and value of all environment variables that match the zero terminated wild card name pMatch.

On the first call *pHandle should contain zero and subsequent calls pass the value that is written by the previous call. The function returns zero if a matching environment variable was found or the negative E_FILE_EOF when there are no more matching names. The wild card match string pMatch should remain the same on successive calls.

Each successful call writes the name as a zero terminated string to pName (which should have room for E_MAX_ENV_SIZE+1 bytes) and its value followed by a zero terminator to pValue (which should have room for P_ENVMAX bytes).

A wild card name of "*" will match all the environment variables.

p_findenviron — Find environment variables

```
INT p_findenviron(TEXT *pMatch, INT mLength, UBYTE *pBuf, HANDLE *pHandle);
```

Called repeatedly to find the name and value of all environment variables that match the wild card name pMatch of length mLength.

On the first call *pHandle should contain zero and subsequent calls pass the value that is written by the previous call. The function returns zero if a matching environment variable was found or the negative E_FILE_EOF when there are no more matching names. The wild card match string should remain the same on successive calls.

Each successful call writes the name and value to pBuf as two successive leading byte count strings giving the environment variable name followed by its value. The maximum length of an environment variable name is E_MAX_ENV_SIZE and the maximum length of a value is P_ENVMAX-1 bytes so pBuf should have room for E_MAX_ENV_SIZE+P_ENVMAX+1 bytes.

A wild card name of "*" will match all the environment variables.

In the following example, PrintAllEnv prints the name and (potentially binary) value of all the environment variables:

```
LOCAL_C VOID PrintData(TEXT *p,UINT len)
    {
    UBYTE *pe;

    p_print("%d [",len);
    for (pe=p+len;p<pe;p++)
        p_isprint(*p) ? p_print("%c",*p) : p_print("<%02x>",*p);
    p_printf("]");
    }
```

```
GLDEF_C VOID PrintAllEnv(VOID)
    {
    UBYTE *p;
    HANDLE h;
    UBYTE b[E_MAX_ENV_SIZE+P_ENVMAX+1];

    for (h=0;p_findenviron("*",1,&b[0],&h)>=0;)
        {
        p=&b[0];
        PrintData(p+1,*p); /* print name */
        p+=*p+1;
        PrintData(p+1,*p); /* print value */
        }
    }
```

CHAPTER 8

ASYNCHRONOUS REQUESTS AND SEMAPHORES

This chapter describes asynchronous requests, semaphores, the I/O semaphore and wait handlers.

Semaphores

Semaphores are provided to synchronise cooperating processes (where, in this context, a process includes a hardware interrupt). There are three common uses:

- Synchronising access to a shared resource
- Synchronising supplier-consumer relationships
- Synchronising the completion of asynchronous requests

The first two are described briefly in this section. The third use is far more important and is discussed more extensively in the following section.

The semaphores in EPOC are counting semaphores, having a signed value that is incremented by calling `p_signal` and decremented by calling `p_wait`. A semaphore with a negative value implies that a process must wait for the completion of some other event, such as the freeing of a shared resource.

The mechanism by which a process waits on a semaphore is part of the overall management of process scheduling.

Process scheduling

In EPOC, if a process is not the currently running process, it is either suspended or in a queue.

A process remains suspended until it is resumed by another process (typically its creator) by that process calling `p_presume`.

If a process is not suspended, it is in one of the following three types of queues:

The ready queue	The ready queue contains processes ordered by process priority. On a reschedule, the process at the high priority end of the ready queue runs. If there is more than one ready process at the highest priority, they take it in turns to run every 4 system ticks.
The time delta queue	There is a single (possibly empty) time delta queue, effectively containing both processes and timer device entries. The processes are waiting for a relative or an absolute time as a result of calling `p_sleep`, `p_sleept` or `p_sleepa`. The timer device entries are associated with processes that have requested an asynchronous timer. The head of the queue has its delta time decremented every system tick and is removed when it reaches zero or negative. If it is a process, it is inserted into the ready queue. If it is a timer device entry, the associated process I/O semaphore is signalled.
The semaphore queues	There is a (possibly empty) queue of processes for each created semaphore in the system. If, on calling `p_wait`, the decremented semaphore is negative, the calling process is placed at the end of the appropriate semaphore queue. When that semaphore is subsequently incremented by a call to `p_signal`, the process at the head of the semaphore queue is either made current (if it has the highest priority) or it is inserted into the ready queue (after any processes of equal priority).

See also the description of *Preemptive scheduling* in the chapter *Processes and Inter-process Messaging*.

Shared access

A mutual exclusion semaphore may be used to serialise access to a shared resource (for example, a shared memory segment).

The creator of the shared resource uses `p_semcrt` to create an associated mutual exclusion semaphore with an initial value of one. Any process wishing to access the resource first calls `p_wait` on the resource semaphore and then calls `p_signal` after completing its access. Because the semaphore was created with an initial value of one, the first process to call `p_wait` will return immediately but any other processes that call `p_wait` will wait in the semaphore queue. Waiting processes are released on a first-in first-out basis when the process currently accessing the resource calls `p_signal`.

The nature of the shared resource should be such that any access completes in a relatively short time so that processes do not wait for extended periods on the mutual exclusion semaphore. Another consideration is that whereas the resource may survive the demise of its creating process, semaphores are automatically deleted when the creating process terminates. In many cases, you may find that EPOC is better suited to supporting the use of server processes for serialising access to a shared resource (as in, for example, the file server and the window server) rather than using a mutual exclusion semaphore. The design of EPOC's inter process messaging was largely driven by the requirements of server processes and their clients.

Supplier-consumer

In this usage, the semaphore is associated with a pool of data (say a circular list in a shared data segment) and is created with an initial value equal to the number of elements in the pool. The consumer calls `p_wait` when it is ready to extract an item from the pool and the supplier calls `p_signal` after it inserted an item into the pool. If there are one or more elements in the pool, the consumer's call to `p_wait` returns immediately. Otherwise, the call to `p_wait` returns when the supplier inserts an element and calls `p_signal`.

Asynchronous requests

Many system services are implemented in two steps:

- make the service request
- wait for the requested operation to complete

In most cases, as well as providing functions for each step, the system provides a function containing both the above steps. Such functions are called *synchronous* because they automatically synchronise the requesting process by waiting until the operation has completed. The internal function that makes the request without waiting for completion is called an *asynchronous* function.

Examples of asynchronous request functions are:

`p_ioa, p_ioc`	for requests on an open I/O channel
`p_mreceive`	to receive an inter process message
`p_execcasync`	for loading images
`p_logona`	for being informed of a process termination

There are also synchronous versions of all the above functions except `p_logona` (but see the example below).

Applications use asynchronous requests in situations like the following:

- make request A
- make request B
- wait for either of the requested operations to complete

Processes wait for the completion of asynchronous requests by waiting on their *I/O semaphore* where each request is associated with a *status word*.

The I/O semaphore

When a process is created, the system automatically creates an *I/O semaphore* on its behalf (a more accurately descriptive name would have been the *asynchronous request semaphore*). After making one or more asynchronous requests, a process calls `p_iowait` to wait on the I/O semaphore for one of the requests to complete. A typical application process spends most of its time waiting on its I/O semaphore. For example, an interactive application process that is waiting for user input from the window server is waiting on the I/O semaphore.

The process or the hardware interrupt handler that implements the requested operation typically uses `p_iosignalbypid` to indicate that the operation has completed. If one or more wait handlers have been installed (wait handlers are described below), they may process the signal and re-signal using `p_iosignal`. In some cases, it is convenient for the requestor to use `p_iosignal` to signal itself and to subsequently process that signal in a central call to `p_iowait`.

Status words

Although the parameters to asynchronous request functions vary, they all take the address of a signed 16-bit status word which subsequently contains the status of the requested operation.

All asynchronous requests exhibit the following behaviour:

- While the request is pending, the status word contains the negative `E_FILE_PENDING` (defined in *p_file.h*)
- When the operation has completed, a value other than `E_FILE_PENDING` is written to the status word. This value should be zero or positive to indicate success or a negative error number to indicate failure.
- The requesting process's I/O semaphore is signalled (after the status word has been written).

Making a request while a previous request on the same status word is still pending will normally result in a call to `p_panic`.

When there are multiple requests, each request is associated with a different status word. After returning from `p_iowait`, the caller typically *polls* each status word until one is found that contains other than `E_FILE_PENDING`. That completion is then processed (which might include renewing the asynchronous request) and `p_iowait` is called again to process the next completion.

Every `p_iosignal` should be matched by a call to `p_iowait` (or a function that calls `p_iowait`). The status word associated with the `p_iowait` must have completed (ie must contain a value other than `E_FILE_PENDING`) at the time that `p_iosignal` is called. A common programming error is to introduce a `p_iosignal` without correctly associating it with a status word, such that the poll after the `p_iowait` cannot find a completed status word.

At Psion this is known as a "stray signal" and programmers should detect this as early as possible by (say) calling `p_panic` when the poll is unable to find a completed status word.

For Series 3, Series 3a and Work*about* developers, using the Spy application supplied with this SDK may help identify an accumulation of such unused signals.

In the following example, the opened asynchronous timer `TimerChannel` is used to construct a synchronous function which attempts to write the passed string to the opened serial channel `SerialChannel`. If it takes more that 5 seconds to complete the write, the function calls `p_leave(SERIAL_TIMEOUT)`. For simplicity it is assumed that there are no other outstanding events which could complete. Thus it is certain that on return from the call to `p_iowait`, one of the two asynchronous requests has completed.

```
LOCAL_C VOID StringToSerial(TEXT *str)
    {
    UWORD len;
    WORD TimerStatus;
    WORD SerialStatus;
    ULONG timeout;

    len=p_slen(str);
    p_ioc(SerialChannel,P_FWRITE,&SerialStatus,str,&len);
    timeout=50; /* 5 second timeout */
    p_ioc(TimerChannel,P_FRELATIVE,&TimerStatus,&timeout);
    p_iowait();
    if (SerialStatus==E_FILE_PENDING)
        { /* must have timed out */
        p_iow(SerialChannel,P_FCANCEL);
        p_waitstat(&SerialStatus);
        p_leave(SERIAL_TIMEOUT); /* never returns */
        }
    p_iow(TimerChannel,P_FCANCEL);
    p_waitstat(&TimerStatus);
    }
```

The functions p_ioc and p_iow are described in the next chapter: *I/O System*.

Cancelling an asynchronous request

In the above example, the asynchronous request which does not complete is cancelled by making a P_FCANCEL request using p_iow (the synchronous version of p_ioc or p_ioa). Most asynchronous request functions have an associated cancel function; the P_FCANCEL, to cancel I/O requests on a device channel, is one example. Other examples are:

p_mcancel	to cancel a call to p_mreceive to receive an inter process message
p_logoffa	to cancel a call to p_logona for being informed of a process termination

The following general principles apply to all functions that cancel an asynchronous request:

- the cancel precipitates the completion of the operation (it does not stop the operation from completing)
- the cancel may or not be effective (that is, the operation may complete naturally before the cancel is processed)
- after a cancel, you must still process the completion of the asynchronous request (typically by immediately calling p_waitstat to "use up" the signal)

Waiting for a particular completion

When waiting for the completion of a *particular* asynchronous request, the wait on the I/O semaphore must be sure that it is not fooled into a premature return by the completion of any other pending asynchronous request. This is done by calling p_waitstat which behaves in a similar way to p_iowait except that it only returns when the associated status word is other than E_FILE_PENDING.

In general, p_waitstat is a safer option than p_iowait to "use up" the signal resulting from the cancelled operation. If the cancel is not immediately effective and another completion causes p_iowait to return, the program could continue and make another request before the cancelled operation completes (which would result in p_panic being called). The above example illustrates the technique although, in this case, it is not strictly necessary since there can be no confusion as to which event has completed.

Constructing synchronous functions

General purpose functions that provide a synchronous interface must use p_waitstat rather than p_iowait since they can not assume that there are no other pending asynchronous requests.

The use of p_waitstat is illustrated by the following example of a synchronous function:

```
GLDEF_C INT ResumeWait(HANDLE pid)
    {
    WORD stat;
    INT ret;

    if (!(ret=p_logona(pid,&stat)))
        {
        p_presume(pid);
        p_waitstat(&stat);
        ret=stat;
        }
    return(ret);
    }
```

This is intended to be used in place of p_presume and behaves like p_presume except that it returns only when the resumed process has terminated. It effectively implements a synchronous version of p_logona.

Wait handlers

Wait handlers are functions that handle the completion of asynchronous requests from within p_iowait (or p_waitstat). Active wait handler functions are called just before p_iowait would have otherwise returned.

Many I/O devices install a *device wait handler* when the device is opened (see the *System Services* reference manual for information on writing device drivers and I/O device wait handlers). An application can install any number of *application wait handlers* using p_svecadd (p_svecrem removes an application wait handler). Before p_iowait will call an installed wait handler, it has to be activated using p_sveccall (p_sveccall can also be used to deactivate a wait handler). An installed wait handler is automatically deactivated when called (which stops it being called recursively since a wait handler often itself calls p_iowait, directly or indirectly).

In the following example (which does not check for errors), SetupHeapChecker installs the wait handler CheckHeap which then calls p_allchk every 2 seconds or so without any cooperation from the rest of the program.

```
typedef struct
    {
    UBYTE *Channel;
    WORD Status;
    ULONG Timeout;
    } HEAP_TIMER;

LOCAL_C INT CheckHeap(HEAP_TIMER *pTimer)
    {
    if (pTimer->Status==E_FILE_PENDING)
        return(P_SIGNAL_UNUSED);
    p_allchk(0);
    p_ioc(pTimer->Channel,P_FREAD,&pTimer->Status,&pTimer->Timeout);
    return(P_SIGNAL_ENABLE)
    }

GLDEF_C VOID SetupHeapChecker(VOID)
    {
    HEAP_TIMER *pHeapTimer;

    pHeapTimer=p_alloc(sizeof(HEAP_TIMER));
    p_open(&pHeapTimer->Channel,"TIM:",-1);
    pHeapTimer->Status=0;
    pHeapTimer->Timeout=20; /* 2 second tick */
    p_sveccall(p_svecadd(CheckHeap,pHeapTimer),TRUE);
    }
```

Wait handlers are only called when the process calls p_iowait (or a function such as p_waitstat that calls p_iowait). While an application performs a computationally intensive task that takes an extended time, it should consider calling p_ioyield (which effectively calls p_iosignal followed by p_iowait) to allow any installed wait handlers to be called. Application programs must never assume that no wait handlers have been installed.

Installed device wait handlers and application wait handlers are represented as a doubly linked queue of data structures allocated out of the process heap. The queue is built from a 4-byte queue header in a

reserved static at address 2 in the process data segment. If this reserved static is corrupted (say because of a write using an uninitialised pointer that happens to have a low value), p_iowait will almost certainly detect an invalid wait handler and call p_panic(27) although the cause of the panic may have nothing to do with wait handlers.

Polling rather than waiting

In a multi-tasking operating system it is extremely anti-social to wait for an operation to complete by polling the status word in a tight loop rather than call p_iowait (because the polling will "hog" the processor to no benefit). However, when there is useful work to be done between each poll, it can be appropriate to poll - for example, to check periodically for user input while performing an extended calculation.

If this approach is used, you should be aware that in some cases asynchronous requests are completed by a wait handler and it is necessary to call p_ioyield *before* each poll to give any wait handlers a chance to run.

This case occurs when making requests on a device driver that services hardware interrupts. For example, the serial port driver services the hardware interrupt generated by the receipt of a serial frame. A hardware interrupt handler cannot write directly to the data segment of the requesting process because that data segment may be moving when the interrupt occurs. Instead, the interrupt handler must write first to a fixed memory location (either in the operating system variables if it is an in-built driver or in a device segment if it is an external driver) and then signal the I/O semaphore of the requesting process. When the requesting process next calls p_iowait (directly or indirectly through say p_ioyield), the device driver's wait handler is called to copy the data safely to the process data segment.

Device drivers that are implemented by a server process (for example, the window server) do not require a wait handler to complete operations.

When a poll detects a completed status word it is still obligatory to "use up" the signal by calling p_iowait (otherwise you will get a "stray signal" later).

Attached I/O devices

Wait handlers are often used in attached I/O drivers that layer over an existing driver (which may be another attached driver or a hardware device driver) to extend or modify its services. The attached driver typically installs a wait handler to handle the completion of requests on the underlying driver. Although attached drivers can be written in C, the interface between the I/O system and the functions that are called requires some assembly language programming. See the *I/O System* chapter for more information on attached drivers.

Primitive semaphore functions

p_semcrt — Create a semaphore

```
HANDLE p_semcrt(INT nCount);
```

Create a semaphore with an initial positive count nCount. Returns the handle of the semaphore if successful, or E_GEN_NOSEM if no semaphores are available.

The created semaphore is owned by the calling process and, in version 3 and later of EPOC, may be deleted using p_semdel before exiting. However, the semaphore will always be automatically deleted on termination of the process.

Calls p_panic if nCount is negative.

p_semdel — Delete a semaphore

```
VOID p_semdel(HANDLE sHandle);
```

Delete semaphore sHandle. Any processes waiting on the semaphore are automatically signalled.

Calls p_panic if sHandle is not the handle of a previously created semaphore.

Prior to version 3 of EPOC the recommended action is not to delete a semaphore, but to let the operating system perform any necessary clean-up actions on termination of the application. This is an acceptable solution for *all* versions of EPOC. Except in extreme cases, where large numbers of semaphores are created, there is no need for an application ever to call p_semdel.

p_wait — Wait on a semaphore

```
VOID p_wait(HANDLE sHandle);
```

Decrement semaphore `sHandle` by one and return immediately if it is zero or positive. If `sHandle` is negative after being decremented, it waits for semaphore `sHandle` to be signalled by another process or by an interrupt handler (or for `sHandle` to be deleted).

More than one process can be waiting on a particular semaphore at a time. When there are multiple processes waiting on a semaphore, they are released on a first-in first-out basis. If a semaphore is deleted, all processes waiting on that semaphore are released.

Calls `p_panic` if `sHandle` is not the handle of a previously created semaphore.

p_signal — Signal a semaphore

```
VOID p_signal(HANDLE sHandle);
```

Signal semaphore `sHandle`, incrementing it by one. If `sHandle` was less than zero, the first process waiting on it is released and a reschedule takes place (before `p_signal` returns).

Calls `p_panic` if `sHandle` is not the handle of a previously created semaphore.

p_signaln — Signal a semaphore n times

```
VOID p_signaln(HANDLE sHandle, INT nTimes);
```

Equivalent to calling `p_signal(sHandle)` `nTimes` times.

Calls `p_panic` if `sHandle` is not the handle of a previously created semaphore or if `nTimes` is not greater than or equal to 1.

p_signalnr — Signal a semaphore with no re-schedule

```
VOID p_signalnr(HANDLE sHandle);
```

Behaves as for `p_signal` except that the re-schedule does not take place.

In the absence of any other cause, a re-schedule will not occur until the next system tick. A re-schedule can always be forced by calling `p_sleept(0L)`.

Calls `p_panic` if `sHandle` is not the handle of a previously created semaphore.

The I/O semaphore

Although the I/O semaphore is indeed associated with I/O operations, it is not used exclusively for I/O operations. In retrospect, a more accurate name would have been the "asynchronous request semaphore".

p_iosignal — Signal the IO semaphore

```
VOID p_iosignal(VOID);
```

Increment the process I/O semaphore.

Used in wait handlers to signal the completion of an asynchronous request after writing the completion status to the associated status word.

It can also be used outside wait handlers to generate "internal events" where the call to `p_iosignal` necessarily precedes the call to `p_iowait`.

p_iosignalbypid — Signal the IO semaphore of another process

```
VOID p_iosignalbypid(HANDLE pid);
```

Signal the I/O semaphore of process `pid`.

Used to signal the completion of a request from process pid (normally a different process but it still works if it is the same process). Before calling p_iosignalbypid, the process should already have set the pid's status word using say p_pcpyto.

p_iowait — Wait on the IO semaphore

```
VOID p_iowait(VOID);
```

Wait for the I/O semaphore to be signalled (of course, it returns immediately if the I/O semaphore has already been signalled).

When the I/O semaphore is signalled, any active wait handlers are called. Only when all the active wait handlers indicate that the signal has nothing to do with them (ie they all return P_SIGNAL_UNUSED) will the p_iowait call return. When it does return, the signal must be associated with an external (ie outside any wait handler) status word.

The application should then poll the request status words to determine which operation has completed.

p_ioyield — Allow any wait handlers to run

```
VOID p_ioyield(VOID);
```

Give an opportunity for any active wait handler to run. Equivalent to calling p_iosignal followed by a p_iowait.

Any application that polls a status word for the completion of an I/O operation (presumably in between performing chunks of a computationally intensive task) should call p_ioyield before polling to give any wait handlers (which are commonly required to complete an asynchronous request) a chance to run.

p_waitstat — Wait for a particular request to complete

```
VOID p_waitstat(WORD *pstat);
```

Wait for the particular asynchronous request associated with *pstat to complete.

It is similar to p_iowait except that rather than just wait for the I/O semaphore to be signalled, it also waits until *pstat is not E_FILE_PENDING. It correctly adjusts the I/O semaphore if any other I/O requests completes in the meantime, as illustrated in the following code:

```
GLDEF_C VOID p_waitstat(WORD *pstat)
/*
Wait for *pstat!=E_FILE_PENDING
*/
    {
    INT i;

    i=(-1);
    do
        {
        p_iowait();
        i++;
        } while (*pstat==E_FILE_PENDING);
    while (i--)
        p_iosignal();
    }
```

The principle may be extended to wait for the completion of more than one asynchronous event. This is illustrated in the following code, which waits until both of two status words are not equal to E_FILE_PENDING:

```
GLDEF_C VOID waitstat2(WORD *pstat1,WORD *pstat2)
/*
Wait until both *pstat1 and *pstat2 are not E_FILE_PENDING
*/
    {
    INT i;

    i=(-1);
    do
        {
        p_iowait();
        i++;
        } while ((*pstat1==E_FILE_PENDING) && (*pstat2==E_FILE_PENDING));
    if (*pstat2==E_FILE_PENDING)
        pstat1=pstat2;
    p_waitstat(pstat1);
    while (i--)
        p_iosignal();
    }
```

Wait handlers

p_svecadd — Add a wait handler function

```
HANDLE p_svecadd(INT (*vec)(VOID *), VOID *pcb);
```

Add function vec to the I/O semaphore wait handler list and return the non-zero handle of the wait handler if successful or zero if there is insufficient memory.

The returned handle is subsequently used to activate (by calling `p_sveccall`) and remove the wait handler (by calling `p_svecrem`).

Initially the wait handler is inactive. The installed wait handler is normally activated to process the completion of one or more asynchronous requests.

When the wait handler is active, the wait handler function is called from within `p_iowait` when the I/O semaphore is signalled. In the interests of efficiency, a wait handler should only be active while there is an associated pending request.

The wait handler function should poll the one or more status words associated with the one or more requests it is monitoring and return one of the following values:

`P_SIGNAL_DISABLE`	the status word was other than `E_FILE_PENDING` and the completion has been processed. There are no more requests to process and the wait handler can now be deactivated.
`P_SIGNAL_ENABLE`	a status word was other than `E_FILE_PENDING` and the completion has been processed. However, there are still pending requests (either because the wait handler is associated with more than one request or because another request was queued) and the wait handler should remain active.
`P_SIGNAL_UNUSED`	all status words contained `E_FILE_PENDING` and no processing took place. The wait handler remains active.

In the first two cases, where the signal is consumed, `p_iowait` loops back and waits on the I/O semaphore again.

If the wait handler does not detect the completion of an internal request and returns `P_SIGNAL_UNUSED`, `p_iowait` will call any other active wait handlers and will only return to the caller when there are no active wait handlers or when all the active wait handlers return `P_SIGNAL_UNUSED`.

If in the processing of the completion of one request the wait handler cancels another request, the wait handler would normally use up the signal from the cancelled requests by calling `p_waitstat`.

An active wait handler is deactivated before being called and is only reactivated when it returns with the value `P_SIGNAL_ENABLE` or `P_SIGNAL_UNUSED`. This normally works such that any calls to `p_iowait` within a wait handler will not cause the same wait handler to be called recursively. Other active wait handlers may still be called within a `p_iowait` within the wait handler.

The wait handler function vec is called with pcb as its single parameter. Where re-entrant code is required, pcb would normally be an address leading to the status word (or words) associated with the requests the wait handler is interested in. In non re-entrant code, where static data is used, it may not be necessary to use pcb.

p_sveccall — Activate/deactivate a wait handler

```
VOID p_sveccall(HANDLE hand, INT isactive);
```

If isactive is TRUE activate wait handler hand (where hand was returned by p_svecadd). If isactive is FALSE deactivate wait handler hand.

In the interests of efficiency, wait handlers should be deactivated when they have no pending requests to process. As well as being externally deactivated by calling p_sveccall, a wait handler can deactivate itself by returning P_SIGNAL_DISABLE.

p_svecrem — Remove a wait handler

```
VOID p_svecrem(HANDLE hand);
```

Remove wait handler hand (where hand was returned by p_svecadd) from the I/O semaphore wait handler list.

CHAPTER 9

I/O SYSTEM

This chapter describes the EPOC I/O system in general and the C functions used to access I/O devices. To use a particular device you need (also) to read a description of the device driver.

The files device driver and the asynchronous timer device driver are described in this manual - in the chapters *Files* and *Time, Timers and Dates* respectively. Other device drivers are described in the *I/O Devices* manual.

The last section of this chapter describes a set of console services for constructing rudimentary user interfaces with the minimum of effort. Worthier user interfaces may be implemented using the services described in the *Window Server* reference manual.

I/O Device Drivers

The principal purpose of device drivers is to provide convenient software interfaces that hide the internal details of the underlying hardware. For example, the software interface to the RS232 driver is independent of the interface to the underlying hardware. The fact that the RS232 hardware is different between SIBO machines and PCs is not apparent to the user of the RS232 driver.

Many device drivers do not themselves interface to hardware but layer over one or more other device drivers that ultimately access the hardware. For example, the majority of the code in the RS232 driver is independent of the hardware interface and the driver is implemented in 2 layers - an upper hardware-independent layer and a lower hardware-dependent layer.

Some device drivers do not access hardware at all - even indirectly. In such cases, the device driver mechanism is used to extend the services available to applications. For example, the C standard floating point library is implemented as a device driver.

LDDs and PDDs

In EPOC there are two types of device driver:

- physical device drivers (PDDs) which are hardware dependent
- logical device drivers (LDDs) which are hardware independent

Applications normally interface to LDDs only. An LDD may use one or more PDDs in its implementation. For example, the RS232 driver is an LDD (the upper layer) using an appropriate PDD (the lower layer) depending on the underlying hardware. PDDs are also used by the file server system process to access different types of SSDs.

Some device drivers are built into the operating system where their code is in the ROM. For example, the RS232 LDD and its PDD are normally built into the operating system whereas a bar code reader device driver is normally external and has to be loaded.

External device drivers

External device drivers are loaded from a device driver file into a device memory segment. The device driver file has the file name extension .LDD or .PDD depending on whether it is an LDD or PDD respectively. The device memory segment is allocated by the segment allocator as described in the chapter *Memory Allocation*.

The ability to load and remove external device drivers without a system reset is a key feature of the EPOC I/O system and contrasts with most other operating systems, which require a system reset to install

a device driver. On a SIBO machine you can physically attach a peripheral device (for example, a bar code wand) and load a device driver without having to exit any application processes. The I/O system also notifies devices when the machine switches off and on so that the driver can take appropriate device specific action before losing power and to recover when the power is resumed.

The interface between the operating system and an LDD does not conform to a C calling convention. An LDD may be written in C but some 8086 assembly language is required to provide the LDD interface. See the *System Services* reference manual for more about device drivers.

Opening a channel to a device

The I/O device driver functions are accessed by opening a channel to the device by calling `p_open` and passing it a name consisting of a 3 character device name and a terminating colon. Examples of device names are:

`FIL:`	for opening a file
`PAR:`	for opening a parallel port
`TTY:`	for opening an RS232 port
`TIM:`	for opening an asynchronous timer

Note that device names of external LDDs bear no relation to the file name from which they were loaded.

Depending on the nature of the device, the ':' may be followed by further text. Where the device driver supports more than one unit, the device name may be followed by a unit letter. For example, `"TTY:A"` is used to open a channel to serial port A and `"PAR:B"` is used to open a channel to parallel port B.

In the file system device `FIL:`, the qualifying text is normally a file name or full path name. Because the file system has more `p_open` calls made to it than any other device, the `p_open` function effectively inserts a `"FIL:"` before a device name if it fails to recognise a valid device name - making the leading `"FIL:"` optional. For example, calling `p_open` with the name `"C:\NOTES\NEW.TXT"` has the same effect as the name `"FIL:C:\NOTES\NEW.TXT"`. Keeping the `FIL:` prefix removes any possibility of mistaking the file specification for a device name - for example to open a file on the default directory with file name `"TTY:"`.

The `p_open` function works such that a loaded device driver supersedes any existing device of the same device name.

Operations on an open I/O channel

Once a channel has been opened on a device, the primitive `p_ioa` function is used (directly or indirectly) to make an asynchronous I/O function request on the channel. Asynchronous requests are described in the chapter *Asynchronous Requests and Semaphores*.

All I/O function requests are asynchronous in principle and the process I/O semaphore is always signalled as a result of an I/O function request. In practice, many I/O functions are implemented synchronously, which means that the I/O operation will have completed before `p_ioa` returns. A typical I/O device will provide zero, one, two or three truly asynchronous functions (where the request will probably not have completed before `p_ioa` returns) with the remaining functions provided synchronously. For example, the I/O function that closes an I/O channel is always implemented synchronously, whereas the I/O function that reads input from a device is commonly implemented asynchronously.

In practice, `p_ioa` is often called indirectly by:

`p_ioc`	which makes an asynchronous I/O request in a way that simplifies the handling of error returns - this should generally be used in preference to `p_ioa`
`p_iow`	which makes the I/O request and then waits for the request to complete (by calling `p_waitstat`)

The particular I/O function requested by a call to `p_ioa`, `p_ioc` or `p_iow` is specified by a function number parameter of the form `P_Fxxx`, defined in *p_file.h*. Some I/O functions are specific to a particular device (for example `P_FSETEOF`, which sets the end of file position on an open file channel). Some I/O functions apply to more than one device, including:

`P_FREAD`	to read data from a channel
`P_FWRITE`	to write data to a channel
`P_FCLOSE`	to close a channel
`P_FCANCEL`	to cancel outstanding asynchronous requests on a channel

P_FSENSE	to sense channel characteristics
P_FSET	to set channel characteristics
P_FFLUSH	to flush out data held in buffers

This manual uses the notation p_ioa(P_Fxxx), p_ioc(P_Fxxx) and especially p_iow(P_Fxxx) to refer to the corresponding function call with P_Fxxx passed as the I/O function number. (In the descriptions of p_ioa, p_ioc and p_iow later in this chapter the I/O function number has the parameter name func.)

The most commonly used I/O functions are supported by their own synchronous convenience functions as follows:

p_close	which calls p_iow(P_FCLOSE)
p_read	which calls p_iow(P_FREAD)
p_write	which calls p_iow(P_FWRITE)
p_seek	which calls p_iow(P_FSEEK)

The functions p_close, p_read and p_write are used across many devices (p_close is applicable to all devices) and are described in this chapter. The function p_seek applies to an open file channel only and is described in the *Files* chapter.

The file server

The file server is a high priority system process (with process name SYS$FSRV) that performs all file related operations. These include those that are accessed using the I/O channel services (the FIL: device) and those that do not require a channel to be opened (eg deleting a file, making a directory). The file server is also responsible for loading executables (see the chapter *Processes and Inter-Process Messages*) and for loading external device drivers (described in this chapter).

The file server serialises access to shared file storage devices (eg SSD drives on a SIBO machine) which may be local or remote. On SIBO machines, the file server uses PDDs extensively to access the many different types of SSD (different configurations of FLASH, RAM, ROM). The file server also uses an installable PDD to connect to remote devices via a remote file server. The file server is a process rather than an LDD because, in a multi-tasking system, more than one process can be accessing a particular file storage device at a time (at a lower level only the file server owns an SSD drive where it performs all operations on the SSD on behalf of requesting processes).

Application processes that use the file server are called "clients". To become a client of the file server, a process must connect to it. Because most applications need the file server, the normal code that precedes main connects to the file server.

The client processes send the file server an inter-process message to request a file server service. If the service accesses local devices, it is implemented synchronously since the file server runs at a higher priority than any application process. If the service accesses remote devices, it may be implemented asynchronously.

The application programmer does not program at the message passing level but uses the interfaces provided by the FIL: device and a number of ROM resident functions (most of which are described in the *Files* chapter). Whether provided by a function or via the FIL: device, all access to the file server ultimately involves the sending of an appropriate inter-process message.

Attached drivers

An attached driver is an LDD using the services of an another LDD to provide a different set of services to its user. Some of the devices described in the *I/O Devices* manual are attached drivers.

For example, the PRD: device is an attached driver, layered over a suitable print output device to provide primitive printer driver services (involving translates, preambles and postambles as driven from a .PRD file). In this case a suitable print output device is any device that supports the normal P_FWRITE operation as provided by the PAR:, TTY: and FIL: devices.

The user of the PRD: device does the following:

- open the print output device using p_open (eg PAR: or TTY:)
- perform any initialisation on the channel (eg to set the Baud rate on a TTY: channel)
- open the PRD: device using p_open, attaching it to the opened print output device

Once the PRD: device has been opened, it replaces the P_FWRITE, P_FCANCEL and P_FCLOSE functions of the underlying device.

In general, an attached driver may completely replace the original driver's functions or it may augment them (possibly also replacing or disabling some functions). It may also allow some functions through to the underlying driver or beyond.

Like any other LDD, an attached driver may be written in C but some assembly language code is required to provide the LDD vector interface. See the *System Services* reference manual for more about writing attached device drivers.

Attached drivers that provide one or more of their services asynchronously make internal asynchronous requests on the device they are attached to. Where this is the case, the attached driver processes the completion of its internal requests in code that is entered via its wait handler vector. The attached device installs its device wait handler when the channel is opened and the wait handler vector is subsequently called from within the `p_iowait` of the process which opened the device whenever the process I/O semaphore is signalled. Except for the way in which they are called, device wait handlers are the same as application wait handlers as described in the chapter *Asynchronous Requests and Semaphores*.

Channel-based I/O functions

p_open (or f_open) — Open a channel to a device

```
INT p_open(VOID **ppfcb, TEXT *name, UINT mode);
INT f_open(VOID **ppfcb, TEXT *name, UINT mode);
```

Open a channel to the device with the zero terminated device name `name` or attach driver `name` to an existing channel.

The parameter `name` consists of a 3 character device name terminated by a ':' and optionally followed by further data, depending on the device.

The interpretation of the `mode` parameter depends upon the device and some devices ignore `mode`. When a device ignores `mode`, the caller should pass a `mode` of -1.

The files device (device name `"FIL:"`) and the asynchronous timer device (device name `"TIM:"`) are described in the chapters *Files* and *Time, Timers and Dates* in this manual. These devices are also described in the *I/O Devices* manual along with many other devices.

If the device `name` is not an attached device and it is opened successfully, the address of the channel control block is written to `*ppfcb`. If the open fails, `*ppfcb` is not written to - setting `*ppfcb` to zero before calling `p_open` can simplify clean-up code since `p_close(0)` has no effect and `p_close` can subsequently be called regardless of whether the open call was successful.

If device `name` is an attached device, the driver control block is attached to `*ppfcb` - the value in `*ppfcb` is not changed.

Returns zero if successful or a negative error number if it failed. As well as device specific errors (as described in the description of each device) the following error numbers may be returned:

`E_FILE_ALLOC`	failed to allocate memory for the control block
`E_FILE_DEVICE`	the device does not exist
`E_GEN_ARG`	the value of the `mode` parameter was invalid (possibly as a result of falling through to the `FIL:` device, as described next)

The function `f_open` is identical to `p_open` except that it calls `p_leave` (passing the error number) rather than return a negative error number.

If `p_open` fails to locate a device that matches `name`, then `name` is passed to the `FIL:` device driver (in which case the process must be connected to the file server). To put it another way, the leading `FIL:` device name is optional when opening files and, for example, calling `p_open` with a `name` of `"LOC::A:\DEF.EXT"` is equivalent to a `name` of `"FIL:LOC::A:\DEF.EXT"` (the `FIL:` device is described in more detail in the *Files* chapter).

This behaviour has an undesirable side effect: an attempt to open a device that does not exist does not give the expected `E_FILE_DEVICE` error since `name` is passed on to the `FIL:` device with a result that depends upon `mode` (with some values of `mode` the open might even be successful). Passing a `mode` of -1 is guaranteed to cause the `FIL:` device driver open to fail - albeit with a misleading error number (`E_GEN_ARG`).

p_ioa — Start an I/O operation

```
INT p_ioa(VOID *pcb, INT func, WORD *pstat, ...);
INT p_ioa3(VOID *pcb, INT func, WORD *pstat);
INT p_ioa4(VOID *pcb, INT func, WORD *pstat, VOID *a1);
INT p_ioa5(VOID *pcb, INT func, WORD *pstat, VOID *a1, VOID *a2);
```

Start the I/O operation func with zero, one or two parameters on the opened channel pcb and return without waiting for the operation to complete.

You can either use p_ioa, which presents the CDECL calling convention, or one of the p_ioa? variants, which uses a more efficient register calling convention.

Note that it is almost always preferable to use p_ioc in preference to p_ioa.

The legitimate values of func depends upon the device. The number of function parameters (zero to two) and the interpretation of the function parameters (if any) depends on the function and the device. If a function parameter exists, it is normally an address that may be used as input, output or both input and output.

The function returns zero if the I/O request was started successfully or a negative error number. Returns E_FILE_INV if func is not valid for this device. Other device specific errors may be returned.

When the successfully started operation completes, the process I/O semaphore is signalled and *pstat contains the completion status. While the operation is pending (ie before the I/O semaphore has been signalled), *pstat contains E_FILE_PENDING. Asynchronous requests in general are described in the chapter *Asynchronous Requests and Semaphores*.

After the operation has completed, *pstat contains zero if the operation completed successfully or a negative error number if the operation completed with an error. The possible error numbers depend upon the device but any asynchronous operation that is successfully cancelled completes with *pstat containing E_FILE_CANCEL.

Drivers normally only support one pending request per I/O operation per channel. For example, on the TTY: (serial port) device you must wait for an asynchronous write request to complete before you can make another write request on the same channel (the driver will call p_panic if this is attempted). However, it is legitimate to have one read request and one write request simultaneously pending.

The storage pointed to by pstat must be retained for the duration of the operation. A common error, which has disastrous results, is to allocate the status word on the stack and then to return from the function before the operation has completed - as in the following example:

```
GLDEF_C INT DisastrousWrite(VOID *pcb,TEXT *buf)
    {
    WORD stat;
    UWORD len;

    len=p_slen(buf);
    return(p_ioa(pcb,P_FWRITE,&stat,buf,&len));
    }
```

The following generic I/O functions (where a1 and a2 are further parameters to p_ioa) are commonly (but not always) implemented asynchronously by device drivers:

P_FREAD — request a read operation where a1 is the address of the buffer to take the data and a2 is the address of a WORD length to read (or the maximum length to read if the device is record oriented). When the read completes, *pstat contains zero if the read was successful or a negative error number if the read failed and *a2 contains the number of bytes written to a1.

P_FWRITE — request a write operation where a1 is the address of the data to write and a2 is the address of a WORD length to write. When the write completes, *pstat contains zero if the write was successful or a negative error number if the write failed.

p_ioc Start an I/O operation with guaranteed completion

```
VOID p_ioc(VOID *pcb, INT func, WORD *pstat, ...);
VOID p_ioc3(VOID *pcb, INT func, WORD *pstat);
VOID p_ioc4(VOID *pcb, INT func, WORD *pstat, VOID *a1);
VOID p_ioc5(VOID *pcb, INT func, WORD *pstat, VOID *a1, VOID *a2);
```

Behaves as for p_ioa except that if the I/O request func fails to start, the failure is reported as if it had started successfully but completed with that error.

You can either use p_ioc, which presents the CDECL calling convention, or one of the p_ioc? variants, which uses a more efficient register calling convention.

The implementation of p_ioc is effectively as follows:

```
GLDEF_C VOID p_ioc(VOID *pcb,INT func,WORD *pstat,VOID *a1,VOID *a2)
    {
    INT ret;

    if (ret=p_ioa(pcb,func,pstat,a1,a2))
        {
        *pstat=ret;
        p_iosignal();
        }
    }
```

In most cases, p_ioc is preferred to p_ioa because there is only one place (*pstat) to check for an error rather than two (the return from p_ioa and *pstat). It is rarely necessary to differentiate between a failure to start the I/O operation and a failure in its completion.

In the following example, WriteTimeout writes the len bytes at buf to channel pcb and returns as for p_write. However, if the write does not complete within secs seconds, the write operation is cancelled and E_FILE_CANCEL is returned.

For simplicity, it assumes that completion of the write or expiry of the timer are the only two events that are expected to cause a return from the call to p_iowait. This will be true if there is no other asynchronous activity.

```
LOCAL_C INT WriteTimeout(VOID *pcb, UBYTE *buf, UWORD len, UINT secs)
    {
    WORD tstat;
    WORD wstat;
    ULONG tval;

    p_ioc(pcb,P_FWRITE,&wstat,buf,&len);
    tval=10L*secs;
    p_ioc(tcb,P_FRELATIVE,&tstat,&tval);
    p_iowait();
    if (tstat!=E_FILE_PENDING)
        { /* the timer expired */
        p_iow(pcb,P_FCANCEL); /* cancel write */
        p_waitstat(&wstat);
        }
    else if (wstat!=E_FILE_PENDING)
        { /* the write completed */
        p_iow(tcb,P_FCANCEL); /* cancel timer */
        p_waitstat(&tstat);
        }
    else
        p_panic(254); /* unexpected, unrecognised signal */
    return(wstat);
    }
```

The call to p_iowait returns when either request completes. If the timer has completed, the write request is cancelled. Otherwise, the timer request is cancelled. If both requests have completed by the time p_iowait returns (quite possible if a higher priority process "hogged" the CPU), the cancel of the write request will have no effect and wstat will contain the completion code of the write.

In the example, the static variable tcb is the channel of a previously opened asynchronous timer, as in:

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

The following version of WriteTimeout is more general, and caters for the presence of other asynchronous activity.

```
LOCAL_C INT WriteTimeout(VOID *pcb, UBYTE *buf, UWORD len, UINT secs)
    {
    WORD tstat;
    WORD wstat;
    WORD count;
    ULONG tval;

    p_ioc(pcb,P_FWRITE,&wstat,buf,&len);
    tval=10L*secs;
    p_ioc(tcb,P_FRELATIVE,&tstat,&tval);
    count=0;
    FOREVER
        {
        p_iowait();
        if (tstat!=E_FILE_PENDING)
            { /* the timer expired */
            p_iow(pcb,P_FCANCEL); /* cancel write */
            p_waitstat(&wstat);
            break;
            }
        else if (wstat!=E_FILE_PENDING)
            { /* the write completed */
            p_iow(tcb,P_FCANCEL); /* cancel timer */
            p_waitstat(&tstat);
            break;
            }
        else
            count+=1; /* count unrecognised signals */
        }
    while (count--)
        p_iosignal(); /* replace unrecognised signals */
    return(wstat);
    }
```

In such a situation, with other asynchronous activity, it may be better to handle the expiry of the timer in the main p_iowait loop. The wait for the write to complete could then be handled by:

```
p_waitstat(&wstat);
p_iow(tcb,P_FCANCEL); /* cancel timer */
p_waitstat(&tstat);
```

p_iow — Start an I/O operation and wait for completion

```
INT p_iow(VOID *pcb, INT func, ...);
INT p_iow2(VOID *pcb, INT func);
INT p_iow3(VOID *pcb, INT func, VOID *a1);
INT p_iow4(VOID *pcb, INT func, VOID *a1, VOID *a2);
```

Behaves as for p_ioa except that it waits for operation func to complete and returns the completion status, providing a synchronous (as opposed to an asynchronous) interface.

You can either use p_iow, which presents the CDECL calling convention, or one of the p_iow? variants, which uses a more efficient register calling convention.

The code for p_iow is effectively:

```
GLDEF_C INT p_iow(VOID *pcb,INT func,VOID *a1,VOID *a2)
    {
    WORD stat;
    INT ret;

    if (!(ret=p_ioa(pcb,func,&stat,a1,a2)))
        {
        p_waitstat(&stat);
        ret=stat;
        }
    return(ret);
    }
```

Calling p_iow is simpler and requires one less parameter (since the status word is returned) than p_ioa or p_ioc and should be used in preference to p_ioa or p_ioc unless there is a need for an asynchronous request.

The following generic I/O functions are essentially synchronous and are normally requested using p_iow:

P_FCANCEL	to cancel outstanding requests on a channel
P_FSENSE	to sense channel characteristics
P_FSET	to set channel characteristics
P_FFLUSH	to flush out data held in buffers

p_close — Close an I/O channel

```
INT p_close(VOID *pcb);
```

Close I/O channel pcb and return zero if successful. If pcb is NULL, just return zero.

The code for p_close is effectively:

```
GLDEF_C INT p_close(VOID *pcb)
    {
    if (!pcb)
        return(0);
    return(p_iow(pcb,P_FCLOSE));
    }
```

Although p_close can return an error, it will always succeed in closing the channel (and pcb should not be used subsequently).

If the device buffers written data, the close operation may need to perform one or more write operations on closing, in which case P_FCLOSE can return similar errors to P_FWRITE. However, the failure to flush the data will not (assuming a competently written device driver) cause the close operation to be aborted although the failure to flush will be reported.

Carefully written applications avoid this problem by using P_FFLUSH to flush the data (and taking appropriate action if this fails) before closing the channel without risk of failure.

p_read (or f_read) — Read from an I/O channel

```
INT p_read(VOID *pcb, VOID *buf, UINT len);
INT f_read(VOID *pcb, VOID *buf, UINT len);
```

Request a P_FREAD of up to len bytes of data into buf from channel pcb, wait for the request to complete and, if successful, return the number of bytes written to buf or a negative error number if the read failed.

The code for p_read is effectively:

```
GLDEF_C INT p_read(VOID *pcb, UBYTE *buf, UINT len)
    {
    INT ret;
    UWORD l;

    l=len;
    ret=p_iow(pcb,P_FREAD,buf,&l);
    if (!ret)
        ret=l;
    return(ret);
    }
```

The function f_read is identical to p_read except that, if there is an error, it calls p_leave(err) rather than return the negative error number err.

p_write (or f_write) — Write to an I/O channel

```
INT p_write(VOID *pcb, VOID *buf, UINT len);
INT f_write(VOID *pcb, VOID *buf, UINT len);
```

Request a P_FWRITE of len bytes of data from buf to channel pcb, wait for the request to complete and, if successful, return zero or a negative error number if the write failed.

The code for p_write is effectively:

```
GLDEF_C INT p_write(VOID *pcb, UBYTE *buf, UINT len)
    {
    UWORD l;

    l=len;
    return(p_iow(pcb,P_FWRITE,buf,&l));
    }
```

The function f_write is identical to p_write except that, if there is an error, it calls p_leave(err) rather than return the negative error number err.

p_iow(P_FCANCEL) — Cancel requests on an I/O channel

```
INT p_iow(VOID *pcb, P_FCANCEL);
```

Cancel any outstanding asynchronous requests on channel pcb and return zero. Harmless if there are no pending requests.

Device drivers that support truly asynchronous services provide a cancel service. The detailed effect of the cancel depends upon the device driver. However, the following general principles apply:

- the cancel precipitates the completion of the request (it does not stop the request from completing)
- the cancel may or not be effective (that is, the request may complete naturally before the cancel is processed)
- after a cancel, you must still process the completion of the asynchronous request (typically by immediately calling p_waitstat to "use up" the signal)

The above principles actually apply to cancelling any asynchronous request (not just an asynchronous I/O request).

Although legitimate, using p_ioa(P_FCANCEL) or p_ioc(P_FCANCEL) (rather than p_iow(P_FCANCEL)) is somewhat perverse since you then have two signals to use up - one for the request being cancelled and one for the cancel itself.

Device driver functions

p_loadldd — Load a logical device driver

```
INT p_loadldd(TEXT *pName);
```

Load the logical device driver (LDD) from the zero terminated file name pName. If pName contains no extension an extension of .LDD is assumed (an LDD file would normally have the extension .LDD). If pName is not a full path name the current path is assumed.

Returns zero if successful or one of the following negative error numbers:

E_FILE_EXIST	an LDD of the same file name has already been loaded
E_FILE_NXIST	the LDD file does not exist
E_GEN_IMAGE	the LDD file does not have the correct format or has been corrupted
E_GEN_NOMEMORY	Not enough memory to satisfy the request
E_GEN_NOSEGMENTS	No memory segment handles are available

After the LDD has been loaded, a channel may be opened to it by calling p_open.

Applications that rely on an external LDD should use p_loadldd and not care if it fails with E_FILE_EXIST.

p_loadpdd — Load a physical device driver

```
INT p_loadpdd(TEXT *pName);
```

Load the physical device driver (PDD) from the zero terminated file name pName. If pName contains no extension an extension of .PDD is assumed (an PDD file would normally have the extension .PDD). If pName is not a full path name the current path is assumed.

Returns zero if successful or any of the same negative error numbers as for p_loadldd, above.

p_devdel — Delete a device driver

```
INT p_devdel(TEXT *pName, INT devType);
```

Delete the logical (devType is E_LDD) or physical (devType is E_PDD) device driver with the zero terminated device name pName (as used in p_open but without a trailing ':').

Only device drivers that have been loaded into RAM (and not those devices that are built into the ROM) can be deleted by this service.

Returns zero if successful or one of the following negative error numbers:

E_FILE_DEVICE	The device driver is not currently loaded.
E_GEN_NSUP	The device driver is a ROM device driver and cannot be deleted.
E_GEN_INUSE	The device driver is currently open and cannot be deleted.
A device dependent error number	Returned by the device driver code.

If you have loaded an external device driver and finished with it, it is good practice to attempt to delete it by calling p_devdel and to ignore the return value. The call is harmless if the device is loaded by another process or if the device is built into the ROM.

WARNING: If pName points to a null string, the first located unloadable device driver of the type specified by devType will be deleted.

p_devqu — Query the number of units supported by a device

```
INT p_devqu(TEXT *pName);
```

Returns the number of units supported by logical device driver with the zero terminated device name pName (as used in p_open but without a trailing ':'). The function is not applicable to PDDs.

Returns a positive number if successful or one of the following negative error numbers:

E_GEN_FAIL	Unlimited units (as returned by the FIL: device driver as it can support a large number of files).
E_FILE_DEVICE	Device driver not found.

For example:

```
p_devqu("TTY");
```

returns 2 if there are two serial expansion boards fitted.

p_devfnd — Find all devices

```
HANDLE p_devfnd(HANDLE fHandle, TEXT *pMatch, INT devType, TEXT *pName);
```

Write the next device name (as used in p_open but without a trailing ':') of type devType (either E_PDD to find PDD devices or E_LDD to find LDD devices) that matches the zero terminated match string pMatch as a zero terminated string to pName where fHandle is zero for the first call and is subsequently the positive return value from the previous call. When there are no further devices of type devType that match pMatch, p_devfnd returns E_FILE_DEVICE.

Used repeatedly to find all the PDD or LDD devices that match the wild card string pointed to by pMatch. The buffer at pName should be big enough to receive E_MAX_NAME+2 bytes. The wild card string pMatch should remain the same between successive calls.

No memory is used by this service and it can be abandoned at any time without taking any further action.

Calls p_panic if fHandle is invalid.

Example

```
LOCAL_C VOID PrintDevices(VOID)
    {
    HANDLE fh;
    TEXT bb[E_MAX_NAME+2];

    fh=0;
    FOREVER
        {
        fh=p_devfnd(fh,"*",E_LDD,&bb[0]);
        if (fh<0)
            break;
        p_puts(&bb[0]);
        }
    }
```

Simple console I/O

The simple console functions are provided for "quick and dirty" applications - for example test programs and software tools. They are not suitable for constructing quality user interfaces.

The console functions use the services of the CON: device driver to implement the following screen output and keyboard input functions:

p_putch	to write a character to the screen
p_puts	to write a line of text to the screen and move to the beginning of the next line
p_printf	to convert numbers to printable form and write them as a line of text to the screen and then move to the beginning of the next line
p_print	to convert numbers to printable form and write them to the screen
p_getch	to get (without echo) a single character from the keyboard
p_gets	to input (with simple backspace editing) a line of text from the keyboard

p_getl — to display a prompt and then input (with simple backspace editing) text from the keyboard

Redirecting console writes

These functions automatically open CON: when they are first used. The open channel is stored in the global static:

```
GLREF_D VOID *winHandle;
```

which is initialised to NULL. You can open a suitable alternative device (eg a file or TTY:) before the first usage of a console function to redirect the output functions p_putch, p_print and p_printf. For example, to redirect to the file *a.lis* in the current path, use:

```
p_open(&winHandle,"a.lis",P_FREPLACE|P_FUPDATE);
```

Note that if you do redirect the console to such a device, you should not use p_getch, p_gets or p_getl.

Changing the size of the console window

If you want to change the size of the console window, you can do this by declaring the global P_RECT structure _DefScreenRect and initialising it to the required size before the first usage of a console function. The P_RECT struct is defined in *p_graf.h* as:

```
typedef struct
    {
    WORD x; /* Horizontal coordinate */
    WORD y; /* Vertical coordinate */
    } P_POINT;

typedef struct
    {
    P_POINT tl; /* Top left point */
    P_POINT br; /* Bottom right point */
    } P_RECT;
```

where the top left coordinates should be (0,0) and the bottom right coordinates should reflect the required dimensions in character columns and rows as in, for example:

```
GLDEF_D P_RECT _DefScreenRect={0,0,10,20};
```

for 10 columns by 20 rows, or:

```
GLDEF_D P_RECT _DefScreenRect;

    _DefScreenRect.tl.x=0;
    _DefScreenRect.tl.y=0;
    _DefScreenRect.br.x=columns;
    _DefScreenRect.br.y=rows;
```

Changing the console window mode

By default the console starts up in the native mode of the machine. At the time of writing, all SIBO machines default to single pixel (non-compatibility) mode, with no access to grey. If you want to change the mode, you can do this by declaring the global variable _DefScreenMode and initialising it to the required value before the first usage of a console function. The possible modes are:

0	use the native mode of the machine - this is the default value
1	compatibility mode, allowing Series 3 software to run on the Series 3a
2	non-compatibility mode, with grey enabled
3	compatibility mode, but with grey enabled

Thus, compatibility mode may be set on the Series 3a as follows:

```
GLDEF_D INT _DefScreenMode;

    ....
    _DefScreenMode=1;
    ...
```

Values that are not relevant to a particular type of machine are simply ignored.

p_putch — Write a character to the console

```
VOID p_putch(UINT c);
```

Write character `c` to the console, opening the console if necessary.

p_puts — Write a string to the console

```
VOID p_puts(TEXT *str);
```

Write the zero terminated string `str` to the console and start a new line. The console is opened if necessary.

p_printf — Convert arguments and write line to console

```
VOID p_printf(TEXT *fstr, ...);
```

Converts multiple arguments to an internal buffer under control of the format string `fstr`, writes the complete string to the console screen and then moves the print position to the beginning of the next line. The console is opened if necessary.

The internal buffer is `P_MAXSYSIO` (258) bytes long and the length of the output (which depends on the arguments) must be limited to `P_MAXSYSIO-2` (256) bytes per call of `p_printf`.

The format of `fstr` is exactly the same as for `p_atob`, which is described in the chapter *Integer Conversion and Rectangle Functions*.

p_print — Convert arguments and write to console

```
VOID p_print(TEXT *fstr, ...);
```

Behaves as for `p_printf` above except that the print position is not automatically moved to the beginning of the next line after the write.

You can embed `\r` and `\n` characters in `fstr` to move to the beginning of the line and to move down a line respectively.

p_getch — Get a character from the console

```
INT p_getch(VOID);
```

Wait for a key to be pressed and return its character code. The console is opened if necessary.

p_gets — Get a string from the console

```
INT p_gets(TEXT *str);
```

Input (with simple backspace editing) a line of up to `P_MAXSYSIO-1` characters from the keyboard, creating a zero terminated string at `str`. The input is terminated by the user pressing the Enter key.

Returns the length of the string at `str`.

There should be at least `P_MAXSYSIO` bytes at `str`. The console is opened if necessary.

p_getl — Get a string with prompt from the console

```
INT p_getl(TEXT *pmt, TEXT *str, INT len);
```

Write the zero terminated string `pmt` to the console and input (with simple backspace editing) up to `len` characters from the keyboard, creating a zero terminated string at `str`. The input is terminated by the user pressing the Enter key.

Returns the length of the string at `str`.

There should be at least `len+1` bytes at `str`. The console is opened if necessary.

CHAPTER 10

TIME, TIMERS AND DATES

System time

The system time is set and sensed as a `ULONG`, counting the number of seconds since 00:00:00, January 1, 1970 (compatible with UNIX system time).

The system time overflows approximately 136 years after 1970 (which is sometime in the year 2106).

p_date — Return the system time

```
ULONG p_date(VOID)
```

Return the system time as the number of seconds since 00:00:00, January 1, 1970.

p_sdate — Set the system time

```
VOID p_sdate(ULONG newTime);
```

Set the system time to `newTime` (the number of seconds since 00:00:00, January 1, 1970).

Note that setting the system time forward by an interval will cause any absolute timers due in that interval to expire.

Absolute and relative timers

In the EPOC operating system a process can be waiting on a timer in two ways:

- The process is in the time delta queue as a result of calling `p_sleep`, `p_sleept` or `p_sleepa`.
- The process is waiting for its I/O semaphore to be signalled by a timer device entry in the time delta queue after making an asynchronous timer request by calling `p_ioc(P_FRELATIVE)` or `p_ioc(P_FABSOLUTE)` on an open timer channel.

Each entry in the time delta queue contains the signed long delta time in system ticks relative to its predecessor while the head of the queue contains the delta time relative the current system time (for more on delta queues see the chapter *Characters, Strings, Buffers and Queues*). The head of the time delta queue is decremented every system tick and the corresponding timer expires (and is removed from the queue) when its delta time is zero or negative. If the entry is a process, the process will either run (if it has the highest priority) or it will wait in the ready queue. If the entry is a timer device entry, the associated process I/O semaphore is signalled. See the chapter *Asynchronous Requests and Semaphores* for information on the I/O semaphore, asynchronous requests and the ready queue.

As well as being marked as processes or timer device entries, entries in the time delta queue are also marked as being *absolute* or *relative*.

Absolute timers

An absolute timer is characterised by the following:

- the timer expiry is set in terms of an absolute time (the number of seconds since 00:00:00, January 1, 1970)
- SIBO machines (as opposed to a PC running EPOC) that have switched off will automatically switch on when an absolute timer is due to expire
- setting the system time forward by an interval will cause any absolute timers due in that interval to expire

Absolute timers are used to implement, for example, alarms for the Diary and Alarms applications on the MC GI machines. You can set absolute timers to expire after a relative time interval (expressed in seconds) by adding the required interval to the time returned by `p_date`.

Although absolute timers are converted to relative signed long delta time in the time delta queue, absolute timer entries can re-launch themselves to cover ranges in excess of `0x7fffffff` ticks.

Relative timers

A relative timer is characterised by the following:

- the timer expiry is set as a time interval relative to the current system time (either as a number of 1/10ths of a second or as a number of system ticks)
- the timer stops running while SIBO machines are switched off and it follows that relative timers do not wake up the operating system up
- changing the system time has no effect on relative timers

If a relative timer is set to expire after 5 seconds when the system time is advanced by 1 hour the timer still expires after 5 seconds (provided the machine is not switched off).

If a relative timer is due to expire in 5 seconds when the machine switches off for 1 hour, the relative timer actually expires after 1 hour plus 5 seconds.

SIBO machines are normally configured to switch off automatically after a period of inactivity (typically 5 minutes). A process that repeatedly waits on a relative timer with an interval shorter than the switch off inactivity period (for example to implement a clock or a flashing cursor) would by default stop the machine from ever switching off. Keeping a battery powered machine on indefinitely is normally undesirable and, in this situation, the application should call `p_unmarka` (described in the chapter *Processes and Inter-Process Messaging*) to stop such activity from keeping the machine on.

Relative timers are ultimately converted to system ticks in a signed long - giving them a range of `0x7fffffff` ticks, or approximately 2.1 years, on a SIBO machine (which ticks 32 times a second).

p_sleep — Suspend process for n tenths of a second

```
INT p_sleep(ULONG n);
```

Return zero after `n` 1/10ths of a second has elapsed.

Returns `E_GEN_OVER` immediately if the conversion from 1/10ths to ticks overflows (ie the number of ticks is greater than `0x7fffffff`).

This function uses a relative timer. If the machine switches off before the function returns, it will take indefinitely longer than `n` tenths of a second to return.

p_sleept — Suspend process for n system ticks

```
INT p_sleept(LONG nTicks);
```

Returns zero after `nTicks` system ticks if successful or `E_GEN_ARG` if `nTicks` is negative.

On the SIBO hardware the system ticks 32 times a second. On IBM PCs and compatibles the system ticks 18.2 times a second. The constant `E_TICKS_PER_SECOND` in *epoc.h* contains the number of ticks per second, to the nearest integer.

A request to sleep for zero ticks is valid and will just force a re-schedule, with the process calling this service losing the remainder of its time slice if there are other processes at the same priority.

This function uses a relative timer. If the machine switches off while the process is suspended, it will take indefinitely longer than `nTicks` system ticks to return. In the absence of any switching off, the actual time the process is suspended is greater than or equal to `nTicks` and less than `nTicks+1`.

p_sleepa — Suspend process until absolute time

```
INT p_sleepa(ULONG time);
```

Return when the system time is `time` (the number of seconds since 00:00:00, January 1, 1970). Returns zero if successful or `E_GEN_ARG` if `time` is earlier than the current system time.

This uses an absolute timer which will wake the machine up if necessary when the timer expires. Another process setting the system time past the expiry time (using `p_sdate`) will cause `p_sleepa` to return.

Asynchronous timers

The functions `p_sleepa`, `p_sleep` and `p_sleept` are *synchronous* functions in the sense that they return only when the requested operation (in this case the expiry of a timer) has completed. With *asynchronous* timers, the function call to start the timer returns immediately. This allows other processing to take place before waiting for any one of a number of events (including the expiry of the timer) by calling `p_iowait`. See the chapter *Asynchronous Requests and Semaphores* for an explanation of asynchronous requests in general.

In EPOC, asynchronous timers are implemented as an I/O device with the device name "TIM:". To use an asynchronous timer, you open a channel to "TIM:" and use:

`p_ioc(P_FRELATIVE)`	to start a relative timer
`p_ioc(P_FABSOLUTE)`	to start an absolute timer
`p_iow(P_FCANCEL)`	to cancel a timer
`p_close`	to close a timer channel

The constants `P_FRELATIVE`, `P_FABSOLUTE` and `P_FCANCEL` are defined in *p_file.h*. See the chapter *I/O System* for a description of the I/O system in general.

If you want to run multiple timers in parallel, you open a channel for each timer needed.

Although you can open a timer channel and use `p_iow(P_FRELATIVE)` or `p_iow(P_FABSOLUTE)` (rather than `p_ioc`) to the same effect as `p_sleep` or `p_sleepa` respectively, the latter should be preferred for their simplicity and efficiency.

p_open("TIM:") — Open a timer channel

```
INT p_open(VOID **pptcb,"TIM:",-1);
```

Open a timer channel and write the address of the timer control block to `*pptcb`. Returns zero if successful or the negative error number `E_GEN_NOMEMORY` if, for example, it failed to allocate memory for the control block.

p_ioc(P_FRELATIVE) — Start a relative timer

```
VOID p_ioc(VOID *ptcb, P_FRELATIVE, WORD *pstat, ULONG *pn);
```

Start a relative timer to expire after `*pn` tenths of a second.

Calls `p_panic` if a timer is already pending on channel `ptcb`.

While the timer is pending, `*pstat` contains `E_FILE_PENDING`. When the timer expires successfully, `*pstat` is set to zero and the I/O semaphore is signalled.

The timer is not started and `*pstat` is set to `E_GEN_OVER` if the conversion from tenths of a second to ticks overflows (ie the number of ticks is greater than `0x7fffffff`).

p_ioc(P_FABSOLUTE) Start an absolute timer

```
VOID p_ioc(VOID *ptcb, P_FABSOLUTE, WORD *pstat, ULONG *ptime);
```

Start an absolute timer to expire when the system time is *ptime (the number of seconds since 00:00:00, January 1, 1970).

Calls p_panic if a timer is already pending on channel ptcb.

While the timer is pending, *pstat contains E_FILE_PENDING. When the timer successfully expires, *pstat is set to zero and the I/O semaphore is signalled.

The timer is not started and *pstat is set to E_GEN_ARG if *ptime is earlier than the current system time.

p_iow(P_FCANCEL) Cancel a timer

```
INT p_iow(VOID *ptcb, P_FCANCEL);
```

Cancel a timer on channel ptcb and return zero.

The operation is harmless if no timer is pending. This is important because there is always a chance that a timer will expire before the cancel gets to it. If the cancel does get to the timer before it expires, the I/O semaphore is still signalled but *pstat is set to E_FILE_CANCEL rather than zero. However, when cancelling a timer, you don't normally care how the timer actually completed. It is common to call p_waitstat immediately after the cancel to "use up" the signal.

To reset a timer, you first cancel the timer using p_iow(P_FCANCEL) followed by a call to p_waitstat and then call p_ioc(P_FRELATIVE) to start the timer again.

p_close Close a timer channel

```
INT p_close(VOID *ptcb);
```

Close the timer channel ptcb and return zero.

You should close a timer channel when you no longer need it.

Converting between binary representations of time

The system time format sacrifices range to obtain high compression and is the most convenient and efficient representation for adding and subtracting small time intervals such as seconds, minutes, hours and days.

There are 86,400 seconds in a day (P_NSECDAY is defined as 86400L in *p_date.h*).

There are three different binary representations for time in PLIB:

- The number of seconds since 00:00:00, January 1, 1970 (system time format)
- Days since January 1, 1900 and seconds in day
- (Year since 1900,Month,Day) and (Hour,Minute,Second)

Note that the first representation works from a base year of 1970 while the second two representations work from a base year of 1900.

PLIB contains conversion routines that convert both ways between adjacent representations in the above list. By combining conversion functions, you can convert between the first and third representation.

The second format measures the days since January 1 1900 (day=0 for Jan 1) and the seconds since 00:00:00 and is stored in a P_DAYSEC struct, defined in *p_date.h* as follows:

```
typedef struct
    {
    ULONG day; /* day number since Jan 1 1900 */
    ULONG sec; /* seconds in day, 0 to 86399 */
    } P_DAYSEC;
```

The day number is useful for date calculations not involving time. Examples are to count the number of days between two dates, or to calculate a new date from a base date and a number of days (positive or negative).

The third format is closest to a human understandable representation of the date and time and is defined by the P_DATE struct in *p_date.h*:

```
typedef struct
    {
    UBYTE year;   /* Year since 1900 (0 is 1900) */
    UBYTE month;  /* Month number in year, 0 to 11 */
    UBYTE day;    /* Day number in month, 0 to 30 */
    UBYTE hour;   /* Hour in day, 0 to 23 */
    UBYTE minute; /* Minute in hour, 0 to 59 */
    UBYTE second; /* Second in minute, 0 to 59 */
    UWORD yrday;  /* Day in year */
    } P_DATE;
```

The last member, yrday is a function of year, month and day. It is provided when P_DATE is generated from P_DAYSEC but is ignored when converting from P_DATE.

p_sttods — Convert system time to P_DAYSEC time

```
VOID p_sttods(ULONG *pstim, P_DAYSEC *pds)
```

Convert *pstim from system time format (the number of seconds since 00:00:00, January 1, 1970) to the number of days since 1900 and the number of seconds in the day, both being written to pds.

p_dstost — Convert P_DAYSEC time to system time

```
INT p_dstost(P_DAYSEC *pds, ULONG *pstim);
```

Convert pds from the number of days since 1900 and the number of seconds in the day to system time format (the number of seconds since 00:00:00, January 1, 1970). The result is written to *pstim.

Returns zero if successful, or one of the following negative error numbers:

E_GEN_OVER	date in pds is too late for system time
E_GEN_UNDER	date in pds is too early for system time (ie before 1970)
E_GEN_ARG	seconds is greater than seconds in day

p_dstodt — Convert P_DAYSEC time to P_DATE time

```
INT p_dstodt(P_DAYSEC *pds, P_DATE *pdt);
```

Convert P_DAYSEC time pds (days since 1900, seconds in day) to P_DATE time pdt (year, month, day, hour, minute, second and day in year) and return zero if successful or one of the following negative error numbers:

E_GEN_OVER	the year is greater than 255 (year 2155)
E_GEN_ARG	seconds is greater than seconds in day

As well as the date and time, this function calculates pdt->yrday (the day number of the year, where day zero is January 1). Although you could calculate this yourself from the year, month and day it is not a simple calculation since it involves adding the number of days in preceding months and thus depends on leap years.

This function actually performs two independent calculations:

- converting the number of days since 1900 to year, month, day and day in year (date calculation)
- converting the number of seconds since 00:00:00 to hours, minutes and seconds (time of day calculation)

Both calculations may be abandoned if either calculation fails. If you are only interested in one of the calculations, set the other member of P_DAYSEC to zero (which is a legal input for both calculations).

p_dttods — Convert P_DATE time to P_DAYSEC time

```
INT p_dttods(P_DATE *pdt, P_DAYSEC *pds);
```

Validate the P_DATE format pdt and convert it to the P_DAYSEC format pds and return zero if successful or the negative E_GEN_ARG if the content of pdt is invalid (ie no such date or time exists). The value of pdt->yrday is ignored.

The validation can fail for one or more of the following reasons:

- pdt->month is outside the range 0 to 11
- pdt->day is outside the range for the particular month taking into account leap years for February
- pdt->hour is outside the range 0 to 23
- pdt->minute is outside the range 0 to 59
- pdt->second is outside the range 0 to 59

This function actually performs two independent calculations:

- converting the year, month and day to the number of days since January 1 1900 (date calculation)
- converting the hours, minutes and seconds to the number of seconds since 00:00:00 (time of day calculation)

Both calculations may be abandoned if either calculation fails. If you are only interested in one of the calculations, set the members of P_DATE corresponding to the other calculation to zero (zero is a legal input for all members in both calculations).

p_dayinm — Find the number of days in the specified month

```
INT p_dayinm(INT year, INT month);
```

Return the number of days in month month of year year where year is the number of years since 1900 (zero is 1900) and month is 0 to 11 inclusive. (Returns the negative E_GEN_ARG if month is greater than 11.)

The year is required because it affects the number of days in February. This may be used to test for a leap year, for example:

```
if (p_dayinm(year,1)==29)
```

p_wkday — Convert day since 1900 to day in week

```
INT p_wkday(ULONG nDay);
```

Returns the week day number given the number of days nDay since January 1 1900. The value returned is in the range 0 to 6, with 0 being Monday and 6 being Sunday.

The day number since 1900 would normally come from a P_DAYSEC struct.

p_weekno — Calculate week number in year

```
INT p_weekno(ULONG nDay);
```

Return the week number, in the range 1 to 53 inclusive, of the week containing day nDay (the number of days since January 1 1900). Returns zero if successful or E_GEN_ARG if nDay exceeds the year 2155.

The value returned is dependent on the value of the system E_CONFIG structure field startOfWeek (see p_getctd in this chapter).

Time and date components in text form

The functions in this section get a range of date and time components in text form. They constitute a more primitive set of functions than those, described in the following section, that generate strings containing full date and/or time representations.

These functions are based on language-dependent information that is built into the ROM. If you use these functions, your code should automatically work on, for example, English, French and German machines.

The functions are:

p_nmday	to get the day names (eg Monday, Tuesday)
p_nmmon	to get the month names (eg January, February)
p_nmdaya	to get the day name abbreviations (eg Mon, Tue)
p_nmmona	to get the month name abbreviations (eg Jan, Feb)
p_getsuffixes	to get the 31 day in month number suffixes (eg st, nd)
p_getampmtext	to get the am and pm suffixes
p_getctd	to get a copy of the setable country-dependent data and time preferences (such as whether to use a 12 or 24 hour clock)

The following example displays the system time in the form:

```
Monday, 11th April 1988   11:03
```

and uses many of the functions described in this chapter.

```
#include <plib.h>

GLDEF_C VOID PrintDateTime()
    {
    ULONG st;
    P_DAYSEC ds;
    P_DATE dt;
    E_CONFIG cfg
    TEXT DayName[32], TEXT MonthName[32], TEXT Suffix[31][3];

    st=p_date();
    p_sttods(&st,&ds);
    p_dstodt(&ds,&dt);
    p_nmday(&DayName[0],p_wkday(ds.day));
    p_nmmon(&MonthName[0],dt.month));
    p_getsuffixes(&Suffix[0][0]);
    p_getctd(&cfg);
    p_printf("%s, %u%s %s %u   %02u%c%02u",
        &DayName[0],dt.day+1,&Suffix[dt.day][0],
        &MonthName[0],dt.year+1900,
        dt.hour,cfg.timeSeparator,dt.minute);
    }
```

This example is somewhat artificial since the same action can be performed more simply by the use of p_nowtostr, described in the following section.

Object-oriented programmers may prefer to use the time class in OLIB (described in the *OLIB Reference* manual of the *SIBO SDK Object Oriented Extension*) to produce textual representations of the date and time.

p_nmday — Get the day name

```
INT p_nmday(TEXT *buf, INT daynum);
```

Write the language dependent name of day daynum as a zero terminated string to buf where daynum should be in the range 0 to 6 inclusive and day 0 is Monday. Returns zero if successful or E_GEN_ARG if daynum is not in the range 0 to 6.

If you are working in a fixed language, you will know how long the longest day name is. If you are writing a program that has to work with different language ROMs, you can use the fact that the tool that builds the day name list for the ROM limits a particular day name to E_MAX_DAY_NAME (32), including the zero terminator.

p_nmdaya — Get the day name abbreviation

```
INT p_nmdaya(TEXT *buf, INT daynum);
```

This function is only available in EPOC version 3.18 or later.

Write the language dependent abbreviation for the name of day daynum as a zero terminated string to buf where daynum should be in the range 0 to 6 inclusive and day 0 is Monday. Returns zero if successful or E_GEN_ARG if daynum is not in the range 0 to 6.

All day name abbreviations for a particular language are of the same length and could be one, two or three characters. In no language will day name abbreviations exceed three characters.

p_nmmon — Get the month name

```
INT p_nmmon(TEXT *buf, INT monthnum);
```

Write the language dependent name of month monthnum as a zero terminated string to buf where monthnum should be in the range 0 to 11 inclusive and month 0 is January.

Returns zero if successful or E_GEN_ARG if monthnum is not in the range 0 to 11.

If you are working in a fixed language, you will know how long the longest month name is. If you are writing a program that has to work with different language ROMs, you can use the fact that the tool that builds the month name list for the ROM limits a particular month name to E_MAX_MONTH_NAME (32), including the zero terminator.

p_nmmona — Get the month name abbreviation

```
INT p_nmmon(TEXT *buf, INT monthnum);
```

This function is only available in EPOC version 3.18 or later.

Write the language dependent abbreviation of the name of month monthnum as a zero terminated string to buf where monthnum should be in the range 0 to 11 inclusive and month 0 is January.

Returns zero if successful or E_GEN_ARG if monthnum is not in the range 0 to 11.

All month name abbreviations for a particular language are of the same length and could be one, two or three characters. In no language will month name abbreviations exceed three characters.

p_getsuffixes — Get the day-in-month suffixes

```
VOID p_getsuffixes(TEXT *buf);
```

Write the language dependent array of the 31 day-in-month number suffixes to buf.

The suffixes are written as an array of 31 3-byte fixed length elements where each element contains 0, 1 or 2 characters followed by a zero terminator (suffixes contain at most 2 characters in any language). There must be at least 31*3 bytes of memory at buf.

Each element in the array contains the suffix for the corresponding day of the month. For example, in English, the first element contains "st" and the second element contains "nd".

p_getampmtext — Get the am and pm suffixes

```
VOID p_getampmtext(TEXT *buf, INT n);
```

Write the language dependent am or pm time suffix as a zero terminated string to buf. If n is 0, write the am suffix. If n is 1, write the pm suffix.

The am and pm suffixes are limited to 2 characters in any language.

p_getctd — Get time representation preferences

```
VOID p_getctd(E_CONFIG *pcfg);
```

Write a copy of the system E_CONFIG struct to pcfg where the E_CONFIG struct is defined, in *p_config.h*, as:

```
typedef struct
    {
    UWORD countryCode;
    WORD gmtOffset;
    UBYTE dateType;
    UBYTE timeType;
    UBYTE currencySymbolPosition;
    UBYTE currencySpaceRequired;
    UBYTE currencyDecimalPlaces;
    UBYTE currencyNegativeInBrackets;
    UBYTE currencyTriadsAllowed;
    UBYTE thousandsSeparator;
    UBYTE decimalSeparator;
    UBYTE dateSeparator;
    UBYTE timeSeparator;
    UBYTE currencySymbol[9];
    UBYTE startOfWeek;
    UBYTE summerTime;
    UBYTE clockType;
    UBYTE dayAbbreviation;
    UBYTE monthAbbreviation;
    UBYTE workDays;
    UBYTE units;
    UBYTE spare[9];
    } E_CONFIG;
```

In the context of this chapter we are interested in the following items:

gmtOffset	the offset in minutes of the local system time from Greenwich Mean Time.
dateType	one of E_DATE_USA for MM/DD/YY, E_DATE_EUROPE for DD/MM/YY or E_DATE_JAPAN for YY/MM/DD.
timeType	either E_TIME_12 for a 12 hour clock or E_TIME_24 for a 24 hour clock.
dateSeparator	the character code of the date separator. For example, the character '/'.
timeSeparator	the character code of the time separator. For example, the character ':'.
startOfWeek	the day number (in the range 0 to 6 inclusive where day 0 is Monday) of the first day in the week - as used by p_weekno. This is normally either Sunday (day 6) or Monday (day 0). For example, it is normally Sunday for USA machines and Monday for UK machines.
summerTime	a bit pattern indicating summer time-zones as follows: E_DST_HOME if the system time should be adjusted for summer time; E_DST_EUROPEAN if a European time-zone should be adjusted for summer time; E_DST_NORTHERN if a non-European time-zone in the Northern hemisphere should be adjusted for summer time; E_DST_SOUTHERN if a time-zone in the Southern hemisphere should be adjusted for summer time.
clockType	either E_ANALOGUE_CLOCK to indicate a preference for an analogue clock display or E_DIGITAL_CLOCK to indicate a preference for a digital clock display
dayAbbreviation	how many leading characters to take from the day name (as returned by p_nmday) to abbreviate the day name
monthAbbreviation	how many leading characters to take from the month name (as returned by p_nmmon) to abbreviate the month name
workDays	a bit mask of 7 bits indicating (by being set) which days are to be considered work days where the least significant bit corresponds to Monday

The data may be set using p_setctd - as described in the chapter *General System Services*.

Generating time and/or date strings

The functions in this section construct text strings that represent the date and time in a range of formats.

The functions are based on language-dependent information that is built into the ROM, in particular, the system `E_CONFIG` struct (see the description of `p_getctd` in the *Language and country* section of the *General System Services* chapter). If you use these functions, your code should automatically work on, for example, English, French and German machines.

Note that the implementation of these functions uses static data. In consequence they may not be used in the code of a dynamic library (DYL).

Date and time format strings

The form of the output of the date and time conversion functions described below is controlled by a format string, in a similar way to the format strings used by `p_printf`, `p_atob` and `p_atos`.

The format string consists of literal text intermixed with embedded commands. The literal text is simply copied to the output and the embedded commands are replaced by the corresponding time or date element.

An embedded command is of the form `%c` or `%*c`, where `c` is one of the characters listed below, and `*` indicates that the output should be in an abbreviated form. The abbreviated form can be specified for all commands, but in some cases there is no difference between the full and abbreviated forms.

The available commands are as follows:

`%%`	replaced by a single '%' character. Abbreviation has no effect.
`%:`	replaced by the time separator character as specified by the `timeSeparator` field of the system `E_CONFIG` struct. Abbreviation has no effect.
`%/`	replaced by the date separator character as specified by the `dateSeparator` field of the system `E_CONFIG` struct. Abbreviation has no effect.
`%A`	depending on the supplied time of day, this is replaced by the appropriate am or pm text (as obtained by use of `p_getampmtext`). Abbreviation supplies just the first character of this text. Note that abbreviation may not be appropriate in some languages.
`%D`	replaced by the day-in-month number, in the range 01 to 31, as two digits with a leading zero as necessary. Abbreviation suppresses any leading zero.
`%E`	replaced by the day name, as supplied by `p_nmday`. Abbreviation causes the name to be truncated to the number of characters specified in the `dayAbbreviation` element of the system `E_CONFIG` struct.
`%H`	replaced by two digits in the range 00 to 23 (24-hour format) corresponding to the hours component of the supplied time of day. Abbreviation suppresses any leading zero.
`%I`	replaced by two digits in the range 01 to 12 (12-hour format) corresponding to the hours component of the supplied time of day. Abbreviation suppresses any leading zero.
`%M`	replaced by two digits in the range 01 to 12 corresponding to the month number for the supplied date. Abbreviation suppresses any leading zero.
`%N`	replaced by the month name for the supplied date. Abbreviation causes the name to be truncated to the number of characters specified in the `monthAbbreviation` element of the system `E_CONFIG` struct.
`%S`	replaced by two digits in the range 00 to 59 corresponding to the seconds component of the supplied time of day. Abbreviation suppresses any leading zero.
`%T`	replaced by two digits in the range 00 to 59 corresponding to the minutes component of the supplied time of day. Abbreviation suppresses any leading zero.

%W replaced by two digits in the range 01 to 53 corresponding to the week number for the supplied date, as provided by p_weekno. Abbreviation suppresses any leading zero.

%X replaced by the suffix text corresponding to the (%D) day number for the supplied date. Abbreviation has no effect.

%Y replaced by four digits, in the range 1900 to 2155, corresponding to the year number for the supplied date. Abbreviation discards the first two digits (note that, for example, both 1900 and 2000 will appear as 00).

%Z replaced by three digits, in the range 001 to 366, corresponding to the day-of-year number for the supplied date. Abbreviation discards any leading zeros.

%1 replaced by the first component of a three-component (day, month and year) date, where the order of the components is determined by the value of the dateType component of the system E_CONFIG struct. Abbreviation has no effect, but the form of the generated text is conditioned by the toggle commands described below. In the absence of any toggles, %1 is equivalent to:

%D if dateType is E_DATE_EUROPE
%M if dateType is E_DATE_USA
%Y if dateType is E_DATE_JAPAN

%2 replaced by the second component of a three-component (day, month and year) date, where the order of the components is determined by the value of the dateType component of the system E_CONFIG struct. Abbreviation has no effect, but the form of the generated text is conditioned by the toggle commands described below. In the absence of any toggles, %2 is equivalent to:

%M if dateType is E_DATE_EUROPE
%D if dateType is E_DATE_USA
%M if dateType is E_DATE_JAPAN

%3 replaced by the third component of a three-component (day, month and year) date, where the order of the components is determined by the value of the dateType component of the system E_CONFIG struct. Abbreviation has no effect, but the form of the generated text is conditioned by the toggle commands described below. In the absence of any toggles, %3 is equivalent to:

%Y if dateType is E_DATE_EUROPE
%Y if dateType is E_DATE_USA
%D if dateType is E_DATE_JAPAN

%4 replaced by the first component of a two-component (day and month) date, where the order of the components is determined by the value of the dateType component of the system E_CONFIG struct. Abbreviation has no effect, but the form of the generated text is conditioned by the toggle commands described below. In the absence of any toggles, %4 is equivalent to:

%D if dateType is E_DATE_EUROPE
%M if dateType is E_DATE_USA or E_DATE_JAPAN

%5 replaced by the second component of a two-component (day and month) date, where the order of the components is determined by the value of the dateType component of the system E_CONFIG struct. Abbreviation has no effect, but the form of the generated text is conditioned by the toggle commands described below. In the absence of any toggles, %5 is equivalent to:

%M if dateType is E_DATE_EUROPE
%D if dateType is E_DATE_USA or E_DATE_JAPAN

%6 replaced by the hour in the format determined by the value of the timeType component of the system E_CONFIG struct. Abbreviation discards any leading zero. The action of %6 is equivalent to:

%H if timeType is E_TIME_24
%I if timeType is E_TIME_12

%7 replaced by the am/pm text (as for %A) if the timeType component of the system E_CONFIG struct has the value E_TIME_12, otherwise produces no output. If output is produced, abbreviation reduces the output to just the first character of the text, as for %A.

Note that format strings of the form "%1%/%2%/%3" and "%4%/%5" respectively generate three- and two-component dates that automatically conform to the system configuration dateType setting, and that a

format string of the form "%6%:%T%:%S%7" generates a time that automatically conforms to the system configuration timeType setting.

The commands %1, %2, %3, %4 and %5 are conditioned by the following toggles:

%F — produces no output, but toggles the behaviour of subsequent day items between numeric (the default) and name generation. For example, if dateType is E_DATE_EUROPE and the supplied date is 09/03/1993, the format string "%1 %F%1 %F%1" will generate the (English) string "09 Tuesday 09". Abbreviation has no meaning, and is ignored. The items conditioned by this toggle depend on dateType as follows:

- %1 and %4 for E_DATE_EUROPE
- %2 and %5 for E_DATE_USA
- %3 and %5 for E_DATE_JAPAN

%O — produces no output, but toggles the behaviour of subsequent month items between numeric (the default) and name generation. For example, if dateType is E_DATE_EUROPE and the supplied date is 09/03/1993, the format string "%2 %O%2 %O%2" will generate the (English) string "03 March 03". Abbreviation has no meaning, and is ignored. The items conditioned by this toggle depend on dateType as follows:

- %2 and %5 for E_DATE_EUROPE
- %1 and %4 for E_DATE_USA
- %2 and %4 for E_DATE_JAPAN

%G — produces no output, but toggles the behaviour of subsequent day items between their full (the default) and their abbreviated forms. For example, if dateType is E_DATE_EUROPE and the supplied date is 09/03/1993, the format string "%F%1 %G%1 %G%1" will generate the (English) string "Tuesday Tue Tuesday". Abbreviation has no meaning, and is ignored. The items conditioned by this toggle depend on dateType as follows:

- %1 and %4 for E_DATE_EUROPE
- %2 and %5 for E_DATE_USA
- %3 and %5 for E_DATE_JAPAN

%P — produces no output, but toggles the behaviour of subsequent month items between their full (the default) and their abbreviated forms. For example, if dateType is E_DATE_EUROPE and the supplied date is 09/03/1993, the format string "%O%2 %P%2 %P%2" will generate the (English) string "March Mar March". Abbreviation has no meaning, and is ignored. The items conditioned by this toggle depend on dateType as follows:

- %2 and %5 for E_DATE_EUROPE
- %1 and %4 for E_DATE_USA
- %2 and %4 for E_DATE_JAPAN

%U — produces no output, but toggles the behaviour of subsequent year items between their full (the default) and their abbreviated forms. For example, if dateType is E_DATE_EUROPE and the supplied date is 09/03/1993, the format string "%3 %U%3 %U%3" will generate the string "1993 93 1993". Abbreviation has no meaning, and is ignored. The items conditioned by this toggle depend on dateType as follows:

- %3 for E_DATE_EUROPE
- %3 for E_DATE_USA
- %1 for E_DATE_JAPAN

%L — produces no output, but toggles between the absence (the default) and the presence of a day number suffix following a day-in-month number (but not a day name) generated by %1, %2 or %3. For example, if dateType is E_DATE_EUROPE and the supplied date is 09/03/1993, the format string "%G%1 %L%1 %L%1" will generate the (English) string "9 9th 9", but "%F%G%1 %L%1 %L%1" will generate the (English) string "Tue Tue Tue". Abbreviation has no meaning, and is ignored.

Function return values

The Window Server animates time displays either by flashing the last time separator character in the text or, if the text contains a seconds display, by updating the time every second.

All the functions described below return a value that indicates the form of animation required. The possible return values are:

-2	The display does not show seconds and contains no time separators. The display is not animated.
-1	The display shows seconds. Animation updates the display every second.
any other value	The return value is the offset in the string to the last time separator character. Animation flashes this character.

p_dt2str — Convert a P_DATE time to a string

```
INT p_dt2str(TEXT *buf, TEXT *fstr, P_DATE *pdt);
```

Write a zero terminated string containing formatted date and time text to `buf`, controlled by the zero terminated format string pointed to by `fstr` and the content of the `P_DATE` struct pointed to by `pdt`.

The format string `fstr` contains literal text, embedded with commands, as described above.

p_ds2str — Convert a P_DAYSEC time to a string

```
INT p_ds2str(TEXT *buf, TEXT *fstr,P_DAYSEC *pds);
```

Write a zero terminated string containing formatted date and time text to `buf`, controlled by the zero terminated format string pointed to by `fstr` and the content of the `P_DAYSEC` struct pointed to by `pds`.

The format string `fstr` contains literal text, embedded with commands, as described above.

p_st2str — Convert a system time to a string

```
INT p_st2str(TEXT *buf, TEXT *fstr, ULONG *pst);
```

Write a zero terminated string containing formatted date and time text to `buf`, controlled by the zero terminated format string pointed to by `fstr` and the system time (seconds since 00:00:00, January 1, 1970) pointed to by `pst`.

The format string `fstr` contains literal text, embedded with commands, as described above.

p_now2str — Convert the current time to a string

```
INT p_now2str(TEXT *buf, TEXT *fmt);
```

Write a zero terminated string containing formatted date and time text to `buf`, controlled by the zero terminated format string pointed to by `fstr` and the currently set time.

The format string `fstr` contains literal text, embedded with commands, as described above.

CHAPTER 11

FILES

Files in EPOC

The file server

The file server is a high priority system process (with process name SYS$FSRV) performing all file related operations on behalf of "client" processes.

An application process must connect to the file server before using its services. However, this is normally taken care of by the C startup module (the code that precedes `main`) and supplied as standard for use with the PLIB library. (Most applications need the services of the file server and the overhead of connecting to the file server is modest.)

A client process sends the file server an inter-process message to request a file server service. However, the application programmer does not program at the message passing level but uses the interface provided by the `FIL:` device (ie using `p_open`) together with the interface provided by a number of ROM resident functions (eg `p_delete` to delete a file) as described in this chapter.

Any process that attempts to send a message to the file server without having connected is panicked with panic number 41.

File systems

The file server supports multiple file systems (also called nodes). In principle, the file server supports any number of file systems (`p_open` may be used to get a list of the file systems as described later in this chapter). At the time of writing, three file systems have been implemented:

`LOC::`	The local filing system with devices M: (the RAM drive) and SSD drives A:, B:, ...(where the quantity depends upon the hardware).
`REM::`	The remote filing system, available while the file server is connected to a remote file server. The structure of the filing system depends upon what the remote system is. If the remote system is a PC or another SIBO machine, the structure is the same as for `LOC::`.
`ROM::`	The ROM filing system, used to access ROM-based files. This filing system is not normally visible to the user. The `ROM::` file system does not support devices or directories.

Within the `LOC::` file system (and where `REM::` is connected to a PC or another SIBO machine), the devices and directories structure is compatible with the MSDOS filing system.

File systems are implemented as PDDs (Physical Device Drivers) which are normally resident in the ROM (PDDs are described in the chapter *I/O System*). If you use (as described in *I/O System*):

```
GLDEF_C VOID PrintPDDs(VOID)
    {
    HANDLE h;
    TEXT b[E_MAX_NAME+2];

    for (h=0;(h=p_devfnd(h,"*",E_PDD,&b[0]))>=0;)
        p_printf("  %s",&b[0]);
    }
```

to get a list of PDDs, the list would include:

FSY.LOC	the LOC:: PDD
FSY.ROM	the ROM:: PDD
FSY.REM	the REM:: PDD

SSD drives

SIBO machines have 2 to 4 (depending on the machine) Solid State Disk (SSD) drives, taking SSDs of various type and capacity (from 32K bytes to 8Mb and beyond). SSDs are so called because they are based on silicon memory with no moving parts. The different types of SSD (in order of decreasing unit cost) include:

- static RAM with integral Lithium battery
- Flash EPROM
- one-time-programmable ROM
- masked ROM

An SSD drive can physically and logically mount any type of SSD.

The physical interface between an SSD and the drive has only 6 connectors (4 for power, and 2 for data). The 2 data connections are an instance of a *high speed serial* channel. This is a fundamental part of the SIBO hardware architecture, implemented in custom chips. It can be used to communicate with other peripherals.

The high speed serial channel is driven synchronously by a clock which normally runs at 3.84 MHz, giving a data transfer speed comparable to the more expensive hard disks on PCs but, without any latency for the drive head to position to the required track. Writing to a Flash SSD is slower because of the time taken to program the EPROM and, in this case, the speed of writing is twice as fast as writing to a typical floppy disk on a PC.

SSDs are driven by the LOC:: file system using a number of subsidiary PDDs (Physical Device Drivers) handling the different SSD types and the internal M: device. The list of PDDs produced by calling PrintPDDs (described above) would include:

LOC.TY0	the RAM SSD PDD
LOC.TY1	the Flash SSD PDD
LOC.TYM	the internal RAM (M:) PDD

Files are stored in a completely different way on RAM SSDs from Flash SSDs.

SSD drive doors have a switch that keeps the file server informed of possible SSD removals and insertions. After an SSD drive door has been opened the file server checks each drive to see if it contains a new SSD. When the file server detects a new SSD it automatically mounts the new SSD.

If a previous occupant of the SSD drive had one or more files open on it, the file server keeps a record of the SSD. When access is subsequently attempted on a file channel on that SSD, the file server uses p_notify (described in the chapter *Error Handling*) to request the user to replace the SSD and to retry or to abandon the channel. If the user replaces the SSD and selects retry such that the operation completes successfully, the application process is not made aware of any problem. If the user chooses to abandon the channel, the operation completes with the error E_FILE_ABORT. When this occurs, the channel is disconnected from the file and put into an abort state where the only possible operation is to close the channel (any operation other than p_close fails with the error E_FILE_ABORT).

An application that receives an E_FILE_ABORT error can either make do without the file (which probably means having to exit) or to appeal yet again to the user to put the SSD back in the drive and, if the user does put the SSD back, to locate and re-open the file and recover. The latter takes more code but is kinder to the user.

Unattended applications

The file server will also normally call p_notify to give the user a chance to retry any file operation that fails on a mounted medium. This kind of failure is rare on SSDs but is more common on magnetic media - especially floppy disks. In some cases, the user may be able to correct the error (eg to close the door on a floppy disk drive) and successfully retry. If the user abandons, the file operation completes with an error other than E_FILE_ABORT (eg E_FILE_READ if a floppy disk read fails).

This scheme whereby the file server uses p_notify to give the user a chance to correct the problem and retry works well when the client process is an interactive application but poorly if the requesting process is designed to run unattended (eg a communications program) or is itself a server process (eg the window server trying to read a font file). Such processes can use p_setnotify(FALSE), described in the chapter *Error Handling*, to stop the file server from using the notify service. In this case all errors are returned directly.

RAM SSDs

RAM SSDs use static RAM backed up by an integral Lithium battery. While the RAM SSD is inserted in an SSD drive, it draws current from the SIBO machine. When the SSD is removed, it relies on its internal battery to maintain its data.

The allocation of the space for directories and files on a RAM SSD is block structured - identical to that which is used on PC hard disks and floppy disks. Just like floppy disks on a PC, there is a limit to the number of files that may appear in the root directory (where this limit includes directory files). This root directory limit does not limit the number of files that can be stored on a RAM SSD since any number of files may be stored in subdirectories.

The format function (described below in this chapter) automatically allocates the capacity of the root directory as a function of the capacity of the SSD. The dependence of the number of files that may appear in the root directory on SSD size is as follows:

Files	RAM SSD size
16	less than 128K
32	128K or greater but less than 256K
48	256K or greater but less than 512K
64	512K or greater but less than 1M
128	1M or greater but less than 2M
256	2M or greater but less than 4M
384	4M
512	6M
512	8M

The RAM SSD PDD on the LOC:: file system on SIBO machines does not buffer written data. This protects the integrity of the SSD contents from being corrupted by application process or system crashes occurring while files are open, or by the removal of the SSD while files are open on it.

Flash SSDs

Flash SSDs use Flash EPROM chips in which all the data bits are initially set. A particular bit may be cleared by programming but it can not be set again unless all the bits on the chip are set (by erasing the chip). Unlike older generation EPROMs, Flash EPROMs are erased electrically (rather than by exposure to UV light). The speed with which Flash can be programmed is also substantially faster than the old UV erasable EPROMS.

A Flash SSDs may be erased in its SSD drive by formatting (formatting is described later in this chapter).

Since Flash memory does not require a battery to maintain it, the data on a Flash SSD is very secure - more so than on RAM SSDs or magnetic media. Flash memory is also cheaper than RAM.

The Flash filing system is designed such that the logical interface to files on a Flash SSD (whether reading or writing) is entirely equivalent to that on other devices (RAM SSD, hard disk etc). However, when overwriting or when deleting a file, space is consumed and is not recovered until the SSD is next formatted. Note also that renaming a file, setting the date and time or setting the file attributes all use up additional space and can therefore fail through lack of remaining capacity (the function returns E_FILE_FULL). However, deleting a file does not consume any further space.

You can randomly access a Flash file and overwrite sections of it in exactly the same way as for any other file. However, in contrast to read/write block storage devices, overwriting incurs a storage overhead. For example, the following code:

```
pos=0L;

do
    {
    p_seek(chan,F_FABS,pos);
    } while (!p_write(chan,"Pointless",8));
```

continuously overwrites the first 8 bytes of the file on channel chan. On a RAM file, this would work indefinitely and just exercise the hardware. On a Flash file, the code would work but each iteration would use up space on the Flash SSD and the p_write would eventually fail and return E_FILE_FULL.

In practice, you should not worry about limited overwriting of file data - for example, to patch a program file or to update a header. However, you cannot reasonably repeatedly and indefinitely overwrite a Flash file and, for example, Flash SSDs are unsuitable for B-tree index files.

See the chapter *Database Files* for a description of a Flash-friendly set of functions allowing random access to, and the deleting and updating of, variable length records. These functions take advantage of the fact that a byte can be physically overwritten (with no storage penalty) provided that the overwrite is such that each bit in the byte either stays the same or is cleared.

Flash SSDs are particularly attractive for storing program files, read-only data (say for data-referral applications), for securely storing logged data in the field and for archiving data.

The storage of files on a Flash SSD is very different from that used on RAM SSDs. The scheme is not at all block structured but is based on linked variable length records. All this is hidden from the caller and the logical interface to a file on a Flash SSD is the same as for a file on a RAM SSD (or any other medium).

The data in each write to a file on a Flash SSD is written straight to the SSD (ie it is not buffered)[1]. However, in the interests of efficient storage, the record just written is kept open (with a `0xffff` length) for as long as possible - until the file channel is closed or flushed using `P_FFLUSH` or until a write to another file on the same SSD occurs. The date and time stamp is also not written until the file channel is flushed or closed.

If the SSD is removed or the machine resets while there is an open file channel with an outstanding write, the record is closed when the file is next accessed (the file is then also stamped with the current date and time).

On a Flash SSD, there is no limit on the number of files or directories in the root directory. You will also find that it is possible to fit slightly more data on a freshly formatted Flash SSD than on a RAM SSD of the same nominal capacity. This is due to the fact that files are allocated space in multiples of whole blocks on a RAM SSD whereas Flash files are stored in exactly sized variable length records.

File specifications

A full file specification has the form:

```
<node><device><dir><name><ext>
```

where the components are:

`<node>`	the file system node (eg `LOC::`)
`<device>`	the device name (eg `B:`)
`<dir>`	the directory name (eg `\NOTES\OLD\`)
`<name>`	the file name (eg `PLANS`).
`<ext>`	the extension name (eg `.TPD`)

An example of a full file specification is:

```
LOC::B:\NOTES\OLD\PLANS.TPD
```

Although the file server assumes that a file specification may be decomposed into a `<node>`, `<device>`, `<dir>`, `<name>` and `<ext>`, it avoids any assumption of the detailed syntax of the `<device>`, `<dir>`, `<name>` and `<ext>` components. This is left to the `<node>` file system code that implements the file specification manipulation functions `p_fparse` and `p_chdir`.

Leaving such detailed considerations to the file system is designed to allow the file stores of remote foreign file systems to be mapped transparently to the file server model (which is compatible with the MSDOS filing system). For example, in the VMS operating system, the above example of a file specification might translate to:

```
REM::USER:[NOTES.OLD]PLANS.TPD
```

and on an Apple Macintosh, it might be:

```
REM::HD40:NOTES:OLD:PLANS.TPD
```

[1]This may not be the case for text files, which are handled by a layer over the `FIL:` device. The buffering of such files is thus outside the file server's control.

and on Unix, it might be:

```
REM::USER::NOTES/OLD/PLANS.TPD
```

To prepare for such foreign systems, applications should follow the example of the file server and avoid making assumptions about the syntax of file specifications and use functions such as p_fparse and p_chdir to manipulate file specifications.

File specifications satisfy the following rules:

- a file specification will not require more than P_FNAMESIZE (128) bytes, including a zero terminator
- the <node> component is always P_FSYSNAMESIZE bytes long (excluding any zero terminator)

where P_FNAMESIZE and P_FSYSNAMESIZE are defined in *p_file.h*. Except for <node> and the P_FNAMESIZE total, you should not make any assumptions about the maximum size of the components of a file specification.

Default path

The file server stores a default node, device and directory, also called the default path, for each of its clients. In addition, the file server stores a lower level default path, the system-wide default path. This is the default path assigned to new clients when they connect to the file server.

A process that is a client of the file server uses:

p_setpth	to set its default path
p_getpth	to get a copy of its default path

A process (normally a system process such as the shell) can change the system-wide default path - the path subsequently assigned to connecting clients - by calling p_setdefaultpath.

Channel-based services

A client of the file server can use p_open on the FIL: device with different values of mode to do the following:

P_FSTREAM, P_FSTREAM_TEXT	to open a file and manipulate it as a flat binary file
P_FTEXT	to open a file and manipulate it as a record-oriented text file
P_FDIR	to get a list of the files and subdirectories in a directory
P_FDEVICE	to get a list of the available devices
P_FNODE	to get a list of the available file systems
P_FFORMAT	to format a LOC:: device

Once a channel has been opened, one or more I/O functions (depending on mode) may be requested synchronously using p_iow (or a convenience function such as p_read) or asynchronously using p_ioc or p_ioa.

There is no practical limit to the number of file channels that may be opened in the system or by a particular process.

Non-channel-based services

These are file operations not involving the FIL: device driver. They are requested by the following function calls:

p_fparse, p_fparseasync	to parse a file specification
p_chdir, p_chdirasync	to change the directory component of a file specification
p_ninfo, p_ninfoasync	to get file system node information
p_dinfo, p_dinfoasync	to get information on the medium in a device

`p_finfo`, `p_finfoasync`	to get information on a file or a directory
`p_rename`, `p_renameasync`	to rename a file or a directory
`p_delete`, `p_deleteasync`	to delete a file or a directory
`p_mkdir`, `p_mkdirasync`	to make a new directory
`p_sfstat`, `p_sfstatasync`	to set the attributes of a file (and to set the volume label of a medium)
`p_fdate`, `p_fdateasync`	set the modification date and time of a file

The name of asynchronous equivalents are generated by appending `async` to the name of the corresponding synchronous function. The asynchronous function takes the same parameters as the synchronous function but with the addition of a status word parameter at the end of the parameter list.

Asynchronous file operations

The file server is essentially asynchronous in its operation and most functions are provided in both asynchronous and synchronous forms.

Because the file server has a higher priority than any of its clients, any operation that does not wait for a slow external device will have completed by the time an asynchronous request has returned. This includes any operation on the `ROM::` file system or where the `LOC::` file system accesses SSDs or the internal RAM drive (`M:`). However, when accessing the `REM::` file system where the connection is via an RS232 cable, any asynchronous request is almost certain to return before the operation is complete.

Although, in practice, most file operations complete in a fraction of a second, a particular file request may take an extended time. For example, a read of 8K bytes from a `REM::` file, connected over a 1200 baud modem link, would take well over a minute. However, a request will never take an indefinite time to complete (as, for example, a write to the parallel port can do when the printer is off line).

For these reasons, the file server does not actively support a cancel service. (Incidentally, it is also worth considering the state in which a file would be left if a partially completed write operation were cancelled.)

A request on an open file channel *may* be cancelled using `p_iow(P_FCANCEL)` but the cancel will not cause the operation to complete any sooner (and it will not complete with `E_FILE_CANCEL`).

In general, it is preferable to *simulate* a cancel by waiting for current service to complete and forcing the status word to `E_FILE_CANCEL`, as illustrated below.

The cancelling of an asynchronous request such as:

```
p_ioc(pcb,P_FREAD,&stat,buf,len);
```

may be simulated by:

```
if (stat==E_FILE_PENDING)
    {
    p_waitstat(&stat);    /* wait for completion */
    stat=E_FILE_CANCEL;   /* report a cancel */
    }
```

This will normally complete almost immediately. You should, however, be aware that in rare cases when using the remote filing system (say, when the remote computer is re-booted and the communications link is set up to have a long timeout period) it may take an extended time to complete.

Manipulating file specifications

p_fparse (or f_fparse) Parse a file specification

```
INT p_fparse(TEXT *name, TEXT *related, TEXT *full, P_FPARSE *pcrk);
INT f_fparse(TEXT *name, TEXT *related, TEXT *full, P_FPARSE *pcrk);
INT p_fparseasync(TEXT *name, TEXT *related, TEXT *full, P_FPARSE *pcrk, WORD *stat);
```

Builds a full file specification of the form:

```
<node><device><dir><name><ext>
```

as a zero terminated string in `full`, where the components are:

`<node>`	the file system node (eg `LOC::`)
`<device>`	the device name (eg `B:`)
`<dir>`	the directory name (eg `\NOTES\OLD\`)
`<name>`	the file name (eg `PLANS`).
`<ext>`	the extension name (eg `.TPD`)

The parameters `name` and `full` may point to the same address, but `related` and `full` should not have the same address.

There should be at least `P_FNAMESIZE` (128) bytes of memory reserved at `full`. Note that `P_FNAMESIZE` bytes are *always* written to `full` even when the generated file specification is shorter. If you have a name produced by a previous call to `p_fparse` and you are calling `p_fparse` again to fill in the struct at `pcrk`, you still need `P_FNAMESIZE` bytes at `full`.

Subject to the restrictions described below, under the heading *Using p_fparse across filing systems*, the components making up the full file specification are taken from the following (in order of precedence):

- the zero terminated file specification `name`
- the zero terminated related file specification `related` (the related file specification may be omitted by passing `NULL`).
- the default path (the node, device and directory as set by `p_setpth`)

Thus components are only taken from `related` if they are missing from `name`. If there are still missing components, they are taken from the default path. Note that the `<name>` and `<ext>` fields in `name` and `related` may contain wildcard characters.

The output file specification in `full` is converted to upper case characters. Prior to EPOC version 2.31, conversion to upper case was performed by folding (as by using `p_tofold`). From EPOC version 2.31 onwards the conversion is by converting to upper case (as by using `p_toupper`). This has no significant effect on UK applications, but improves the handling of file names containing, for example, accented characters.

Further information on the content of `full` is written to the `P_FPARSE` struct `pcrk`. If this information is not required, `pcrk` may be passed as `NULL`. If `pcrk` is not `NULL`, it should be the address of a `P_FPARSE` struct, defined in *p_file.h* as follows:

```
typedef struct
    {
    UBYTE system;  /* file system name length */
    UBYTE device;  /* device name length */
    UBYTE path;    /* path name length */
    UBYTE name;    /* name length */
    UBYTE ext;     /* extension length */
    UBYTE flags;   /* information on the presence of wildcards */
    } P_FPARSE;
```

The members system, device, path, name and ext are set to the lengths of their corresponding fields in full, including delimiters. The flags field contains information on the occurrence of wild card characters in the extended specification according to the following masks:

P_PWILD_ANY	if set, the file specification contains one or more wild card characters and at least one of the following bits are set
P_PWILD_NAME	if set, the name field contains one or more wild card characters
P_PWILD_EXT	if set, the extension contains one or more wild card characters

The function returns zero if successful or one of the following negative error numbers:

E_FILE_NAME	either name or related contains an invalid name
E_FILE_DEVICE	either name or related contains an invalid device name
E_FILE_DIR	either name or related contains an invalid directory name
E_GEN_FSYS	either name or related contains an invalid file system name or the file system does not exist

The function f_fparse is identical to p_fparse except that it calls p_leave (passing the error number) rather than return a negative error number.

As an example, if the current default path is "LOC::A:\", then

```
p_fparse("FRED","\\FILES\\DOCS\\JOE.DOC",buf,&crk);
```

writes the zero terminated string LOC::A:\FILES\DOCS\FRED.DOC to buf and {5,2,12,4,4,0} to crk.

The function is often used to change the extension of a file as, for example, in:

```
p_fparse(".BCK","\\FILES\\DOCS\\JOE.DOC",buf,&crk);
```

Using p_fparse across filing systems

Since the syntax of a file specification may differ between filing systems (see *File specifications*, earlier in this chapter) the functions p_fparse and p_chdir are implemented in the code of each file system. The function p_fparse builds a full file specification from up to three sources, which may therefore specify two or more different filing system nodes.

This section describes the rules that determine which implementation performs the function p_fparse in any particular case, and which of the three sources contribute towards the full file specification. The three sources will be referred to as name, related and the default path, as in the description of p_fparse.

The node which performs the p_fparse is determined from name, related and the default path, according to the following rules:

- If neither name nor related contain an explicit <node> component, the node specified by the default path is assumed (the default path always has an explicit <node> component).
- If either name or related (but not both) contain an explicit <node> component, this is taken to be the node which performs the p_fparse.
- If both name and related contain explicit <node> components, the <node> specified in name is taken to be the node which performs the p_fparse.

If an explicit <node> in related differs from that determined from the above rules, related is assumed not to be valid for the specified node and is ignored by p_fparse.

If the <node> in the default path differs from that determined from the above rules, the default path is assumed not to be valid for the specified node and is ignored by p_fparse.

Thus, assuming that the default path is "LOC::A:\", then

```
p_fparse("PLIB.MAK","REM::HD40:SOURCE:",buf,NULL);
```

is handled by the remote filing system and writes the string REM::HD40:SOURCE:PLIB.MAK to buf, but

```
p_fparse("LOC::PLIB.MAK","REM::HD40:SOURCE:PLIB.MAKE",buf,NULL);
```

is handled by the local filing system and writes the string LOC::A:\PLIB.MAK to buf (related is ignored).

All sources that are not ignored by p_fparse are checked as being valid for the specified node, even if they do not contribute to final full file specification. Thus

```
p_fparse("LOC::C:\\PLIB.MAK","PLIB.MAKE",buf,NULL);
```

will fail with error E_FILE_NAME because the <ext> in related is not valid for the LOC:: filing system, even though all the components of a full file specification are present in name. This type of situation is typically likely to occur when copying files between the REM:: and LOC:: filing systems.

p_chdir — Change the directory in a file specification

```
INT p_chdir(TEXT *src, TEXT *outp, INT mode, TEXT *subdir);
INT p_chdirasync(TEXT *src, TEXT *outp, INT mode, TEXT *subdir, WORD *stat);
```

Parse src with a NULL related name and change its directory to produce a new file specification outp according to mode, which should be one of:

P_CD_ROOT	to move to the root directory
P_CD_PARENT	to move to the parent directory
P_CD_SUBDIR	to move to the zero terminated subdirectory subdir

The parameter subdir is ignored if mode is not P_CD_SUBDIR.

If src contains a file name then this name is retained and appended to the new directory specification outp.

There should be at least P_FNAMESIZE (128) bytes of memory reserved at outp. Note that P_FNAMESIZE bytes are *always* written to outp even when the generated file specification is shorter.

The parameters src and outp may point to the same address.

Note that p_chdir is just a means of setting up outp from mode, src and subdir with no consideration to the existence of the directories in src and subdir. The operation is performed by the appropriate file system (as specified by the <node> component of the full file specification) allowing a file specification to be manipulated without knowledge of the detailed structure (eg the delimiters used) of file specifications on that <node>.

When mode is P_CD_SUBDIR, subdir is just inserted into the full file specification at the appropriate position. Except for checking that the resultant full file specification length does not exceed P_FNAMESIZE bytes, the validity of outp is not checked.

The function returns zero if successful or a negative error number if it fails. As well as the errors that can be returned by the internal parse of src, p_chdir can return:

E_FILE_NXIST	mode was P_CD_PARENT and src was already at the root directory
E_FILE_DIR	subdir contains an invalid directory name
E_FILE_NAME	inserting subdir would make the file specification length exceed P_FNAMESIZE bytes

For example, if the current default path is "LOC::A:\", then

p_chdir("\\fred*.c",buf,P_CD_SUBDIR,"bill") writes LOC::A:\FRED\bill*.C to buf

p_chdir("\\fred\\aa.c",buf,P_CD_SUBDIR,"jim") writes LOC::A:\FRED\jim\AA.C to buf

p_chdir("\\fred\\aa.c",buf,P_CD_SUBDIR,"bill\\jim") writes LOC::A:\FRED\bill\jim\AA.C to buf
(although, depending on where bill\jim came from, this breaks the spirit of p_chdir by including a delimiter in subdir)

p_chdir("\\fred\\jim*",buf,P_CD_PARENT,NULL) writes LOC::A:\FRED* to buf

p_chdir("\\fred\\jim*",buf,P_CD_ROOT,NULL) writes LOC::A:* to buf

p_chdir("rem::hd40:fred:aa.c",buf,P_CD_SUBDIR,"jim") writes REM::HD40:FRED:jim:AA.C to buf

The default node, device and directory

p_setdefaultpath — Set the system-wide default path

```
INT p_setdefaultpath(TEXT *name);
```

Set the file server system-wide default path to name. This is the default path that is assigned to a process when it connects to the file server.

The parameter name is a zero terminated string of the form:

```
<node><device><dir>
```

where

<node>	is the file system node (eg LOC::)
<device>	is the device name (eg B:)
<dir>	is the directory name (eg \NOTES\OLD\)

The parameter name is parsed with a NULL related file name and any file name and extension component is discarded.

The function returns zero if successful or the same error numbers as for p_fparse if name failed to parse.

A successful call implies that file system <node> exists at the time of the call (note that some file systems such as REM:: are not permanently installed) and that <device> and <dir> are valid for the file system. It does not mean that <device> contains a medium, or that <dir> exists.

For example, after:

```
p_setdefaultpath("LOC::B:\\");
```

all new file server clients will initially have the default path of LOC::B:\.

p_setpth — Set the default path of this process

```
INT p_setpth(TEXT *name);
INT p_setpthasync(TEXT *name, WORD *stat);
```

Set the default node, device and directory for this process to name where name is a zero terminated string of the form:

```
<node><device><dir>
```

where

<node>	is the file system node (eg LOC::)
<device>	is the device name (eg B:)
<dir>	is the directory name (eg \NOTES\OLD\)

The parameter name is parsed with a NULL related file name and any file name and extension component is discarded. The specified directory must exist.

The function returns zero if successful or a negative error from p_fparse if name failed to parse or:

E_FILE_DEVICE	if the device does not exist
E_FILE_NOTREADY	if the device does not contain a medium
E_FILE_DIR	if the directory does not exist

When a process connects to the file server, it is assigned the file server system-wide default path, set by the last call to p_setdefaultpath.

For example, if the current default path is LOC::M:\, then:

```
p_setpth("B:");
```

sets the default path to LOC::B:\.

p_getpth — Get the default path of this process

```
VOID p_getpth(TEXT *name);
```

Write the default node, device and directory for this process as a zero terminated string to name of the form:

```
<node><device><dir>
```

where

<node>	is the file system node (eg LOC::)
<device>	is the device name (eg B:)
<dir>	is the directory name (eg \NOTES\OLD\)

There should be P_FNAMESIZE bytes of memory reserved at name.

There is no asynchronous version of p_getpth because the process default node, device and directory is stored by the file server without recourse to the appropriate file system (although the interpretation of <device> and <dir> may depend on <node>).

p_getpthbyid — Get the default path by process id

```
INT p_getpthbyid(HANDLE pid, TEXT *name);
```

Write the default node, device and directory of process pid as a zero terminated string to name. The content of name is as for p_getpth, described above.

Returns zero if successful or E_FILE_NXIST if pid is not a client of the file server.

Operations on nodes and devices

p_open(P_FNODE) — Get a list of nodes

```
INT p_open(VOID **ppfcb, TEXT *name, UINT mode);
INT p_iow(VOID *pfcb, INT func, TEXT *buf, P_NINFO *pinfo);
INT p_close(VOID *pfcb);
```

To get a list of file system node names, you:

- call p_open with a mode of P_FNODE to open a node list channel
- repeatedly call p_iow with a func of P_FREAD to read each node name (until it returns E_FILE_EOF)
- call p_close to close the node list channel

The name parameter to p_open may be "FIL:" or NULL. The mode parameter must be P_FNODE.

Each successful call to p_iow(P_FREAD) writes the next file system node name as a zero terminated string to buf (buf should have a capacity of P_FSYSNAMESIZE+1 (6) bytes). The call to p_iow(P_FREAD) returns E_FILE_EOF after all the node names have been read.

If the parameter pinfo is not NULL it is taken as the address of a P_NINFO struct which is filled with the same node information as would be obtained by calling p_ninfo, described below.

For example:

```
LOCAL_C VOID ListNodes(VOID)
    {
    VOID *ncb;
    TEXT buf[P_FSYSNAMESIZE+1];

    p_open(&ncb,"FIL:",P_FNODE);
    while (!p_iow(ncb,P_FREAD,&buf[0],NULL))
        p_puts(&buf[0]);
    p_close(ncb);
    }
```

lists the current node names.

p_ninfo — Get node information

```
INT p_ninfo(TEXT *node, P_NINFO *pninfo);
INT p_ninfoasync(TEXT *node, P_NINFO *pninfo, WORD *stat);
```

Write information on the zero terminated file system node to the P_NINFO structure at pninfo and return zero if successful or the negative E_GEN_FSYS if node is invalid or does not exist.

The P_NINFO struct is defined in *p_file.h* as:

```
typedef struct {
    UWORD version;
    UWORD type;
    UWORD formattable;
    UBYTE spare[26];
    } P_NINFO;
```

where pninfo->type contains:

P_FSYSTYPE_FLAT	if file system node does not support hierarchical directories
P_FSYSTYPE_HIER	if file system node does support hierarchical directories

and pninfo->formattable is TRUE if file system node supports the formatting of its devices and FALSE otherwise.

The version field pninfo->version is designed to allow future versions of p_ninfo (which would write further information to pninfo->spare[]) to be identified by the caller. At the time of writing, pninfo->version is set to 2.

p_locchg — Check if LOC:: has changed

```
INT p_locchg(INT mask);
```

This function is only available in EPOC version 2.16 or later.

Check if LOC:: has changed (for example, if the SSD door has been opened) since the last call to p_locchg. The return value has the bits that were set in mask either set or cleared, depending on whether or not LOC:: has changed.

The use of a mask allows multiple independent calls from within one application, with each caller consistently using one specific bit of mask. Bit 15 is ignored, so as never to return a negative value (even though the call can never fail). Bits 8 to 14 inclusive are reserved for system use. An application may therefore make up to eight independent calls, using bits 0 to 7.

Typical calling code would make an initial call to p_locchg, to ensure that only subsequent changes are detected. It would then poll for changes by calling p_locchg, say, every two seconds.

p_open(P_FDEVICE) — Get a list of devices

```
INT p_open(VOID **ppfcb, TEXT *name, UINT mode);
INT p_iow(VOID *pfcb, VOID *buf, NULL);
INT p_close(VOID *pfcb);
```

To get a list of device names for a particular file system you:

- call p_open with a mode of P_FDEVICE to open a device list channel
- repeatedly call p_iow with a func of P_FREAD to read each device name (until it returns E_FILE_EOF)
- call p_close to close the device list channel

The name parameter to p_open should be a zero terminated node name (with or without a leading FIL:). The parameter mode must be P_FDEVICE. If the node name is illegal or does not exists, p_open fails and returns E_GEN_FSYS. If the node does not support devices (as is the case for ROM::), p_open fails and returns E_GEN_NSUP.

Each successful call to p_iow(P_FREAD) writes the next device name as a zero terminated string to buf (buf should have a capacity of P_FNAMESIZE (128) bytes). The call to p_iow(P_FREAD) returns E_FILE_EOF after all

the device names have been read. The parameter following buf in the call to p_iow(P_FREAD) should be NULL.

For example:

```
LOCAL_C VOID ListDevices(VOID)
    {
    VOID *ncb,*dcb;
    INT ret;
    TEXT node[P_FSYSNAMESIZE+1];
    TEXT device[P_FNAMESIZE];

    p_open(&ncb,"FIL:",P_FNODE);
    while (!p_iow(ncb,P_FREAD,&node[0],NULL))
        {
        p_puts(&node[0]);
        if ((ret=p_open(&dcb,&node[0],P_FDEVICE))<0)
            {
            p_puts("\tDevices not supported");
            continue;
            }
        while (!p_iow(dcb,P_FREAD,&device[0],NULL))
            p_printf("\t%s",&device[0]);
        p_close(dcb);
        }
    p_close(ncb);
    }
```

lists the current node names with their devices.

p_dinfo — Get device information

```
INT p_dinfo(TEXT *dname, P_DINFO *pdinfo);
INT p_dinfoasync(TEXT *dname, P_DINFO *pdinfo, WORD *stat);
```

Write information on the device with zero terminated name dname and on the medium (eg SSD or diskette) that is mounted on device dname to the P_DINFO struct at pdinfo. Returns zero if successful or one of the following negative error numbers:

E_FILE_DEVICE	if an invalid or non-existent device is specified
E_FILE_NOTREADY	if the device does not contain a medium
E_FILE_CORRUPT	the LOC:: sub file system on a SIBO machine has recognised a RAM or Flash SSD without a valid boot record (probably because the SSD is not formatted)
E_FILE_UNKNOWN	none of the file systems recognise this medium and formatting is unlikely to make it readable

The P_DINFO struct is defined in *p_file.h* as:

```
typedef struct {
    UWORD version;
    UWORD mediatype;
    UWORD removable;
    ULONG size;
    ULONG free;
    UBYTE name[P_VOLUMENAME];
    WORD batterystate;
    UBYTE spare[16];
    } P_DINFO;
```

Except for pdinfo->removable, which is TRUE if the device has removable media, all information describes the medium mounted on device dname.

The least significant byte of pdinfo->mediatype takes one of the following values:

P_FMEDIA_UNKNOWN	unknown media type
P_FMEDIA_FLOPPY	drive takes 3.5 inch or 5.25 inch diskettes
P_FMEDIA_HARDDISK	drive contains a hard disk

P_FMEDIA_RAM	drive contains a RAM disk (read/write)
P_FMEDIA_FLASH	drive contains a Flash disk
P_FMEDIA_ROM	drive contains a ROM disk (read only)
P_FMEDIA_WRITEPROTECTED	drive contains a write protected medium

The most significant byte of pdinfo->mediatype takes a combination of the following bit flags:

P_FMEDIA_COMPRESSIBLE	it is worth compressing out logically deleted records from a file on this medium to reduce both the file size and the storage consumed by the file (not set for Flash SSDs)
P_FMEDIA_DYNAMIC	the media capacity pdinfo->size can change over time
P_FMEDIA_INTERNAL	media is internal (implicitly not removable)
P_FMEDIA_DUAL_DENSITY	device drive is dual density (if this is set then p_open(P_FFORMAT) can take the optional P_FLOWDENSITY for a low density format)
P_FMEDIA_FORMATTABLE	media is formattable

The members pdinfo->size and pdinfo->free give the total capacity of the medium in bytes and how much of that capacity is free (also in bytes) respectively. If P_FMEDIA_DYNAMIC is set in pdinfo->mediatype (as it is for the internal RAM device M:), pdinfo->size and pdinfo->free may change over time.

The volume name, with a maximum of 12 characters (for example, "DISKNAME.DSK"), is written as a zero terminated string to &pdinfo->name[0].

The system is designed to be able to take advantage of hardware which can detect a low battery voltage in a RAM SSD. When this is not possible (either because the medium is not a RAM SSD or because the hardware does not support it), pdinfo->batterystate contains E_GEN_NSUP. If the SSD does contain a battery and the hardware to detect a low voltage, pdinfo->batterystate contains FALSE if the battery is low or TRUE (more precisely, a non-zero value other than E_GEN_NSUP) if the battery voltage is acceptable. The content of this field is undefined for values of pdinfo->version less than 3.

The version field pdinfo->version is designed to allow future versions of p_dinfo (which would write further information to pdinfo->spare[] to be identified by the caller.

In the following example, ListDevices produces a device list with some device information, including the devices on REM:: (if the file server is connected to a remote file server) as well as LOC::.

```
LOCAL_C TEXT *GetTypeText(UINT type)
    {
    switch (type)
        {
        case P_FMEDIA_FLOPPY:
            return "Floppy";
        case P_FMEDIA_HARDDISK:
            return "Hard";
        case P_FMEDIA_FLASH:
            return "Flash";
        case P_FMEDIA_RAM:
            return "RAM";
        case P_FMEDIA_ROM:
            return "ROM";
        case P_FMEDIA_WRITEPROTECTED:
            return "Protected";
        }
    return "Unknown";
    }
```

```
LOCAL_C VOID ListDevices(VOID)
    {
    VOID *ncb,*dcb;
    INT ret;
    TEXT device[P_FNAMESIZE];
    TEXT bb[E_MAX_ERROR_TEXT_SIZE];
    P_DINFO dinfo;

    p_printf("   Device       Name        Type       Size      Free");
    p_printf("===========  ==========  ==========  ========  ========");
    p_open(&ncb,"FIL:",P_FNODE);
    while (!p_iow(ncb,P_FREAD,&device[0],NULL))
        {
        if (p_open(&dcb,&device[0],P_FDEVICE))
            continue;
        while (!p_iow(dcb,P_FREAD,&device[P_FSYSNAMESIZE],NULL))
            {
            if ((ret=p_dinfo(&device[0],&dinfo))<0)
                {
                p_errs(&bb[0],ret);
                p_printf("%- 12s %- 12s<%s>",&device[0],"**Failed**",&bb[0]);
                continue;
                }
            p_printf("%- 12s %- 12s%- 11s %7ldK  %7ldK",
                &device[0],&dinfo.name[0],GetTypeText(dinfo.mediatype&0xff),
                (dinfo.size+512)>>10,(dinfo.free+512)>>10);
            }
        p_close(dcb);
        }
    p_close(ncb);
    }
```

p_open(P_FFORMAT) — Format a device

```
INT p_open(VOID **ppfcb, TEXT *name, UINT mode);
INT p_read(VOID *pfcb, VOID *buf, UINT len);
INT p_close(VOID *pfcb);
```

To format a medium in a device you:

- call p_open with a mode of P_FFORMAT to open a device format channel
- call p_read to get the total format count (optional)
- repeatedly call p_read until it returns E_FILE_EOF
- call p_close to close the device format channel

The name parameter to p_open should be a zero terminated file specification with a root directory (name is parsed with a NULL related name) of the form:

```
LOC::<device>\<name><ext>
```

to format the medium in <device>, giving it the volume name <name><ext>. The volume name may subsequently be changed using p_sfstat, described later in this chapter.

The mode parameter to p_open must be P_FFORMAT. If the device supports dual density formatting (as may be established by calling p_dinfo), P_FLOWDENSITY may be OR'ed into mode to format at the lower density.

The call to p_open returns zero if successful. As well as the error numbers that can be returned by p_fparse, p_open(P_FFORMAT) can also return one of the following negative error numbers:

E_GEN_NSUP	the file system, device or medium does not support formatting
E_FILE_PROTECT	the medium is write protected or is read-only (eg a ROM SSD)

At the time of writing, formatting is only supported on the LOC:: system on a SIBO machine. You can tell in advance if a file system supports formatting by calling p_ninfo (described above). If the file system does support formatting, you can tell if a device contains a formattable medium by calling p_dinfo (also described above).

The first call to p_read writes a UWORD total count to buf and returns zero. This count represents the number of subsequent p_read calls required to complete the format. The len parameter to p_read is ignored in all cases.

A call count can be combined with the total format count to present a percentage done indication. When the format is complete, p_read returns E_FILE_EOF; you should then call p_close. Abandoning the format, by calling p_close prematurely, will leave the medium in a corrupted state.

For example, the following function:

```
LOCAL_C VOID FormatDevice(TEXT *name)
    {
    INT err,i;
    UWORD count;
    VOID *chan;
    TEXT bb[E_MAX_ERROR_TEXT_SIZE];

    chan=NULL;
    if ((err=p_open(&chan,name,P_FFORMAT))<0)
        goto exit;
    if ((err=p_read(chan,&count,0))<0)
        goto exit;
    p_printf("Formatting %s count=%d",name,count);
    i=1;
    while ((err=p_read(chan,&val,0))>=0)
        p_print("\r%05u",i++);
    if (err==E_FILE_EOF)
        err=0;
    exit:
    p_close(chan);
    if (err<0)
        {
        p_errs(&bb[0],err);
        p_printf("\r\nFormat failed: %s",&bb[0]);
        }
    else
        p_printf("\r\nFormat complete");
    }
```

could be called with:

```
FormatDevice("LOC::A:\\BACKUP");
```

to format the SSD in drive A:, giving it the volume name BACKUP.

p_locdevice — Read media information of a local device

```
INT p_locdevice(INT aDevice, UWORD *pMedia);
```

This function is only available in EPOC version 3.18 or later.

Write, to *pMedia, the media type of the **local** device (that is, a device on the LOC:: file system) specified by aDevice. The value of aDevice must be one of:

- 'M' (0x4d)
- 'I' (0x49)
- 'A' (0x41) to 'H' (0x48) inclusive.

where 'I' (Internal) is an alias for 'M'.

Apart from two additional flags, the value written to *pMedia is the same as the value written to the mediatype of a P_DINFO struct by p_dinfo. The call is more efficient than a call to p_dinfo, provided that the media type is the only information that is required.

The two extra flags that may be written to *pMedia are E_FMEDIA_BATTERY_VALID and E_FMEDIA_BATTERY_GOOD, defined in *epoc.h*. If E_FMEDIA_BATTERY_VALID is not set then the device does not support battery measurement. If it is set then E_FMEDIA_BATTERY_GOOD will be clear if the battery voltage is too low, otherwise it will be set.

Returns zero or a negative error number. Errors that may be returned are:

E_FILE_NOTREADY	no device is available
E_FILE_DEVICE	the device specified by aDevice is not in the valid range of devices
E_GEN_NSUP E_GEN_UNKNOWN	no PDD exists which can handle the device

Note that the media does not have to be mountable for this service to work.

p_locreadpdd — Direct read of local SSD

```
INT p_locreadpdd(INT aDevice, LONG *aPos, VOID *aPtr, UINT aLen);
```

This function is only available in EPOC version 3.18 or later.

Read, to the buffer pointed to by aPtr, aLen bytes starting at an offset of *aPos bytes into the SSD from the **local** SSD (that is, an SSD on the LOC:: file system) specified by aDevice. The data is read by means of direct access to the physical device driver (PDD) and the reading is therefore very efficient.

The value of aDevice must be one of:

- 'M' (0x4d)
- 'I' (0x49)
- 'A' (0x41) to 'H' (0x48) inclusive.

where 'I' (Internal) is an alias for 'M'.

The medium must have been mounted prior to using this service. (To ensure the medium is mounted, just make any normal device access.)

The service returns zero or one of the following negative error numbers:

E_GEN_OS	the medium is not mounted
E_FILE_CORRUPT	the specified offset is greater than the size of the SSD

Operations on directories and files

p_open(P_FDIR) — Get a directory list

```
INT p_open(VOID **ppfcb, TEXT *name, UINT mode);
INT p_iow(VOID *pfcb, INT func, TEXT *buf, P_INFO *pinfo);
INT p_close(VOID *pfcb);
```

To get a list of files in a directory you:

- call p_open with a mode of P_FDIR to open a directory list channel
- repeatedly call p_iow with a func of P_FREAD to read each directory entry (until it returns E_FILE_EOF)
- call p_close to close the directory list channel

The name parameter to p_open is a zero terminated file specification. Internally to the call to p_open, this is parsed with a wild card related name (such as "*.*") that will find all files in the directory. It is therefore not necessary to include a file name or file name extension in name unless you wish to restrict the directory search to files with that name and/or extension. If present, the file name and the file name extension in name would normally contain wildcards. Passing a name of "" produces the names of all the files in the current directory. The parameter mode must be P_FDIR.

Each successful call to p_iow(P_FREAD) writes the next matching file name (excluding the node, device and directory component) as a zero terminated string to buf (buf should have a capacity of P_FNAMESIZE (128) bytes). The call to p_iow(P_FREAD) returns E_FILE_EOF after all the file names have been read.

If the parameter pinfo is not NULL it is taken as the address of a P_INFO struct where P_INFO is defined in *p_file.h* as:

```
typedef struct {
    UWORD version;
    UWORD status;   /* status bits */
    ULONG size;     /* size of the file in bytes */
    ULONG modst;    /* system time of last modification */
    UBYTE spare[4];
    } P_INFO;
```

The file status pinfo->status has the following bit fields:

P_FAWRITE	set if file is not read-only
P_FAMOD	set if the file has been modified
P_FAHIDDEN	set if file is hidden
P_FASYSTEM	set if file is a system file
P_FADIR	set if the file is a directory file
P_FAVOLUME	set if the file is a volume name directory
P_FATEXT	set if file is a text file

Note that a directory read may return with a volume name written to buf and P_FAVOLUME set in pinfo->status. This will only occur if the directory is a root directory of a PC-based device that has a volume name.

A directory read only returns with P_FATEXT set in pinfo->status when the file system can recognise a text file. The LOC:: file system cannot recognise text files but a remote file server, running on an operating system (eg VMS) which can recognise text files, will set the P_FATEXT bit as appropriate.

The field pinfo->size gives the logical file length (the end-of-file position).

The field pinfo->modst gives the time the file was last modified, expressed in system time format (the number of seconds since 00:00:00 January 1 1970). See the chapter *Time, Timers and Dates* for details on how to convert to and from the system time.

The P_INFO file information may also be obtained when using p_finfo, described below.

Example

```
#include <plib.h>

LOCAL_D VOID *dcb=NULL;

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

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

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

    p=&b[0];
    if (pinfo->status&P_FAVOLUME)
        p=p_scpy(p,"Vol,");
    if (pinfo->status&P_FADIR)
        p=p_scpy(p,"Dir,");
    if (pinfo->status&P_FAMOD)
        p=p_scpy(p,"Mod,");
    if (!(pinfo->status&P_FAWRITE))
        p=p_scpy(p,"Read,");
    if (pinfo->status&P_FASYSTEM)
        p=p_scpy(p,"Sys,");
    if (pinfo->status&P_FAHIDDEN)
        p=p_scpy(p,"Hid,");
    if (*(p-1)==',')
        *--p=0;
    p_sttods(&pinfo->modst,&ds);
    p_dstodt(&ds,&dt);
    p_printf("%- 12s %8lu  %02u-%02u-%02u %02u:%02u  %s",
        name,pinfo->size,dt.day+1,dt.month+1,dt.year,dt.hour,dt.minute,&b[0]);
    }

LOCAL_C VOID CDECL PrintDirList(TEXT *dir)
    {
    INT err,NoFiles;
    P_INFO info;
    TEXT name[P_FNAMESIZE];

    if ((err=p_open(&dcb,dir,P_FDIR))!=0)
        panic("Failed to open directory file",err);
    NoFiles=TRUE;
    while (!(err=p_iow(dcb,P_FREAD,&name[0],&info)))
        {
        NoFiles=FALSE;
        PrintDirLine(&name[0],&info);
        }
    p_close(dcb); dcb=NULL;
    if (err!=E_FILE_EOF)
        panic("Failed to read directory",ret);
    if (NoFiles)
        p_printf("No files found");
    }

GLDEF_C INT main(VOID)
    {
    TEXT name[P_FNAMESIZE];

    while (p_getl(">",&name[0],P_FNAMESIZE))
        p_enter((VOID *)PrintDirList,&name[0]);
    return(0);
    }
```

p_finfo — Return file information

```
INT p_finfo(TEXT *name, P_INFO *pinfo);
INT p_finfoasync(TEXT *name, P_INFO *pinfo, WORD *stat);
```

Parse `name` with a `NULL` related name and write information about the specified file (which may be a directory file) to the `P_INFO` struct at `pinfo`. The same file information is written to `pinfo` as is obtained when using `p_iow(P_FREAD)` on a directory list channel that was opened with `p_open(P_FDIR)`, as described above.

The version field pinfo->version is designed to allow future versions of p_finfo (which would write further information to pinfo->spare[]) to be identified by the caller. At the time of writing, pinfo->version is set to 2.

Returns zero if successful or a negative error number. As well as the p_fparse error numbers if the parse fails, p_finfo can return:

E_FILE_DEVICE	if the device does not exist
E_FILE_NOTREADY	if the device does not contain a medium
E_FILE_DIR	if the directory does not exist
E_FILE_NXIST	if the file does not exist

The atomic nature of the p_finfo (unlike p_open) makes it a good choice for checking whether a file exists (it returns E_FILE_NXIST if the file does not exist). To test for the existence of a directory, you can also use p_testpth, described below.

p_testpth — Test for the existence of a directory

```
INT p_testpth(TEXT *dname);
INT p_testpthasync(TEXT *dname, WORD *stat);
```

Return zero if the directory component of the zero terminated file specification dname exists.

The file specification dname is parsed with a NULL related name and any file name component is discarded.

If the parse fails, p_testpth returns the error return from p_fparse. It can also fail with:

E_FILE_DEVICE	if the device does not exist
E_FILE_NOTREADY	if the device does not contain a medium
E_FILE_DIR	if the directory does not exist

For example, if the current path is LOC::A:\, then:

```
p_testpth("\\dir1\\dir2\\fred.c");
```

returns zero if the directory LOC::A:\DIR1\DIR2\ exists.

p_rename — Rename a file or directory

```
INT p_rename(TEXT *oldname, TEXT *newname);
INT p_renameasync(TEXT *oldname, TEXT *newname, WORD *stat);
```

Parse each of newname and oldname with a NULL related name and, if successful, rename oldname to newname. Both buffers should be at least P_FNAMESIZE bytes in length.

The file specified by oldname must exist and newname must not already exist.

Neither file name may include wild cards.

Both files should be on the same node and device. On LOC::, files may be renamed across directories (but this may not be supported by some remote systems). Prior to version 3.5 of EPOC, renaming across directories can fail when the target directory is the root of LOC::M:.

Directory files (which do not need to be empty) may be renamed, but not across different directories.

The function returns zero if successful or a negative error number if it fails. As well as a p_fparse error number if either oldname or newname fails to parse, p_rename can return the following error numbers:

E_FILE_DEVICE	oldname and newname are on different devices or the device does not exist
E_FILE_NOTREADY	the device does not contain a medium
E_FILE_DIR	the directory does not exist
E_FILE_EXIST	newname already exists
E_FILE_NXIST	oldname does not exist
E_FILE_LOCKED	oldname exists but a process has the file open

E_FILE_PROTECT	the medium is write protected or is read-only (eg a ROM SSD)
E_FILE_FULL	not enough room on the medium for the rename (possible on Flash SSDs)

For example, if the current path is LOC::A:\, then:

```
p_rename("\\dir1\\dir2\\fred.c","\\dir1\\dir2\\jim.c");
```

renames FRED.C in LOC::A:\DIR1\DIR2\ to JIM.C while:

```
p_rename("\\dir1\\dir2\\fred.c","fred.c");
```

effectively moves FRED.C in LOC::A:\DIR1\DIR2\ to the root directory.

It is sometimes useful to parse newname with oldname as a related name before calling p_rename as in:

```
GLDEF_C INT RenameFile(TEXT *oldname, TEXT *newname)
    {
    TEXT buf[P_FNAMESIZE];

    if (ret=p_fparse(newname,oldname,&buf[0],NULL))
        return(ret);
    return(p_rename(oldname,&buf[0]));
    }
```

Then, for example, calling:

```
RenameFile("\\dir1\\dir2\\fred.c","jim.c");
```

renames FRED.C in LOC::A:\DIR1\DIR2\ to JIM.C while:

```
RenameFile("\\dir1\\dir2\\fred.c","\\jim.c");
```

renames FRED.C in LOC::A:\DIR1\DIR2\ to JIM.C and moves JIM.C to the root directory.

p_delete — Delete a file or directory

```
INT p_delete(TEXT *name);
INT p_deleteasync(TEXT *name, WORD *stat);
```

Parse name with a NULL related name and, if successful, delete the specified file (which may be a directory file).

A directory can not be deleted unless it is empty.

The function returns zero if successful or a negative error number if it fails. As well as a p_fparse error number if name fails to parse, p_delete can return the following negative error numbers:

E_FILE_DEVICE	the device does not exist
E_FILE_NOTREADY	the device does not contain a medium
E_FILE_DIR	the directory does not exist
E_FILE_NXIST	the file does not exist
E_FILE_LOCKED	a process has name open
E_FILE_ACCESS	name is a read-only file
E_FILE_PROTECT	the medium is write protected or is read-only (eg a ROM SSD)
E_FILE_EXIST	name is a directory that contains files

For example, if the current path is LOC::A:\, then:

```
p_delete("\\dir1\\dir2\\fred.c");
```

deletes LOC::A:\DIR1\DIR2\FRED.C.

p_mkdir — Make a new directory

```
INT p_mkdir(TEXT *name);
INT p_mkdirasync(TEXT *name, WORD *stat);
```

Parse name with a NULL related name and create the specified directory. The name of the directory to be created is specified by the file name component of name if name contains a file name component. Otherwise it is specified by the directory component of the parsed file specification.

Intermediate directories are also created if necessary.

Although unusual, there is no reason why a directory should not have an extension, and any extension in name is significant.

The function returns zero if successful or a negative error number if it fails. As well as the p_fparse error numbers returned if name fails to parse, p_mkdir can return the following negative error numbers:

E_FILE_DEVICE	the device does not exist
E_FILE_NOTREADY	the device does not contain a medium
E_FILE_EXIST	the directory already exists
E_FILE_PROTECT	the medium is write protected or is read-only (eg a ROM SSD)
E_FILE_FULL	there is no more room on the medium
E_FILE_DIRFULL	there is no more room in the root directory

For example, if the current path is LOC::A:\, then either:

```
p_mkdir("\\dir1\\dir2\\");
```

or

```
p_mkdir("\\dir1\\dir2");
```

makes the directory LOC::A:\DIR1\DIR2\ and also makes LOC::A:\DIR1\ if it does not already exist.

p_sfstat — Set file attributes or label medium

```
INT p_sfstat(TEXT *name, UINT status, UINT mask);
INT p_sfstatasync(TEXT *name, UINT status, UINT mask, WORD *stat);
```

Set or clear the attributes of the file specified by the zero terminated file name name (but see also *Setting the volume name* below). Both status and mask contain a bit field made up of the following bit masks:

P_FAWRITE	set if file may be written to
P_FAMOD	set if the file has been modified
P_FAHIDDEN	set if file is hidden
P_FASYSTEM	set if file is a system file

The function only modifies those bits that are set in mask where attribute is set or cleared depending on the value of the corresponding bit in status.

The function returns zero if successful or a negative error number if it fails. As well as a p_fparse error number if name fails to parse, p_sfstat can return the following negative error numbers:

E_FILE_DEVICE	the device does not exist
E_FILE_NOTREADY	the device does not contain a medium
E_FILE_DIR	the directory does not exist
E_FILE_NXIST	the file does not exist
E_FILE_LOCKED	a process has name open
E_FILE_PROTECT	the medium is write protected or is read-only (eg a ROM SSD)

For example:

```
p_sfstat("joe.doc",0,P_FAWRITE);
```

makes the file joe.doc read-only.

Setting the volume label

If mask has the P_FAVOLUME bit set, status is ignored p_sfstat and p_sfstat sets or deletes the volume name.

At the time of writing, only the LOC:: file system supports setting the volume label (the function returns E_GEN_NSUP if it is not supported).

The name parameter should be a zero terminated file specification with a root directory (name is parsed with a NULL related name) of the form:

```
LOC::<device>\<name><ext>
```

to label the medium in <device>, giving it the volume name <name><ext>. If name does not contain a <name><ext> component, the volume label is deleted.

The function returns zero if successful or a negative error number if it fails. As well as a p_fparse error number if name fails to parse, p_sfstat can return the following negative error numbers:

E_FILE_DEVICE	the device does not exist
E_FILE_NOTREADY	the device does not contain a medium
E_FILE_PROTECT	the medium is write protected or is read-only (eg a ROM SSD)

For example, if the current path is LOC::A:\, then:

```
p_sfstat("d:\mydisk",0,P_FAVOLUME);
```

gives the medium in device LOC::D: the label MYDISK,

```
p_sfstat("d:\diskname.dsk",0,P_FAVOLUME);
```

gives the medium in device LOC::D: the label DISKNAME.DSK and:

```
p_sfstat("d:\",0,P_FAVOLUME);
```

deletes any volume name from device LOC::D:.

p_fdate — Set file creation date

```
INT p_fdate(TEXT *name, ULONG date);
INT p_fdateasync(TEXT *name, ULONG date, WORD *stat);
```

Set the file creation date to date where date is in the form of the system time - that is, the number of seconds since January 1st 1970.

The date may not be set earlier than January 1st 1980. If p_fdate is called with any earlier system time, the date is forced (without error) to January 1st 1980.

The system time used for the creation date is always converted to an even number of seconds (i.e. the least significant bit of date is discarded).

The function returns zero if successful or a negative error number if it fails. As well as a p_fparse error number if name fails to parse, p_fdate can return the following negative error numbers:

E_FILE_DEVICE	the device does not exist
E_FILE_NOTREADY	the device does not contain a medium
E_FILE_DIR	the directory does not exist
E_FILE_NXIST	the file does not exist
E_FILE_LOCKED	a process has name open
E_FILE_ACCESS	name is a read-only file
E_FILE_PROTECT	the medium is write protected or is read-only (eg a ROM SSD)

The following example copies the date of fred.doc to fred.txt.

```
P_INFO info;

p_finfo("fred.doc",&info);
p_fdate("fred.txt",info.modst);
```

Binary file access

To open a file to be manipulated as a flat binary file, you call p_open with the 3rd parameter mode OR'ed with P_FSTREAM. For example:

p_open(&fcb,"fred.dat",P_FSTREAM|P_FSHARE) opens a file for reading only.

p_open(&fcb,"fred.dat",P_FSTREAM|P_FUPDATE|P_FREPLACE|P_FRANDOM) creates a writable binary file.

The mode flags (eg P_FREPLACE) are described under p_open, below.

There is no practical limit to the number of channels that may be opened in the system or by a particular process.

Once you have opened a binary file channel, the possible I/O operations are:

p_read p_ioc(P_FREAD) p_ioa(P_FREAD)	to read from the file channel
p_write p_ioc(P_FWRITE) p_ioa(P_FWRITE)	to write to the file channel
p_close	to close the file channel
p_seek	to set and sense the channel file position
p_iow(P_FSETEOF)	to set the logical end of file
p_iow(P_FFLUSH)	to flush the file buffers
p_iow(P_FCANCEL)	to cancel all pending asynchronous requests (not actively supported)

Bytes can be read or written to the file in any length up to a maximum of P_FMAXSSIZE bytes per read or write.

Shared access

Any number of processes can open the same file for reading only (but only if all the readers specify P_FSHARE when they open the file). However, the file server does not allow multiple processes to open a channel to the same file for writing.

Once a file has been opened for reading, it may not be opened again for writing (but it may be opened again for reading). Once a file has been opened for writing, it may not be opened again for reading or for writing.

If you have an multi-process application design which needs to update shared data, consider one of the following:

- put the shared data in a named segment (see the chapter *Memory Allocation*)
- access the shared file data via a server process (see the chapter *Processes and Inter-Process Messaging*)

Example of binary file access

The following example, which compares the contents of two files, uses many of the functions described in this chapter.

```
#include <plib.h>

typedef struct
    {
    INT ret;
    VOID *chan;
    LONG len;
    TEXT name[P_FNAMESIZE];
    UBYTE buf[P_FBLKSIZE];
    } FILE_DATA;

LOCAL_D FILE_DATA f1={0,NULL};
LOCAL_D FILE_DATA f2={0,NULL};

LOCAL_C VOID CleanUp(TEXT *msg, FILE_DATA *pf)
    {
    TEXT bb[E_MAX_ERROR_TEXT_SIZE];

    p_close(pf->chan);
    pf->chan=NULL;
    if (pf->ret<0)
        {
        p_errs(&bb[0],pf->ret);
        p_printf("Failed to %s %s (%s)",msg,&pf->name[0],&bb[0]);
        pf->ret=0;
        }
    }

LOCAL_C VOID Exit(TEXT *msg)
    {
    if (f1.ret>=0 && f2.ret>=0)
        p_printf(msg);
    CleanUp(msg,&f1);
    CleanUp(msg,&f2);
    p_leave(0);
    }

LOCAL_C VOID OpenFile(FILE_DATA *pf, TEXT *name, TEXT *related)
    {
    LONG pos;

    if (pf->ret=p_fparse(name,related,&pf->name[0],NULL))
        Exit("parse");
    if (pf->ret=p_open(&pf->chan,&pf->name[0],P_FSTREAM|P_FSHARE|P_FRANDOM))
        Exit("open");
    pf->len=0L; p_seek(pf->chan,P_FEND,&pf->len);
    pos=0L; p_seek(pf->chan,P_FABS,&pos);
    }

LOCAL_C VOID ReadFile(FILE_DATA *pf)
    {
    pf->ret=p_read(pf->chan,&pf->buf[0],sizeof(pf->buf));
    if (pf->ret!=E_FILE_EOF && pf->ret<0)
        Exit("read");
    }
```

```
LOCAL_C VOID CDECL CompareFiles(TEXT *file1, TEXT *file2)
    {
    OpenFile(&f1,file1,NULL);
    OpenFile(&f2,file2,&f1.name[0]);
    p_printf("Compare %s (%ld)",&f1.name[0],f1.len);
    p_printf("   with %s (%ld)",&f2.name[0],f2.len);
    if (f1.len!=f2.len)
        Exit("Files are of different length");
    FOREVER
        {
        ReadFile(&f1);
        ReadFile(&f2);
        if (f1.ret==E_FILE_EOF && f2.ret==E_FILE_EOF)
            {
            f1.ret=f2.ret=0;
            Exit("Files are identical");
            }
        if (p_bcmp(&f1.buf[0],f1.ret,&f2.buf[0],f2.ret))
            Exit("Files are different");
        }
    }

GLDEF_C INT main(VOID)
    {
    TEXT *p;
    TEXT bb[P_FNAMESIZE];

    while (p_getl("Enter <file1> <file2> ? ",&bb[0],P_FNAMESIZE))
        {
        p=p_skipch(&bb[0]);
        if (*p)
            *p++=0;
        p_enter((VOID *)CompareFiles,&bb[0],p_skipwh(p));
        }
    return(0);
    }
```

The program solicits two file names to compare (where the second name is parsed with the first name as a related name in `CompareFiles`) and reports whether they are the same or different. As well as illustrating binary file access, the example gives a realistic illustration of the use of `p_enter` and `p_leave`. Note also that the function `CleanUp` takes advantage of the fact that `p_close(NULL)` is harmless.

p_open(P_FSTREAM) — Open a binary file

```
INT p_open(VOID **ppfcb, TEXT *name, UINT mode);
```

Open a channel to the file specified by the zero terminated file specification `name` and, if successful, return zero and write the channel to `*ppfcb` (`ppfcb` is not written to if the open fails). To open a file to be manipulated as a flat binary file, `mode` should be `OR`'ed with `P_FSTREAM`.

The file specification `name` is parsed with a `NULL` related file name (see `p_fparse`). If this fails, the open fails and returns the return value from `p_fparse`.

The mode in which the file is opened is selected by `OR`ing in one (and only one) of the following bit fields into `mode`:

`P_FOPEN`	Open an existing file. If the file does not exist, the error `E_FILE_NXIST` is returned. This option would normally only be used to open a file for read access.
`P_FCREATE`	Create a file which must not already exist. If the file does exist, the error `E_FILE_EXIST` is returned. To enable write access to the file you must specify `P_FUPDATE` (described below).
`P_FREPLACE`	If the file exists, open it and truncate it to zero length. If the file does not exist, then create a file. To enable write access to the file you must specify `P_FUPDATE` (described below).

P_FAPPEND	This is the same as for P_FOPEN except that the initial current position is set to the end of file such that the next write will append to the file. It is not necessary to specify P_FRANDOM (described below) but to enable write access to the file you must specify P_FUPDATE (described below).
P_FUNIQUE	Create a unique file using the passed path as the related path name in which to create the file. The unique file name is written back to name (there should be room for P_FNAMESIZE bytes). It is not necessary to specify P_FUPDATE (described below).

The access which is subsequently permitted is formed by ORing together a combination of the following flags in mode:

P_FUPDATE	specifies that write access as well as read access is required for the file. If an attempt is made to write to a file when this flag has not been set, the error E_FILE_RDONLY is returned.
P_FRANDOM	specifies that random access (as opposed to sequential access) is required for the file. If a call is made to p_seek with a file that has not been opened with this option, the error E_FILE_INV is returned. You should not specify P_FRANDOM unless you do intend to use p_seek because the file system may be able to optimise the device access if it knows that only sequential access is required.
P_FSHARE	specifies that this open should not block the file from being opened again for read access. If this flag is not set, a subsequent request to open a file (by the same or another process) will fail with E_FILE_LOCKED.

Note that shared write access is not supported and P_FSHARE can not be combined with P_FUPDATE.

Returns zero if successful or a negative error number if it fails. As well as the p_fparse error numbers which may be returned if name fails to parse, p_open(P_FSTREAM) can return the following negative error numbers:

E_GEN_NOMEMORY	failed to allocate memory for the control block
E_GEN_ARG	mode contains an illegal combination of flags
E_FILE_DEVICE	the device does not exist
E_FILE_NOTREADY	the device does not contain a medium
E_FILE_EXIST	attempt to create a file which already exists
E_FILE_NXIST	attempt to open a file which does not exist
E_FILE_ACCESS	access to the file in the requested mode is not available
E_FILE_DIRFULL	there is no more room in the root directory
E_FILE_PROTECT	attempted to create/replace when the medium is write protected or is read-only (eg a ROM SSD)
E_FILE_FULL	there is no room on the device to create/replace this file
E_FILE_LOCKED	the file is already open (possibly by another process)
E_FILE_DEVICE	the device in name does not exist
E_FILE_DIR	the directory in name does not exist

p_close — Close a binary file channel

```
INT p_close(VOID *pfcb);
```

Close the binary file channel pfcb and return zero if successful. If pcb is NULL, just return zero.

Although p_close can return an error, it will always succeed in closing the channel (and pfcb should not be used subsequently).

If the file system buffers written data, the close operation may need to perform one or more write operations on closing. Even if written data is not buffered, if data has been written to the file, the close will update the modification date on the file. Because of this, p_close can return the some of the errors numbers which can be returned from p_write and p_fdate. However, the failure to flush the data or to

write the new date will not cause the close operation to be aborted although the failure will be reported by an error return.

Carefully written applications avoid this problem by using p_iow(P_FFLUSH) to flush the data and to apply the date (and taking appropriate action if this fails) before closing the channel without risk of failure.

The LOC:: file system on SIBO machines does not buffer written data. When EPOC is running on a PC, the LOC:: file system is built over the MSDOS filing system which does buffer written data.

p_read — Read from a binary file channel

```
INT p_read(VOID *pfcb, VOID *buf, UINT len);
```

Reads len bytes or the number of bytes remaining before the end of file, whichever is smaller, from the current position of binary file channel pfcb and writes the data to buf. The current position is incremented by the number of bytes read.

If the current position is already at the end of file, zero bytes are read and the negative error number E_FILE_EOF is returned.

The parameter len must not be greater than P_FMAXSSIZE (16K bytes).

The most efficient way of processing files is to read in multiples of P_FBLKSIZE (512) while ensuring that the file position remains on P_FBLKSIZE boundaries.

The function returns the number of bytes written to buf if successful or one of the following negative error numbers:

E_FILE_EOF	end of file encountered
E_FILE_ABORT	the SSD containing the file is or was previously not present and the channel is now in the abort state
E_FILE_READ	failed to read from the file (eg because of a CRC failure on a floppy disk)

p_write — Write to a binary file channel

```
INT p_write(VOID *pfcb, VOID *buf, UINT len);
```

Write len bytes from buf to file channel pfcb at the current file position. The current position is incremented by the number of bytes written.

The parameter len must not be greater than P_FMAXSSIZE (16K bytes).

The most efficient way of processing files is to write in multiples of P_FBLKSIZE (512) while ensuring that the file position remains on P_FBLKSIZE boundaries.

When writing to Flash SSDs, you can physically overwrite a single byte (where len is one) provided that the new byte can be written by just clearing bits in the old byte (if this is possible, the file modification date is not changed). In general, overwriting data on a Flash SSD-based file will consume space and reduce the remaining capacity of the SSD.

The function returns zero if successful or one of the following negative error numbers:

E_FILE_FULL	not enough room on the device for the data
E_FILE_RDONLY	the file channel was opened without the P_FUPDATE access bit set
E_FILE_ABORT	the SSD containing the file is or was previously not present and the channel is now in the abort state
E_FILE_WRITE	failed to write to the file (eg because of a CRC failure on a floppy disk)

p_seek (or f_seek) — Position a binary file channel

```
INT p_seek(VOID *pfcb, INT sense, LONG *ppos);
INT f_seek(VOID *pfcb, INT sense, LONG *ppos);
```

Set the current file position of file channel pfcb to a new file position which depends on sense and *ppos, where sense is one of:

P_FABS	to set the current file position to *ppos

P_FEND	to set the current file position to *ppos relative to the current end-of-file position
P_FCUR	to set the current file position to *ppos relative to the current file position

If successful, the new file position (which may be the same as the old) is written to *ppos and p_seek returns zero. Otherwise, it returns one of the following negative error numbers:

E_FILE_INV	the channel was not opened with P_FRANDOM
E_FILE_ABORT	the SSD containing the file is or was previously not present and the channel is now in the abort state

If the channel was opened with P_FSTREAM_TEXT on a remote file system, the full P_FSEEK functionality may not be supported. When this is the case, p_seek fails with E_FILE_INV. However, the following is always supported on P_FSTREAM_TEXT channels:

- using P_FABS to set the current position to zero
- using P_FEND with a zero relative position to set the current position to the end of the file (although with P_FSTREAM_TEXT you should not use the value which is written to *ppos)

If calling p_seek results in an absolute file position which is negative then the file is positioned at the beginning of the file. If the absolute file position is greater than the end of file then the new position is set to the end of file. See p_iow(P_FSETEOF) for setting a new end of file.

The function f_seek is identical to p_seek except that, if there is an error, it calls p_leave(err) rather than return the negative error number err.

For example, if the current file position is 0x2000 then

```
LONG pos;

pos=256L;
p_seek(pfcb,P_FCUR,&pos);
```

sets the current position to 0x2100 and

```
pos=-256L;
p_seek(pfcb,P_FCUR,&pos);
```

sets the current position to 0x1f00 and

```
pos=0L;
p_seek(pfcb,P_FCUR,&pos);
```

does not change the current position but could be used to sense the current position (pos contains 0x2000 after p_seek has returned). If the current end of file is 0x4000 then

```
pos=0L;
p_seek(pfcb,P_FEND,&pos);
```

sets the current position to the end of the file and senses the length of the file (pos contains 0x4000 after p_seek has returned). To set the current position to the beginning of the file, use:

```
pos=0L;
p_seek(pfcb,P_FABS,&pos);
```

p_iow(P_FFLUSH) Flush internal file buffers

```
INT p_iow(VOID *pfcb, P_FFLUSH);
```

Flush all buffered written data to binary file channel pfcb and write the file's modification date.

Returns zero if successful (the negative error number returns are as for p_write).

The amount of information which is flushed depends on the file system and on the medium in the drive. The RAM and Flash SSD PDDs do not buffer any file data (in this case, calling p_iow(P_FFLUSH) just writes the file date). On Flash SSDs the last record written is held open until the file is closed or a write occurs to another file on the same SSD (see the section *Flash SSDs* at the beginning of this chapter). When EPOC is running on a PC, written file data is buffered to improve performance. Calling p_iow(P_FFLUSH) will ensure that any buffered data is written to the file but repeated flushing is likely to degrade performance.

Although harmless, there is absolutely no benefit in calling p_iow(P_FFLUSH) on file channels which were opened without the P_FUPDATE flag.

p_iow(P_FSETEOF) Set end of file

```
INT p_iow(VOID *pfcb, P_FSETEOF, ULONG *peof);
```

Set the logical end of file on channel pfcb to position *peof and return zero if successful (the negative error number returns are as for p_write). The the file channel must have been opened with P_FUPDATE.

If *peof is greater than the current end of file, the file size is extended to *peof. On block-structured devices (ie excluding Flash SSDs), this is equivalent to appending indeterminate data to the end of the file to pre-allocate storage. (It is not useful to extend a file on a Flash SSD.) The current position is not affected when the file is extended with p_iow(P_FSETEOF).

If *peof is less than the current end of file, the file is truncated to *peof (the current position is also reduced if necessary to the new end of file).

p_iow(P_FCANCEL) Cancel an asynchronous file channel request

```
INT p_iow(VOID *pfcb, P_FCANCEL);
```

Cancel all asynchronous I/O requests on channel pfcb and return zero.

The operation is harmless if no request is pending. In fact, for reasons which were described at the beginning of this chapter (under the heading *Asynchronous file operations*), a call to p_iow(P_FCANCEL) has no effect even when a request *is* pending.

However, if you are performing asynchronous file access (which is not at all common) you may (for the sake of consistency and provided the status word is currently E_FILE_PENDING) use p_iow(P_FCANCEL) - normally followed by a call to p_waitstat - to effect cancellation as you would do for any other asynchronous request. It is always possible that a future implementation of the file server may process P_FCANCEL operations.

At the time of writing, a preferable alternative is to simulate a cancel service, as illustrated in the earlier example, under the heading *Asynchronous file operations*.

Stream text file access

Different file systems vary in their support of a text file type and (if such support is provided) implement text files in different ways.

Some systems (for example, DEC's VMS operating system which runs on a Vax) support text file types that are not implemented as CRLF terminated records. Other systems (such as MSDOS, UNIX and the EPOC LOC:: file system on a SIBO machine or a PC) do not support a text file type, and on such systems the following convention predominates:

- text records are terminated by a CRLF sequence - a carriage return (code 13) followed by a line feed (code 10)
- a file may optionally be terminated by a SUB character (code 26)

The terminating SUB is not necessary on systems that store a logical file length and it is falling out of use.

Applications running under EPOC that read or write text files can choose either to process the records in a text file themselves, or to take advantage of the system's text file handling. The system support for text files, using p_open(P_FTEXT), is described in the next section of this chapter.

Applications that do their own text file processing, for performance purposes or otherwise, should use p_open(P_FSTREAM_TEXT) in preference to p_open(P_FSTREAM).

Using P_FSTREAM_TEXT when opening such a text file on the REM:: file system declares the intention that the file is to be considered as a text file. It causes the remote file server to present the data from the file as if it were implemented with CRLF terminated records. In more detail, this presentation layer does the following:

when reading	the text record content on the remote system is converted to that content followed by a CRLF

when writing — the data is parsed for CRLF record terminators and converted into text records on the remote system (the parser will also recognise CR, LF or LFCR as a record terminator)

Although in many cases (and certainly when opening a local file) opening with P_FSTREAM_TEXT has an identical effect to opening with P_FSTREAM, there is no guarantee that the effect will be the same on a remote file. There is no penalty to using P_FSTREAM_TEXT - only a potential gain when accessing a remote file.

The upshot of all this is...

All this sounds very complicated (and it is - especially for the implementor of the remote file server) but you should trust the system and just remember this:

> To open a text file to be manipulated as a flat binary file, you should call p_open with mode OR'ed with P_FSTREAM_TEXT and not P_FSTREAM.

With the exception of p_open, all functions are exactly as for flat binary files, described in the previous section.

p_open(P_FSTREAM_TEXT) — Open a stream text file

```
INT p_open(VOID **ppfcb, TEXT *name, UINT mode);
```

Open a channel to the file specified by the zero terminated file specification name and, if successful, return zero and write the channel to *ppfcb (ppfcb is not written to if the open fails).

To open a stream text file (a text file to be manipulated as a flat binary file) mode should be OR'ed with P_FSTREAM_TEXT.

For all other details, see the description p_open(P_FSTREAM) in the previous section.

Text file access

To open a file to be manipulated as a record oriented text file, you call p_open with the 3rd parameter mode OR'ed with P_FTEXT. For example:

p_open(&fcb,"fred.lis",P_FTEXT|P_FSHARE) opens a text file for reading only.

p_open(&fcb,"fred.lis",P_FTEXT|P_FUPDATE|P_FREPLACE) creates a text file which may be written to.

Once you have opened a text file channel, the possible I/O operations are:

p_read p_ioc(P_FREAD) p_ioa(P_FREAD)	to read the next text record from the file channel
p_write p_ioc(P_FWRITE) p_ioa(P_FWRITE)	to append a text record to the file channel
p_close	to close the text file channel
p_seek	to set and sense the channel file position
p_iow(P_FSETEOF)	to set the logical end of file
p_iow(P_FFLUSH)	to flush the file buffers
p_iow(P_FCANCEL)	to cancel all pending asynchronous requests (not actively supported)

The text file handling is not part of the file server but is implemented as a layer of code over the P_FSTREAM_TEXT binary file access. The text file handling code runs in the caller's context ("on the client side") in the same way as regular function calls. This layer of code is inserted between the application and the file server by the FIL: device driver p_open code when it sees the P_FTEXT bit set in the mode parameter. The data is, however, buffered in this layer, so flushing is necessary to ensure that any buffered data is written to the file.

Implementing the code as a layer over P_FSTREAM_TEXT binary file access makes it independent of the file system node which is providing the services.

What text file handling does

The text file handling assumes that text files obey the following conventions:

- text records are terminated by a CRLF sequence - a carriage return (code 13) followed by a line feed (code 10)
- a file may optionally be terminated by a SUB character (code 26)

The text file handling also assumes that the record content (excluding the record termination) does not exceed a length of P_FMAXRSIZE (256) bytes. A text record may not contain CR, LF or SUB characters, but there is no other restriction on the contents.

When reading a file, the text file handling parses a stream of bytes for CRLF record terminators such that the application gets the data a record at a time (excluding the terminator). The parser will also recognise CR, LF, LFCR or a SUB as a record terminator. If it sees a SUB (either as a record terminator or immediately after a CRLF, CR, LF or LFCR record terminator), it will behave as if the end of file had been reached.

When writing to a file, the text file handling simply adds the record terminator (which is always CRLF) to the end of the record data. A terminating SUB is not written.

For enthusiasts ...

This is for interest only and may certainly be skipped.

In case you were wondering what the TXT: device is, text file handling is implemented as an I/O device driver (TXT:) which layers over the FIL: P_FSTREAM_TEXT mode binary file access (which was described in the previous section). The FIL: device redirects the p_open to TXT: when P_FTEXT is set in mode. This means that the following calls to p_open:

```
p_open(&fcb,"fred.dat",P_FTEXT|P_FUPDATE|P_FREPLACE);
p_open(&fcb,"FIL:fred.dat",P_FTEXT|P_FUPDATE|P_FREPLACE);
p_open(&fcb,"TXT:fred.dat",P_FUPDATE|P_FREPLACE);
```

are all equivalent. (The above examples are provided to help de-mystify the I/O system and the FIL: device and it would be obscure to use TXT: without good cause.)

p_open(P_FTEXT) — Open a text file

```
INT p_open(VOID **ppfcb, TEXT *name, UINT mode);
```

Open a file to be manipulated as a record oriented text file where mode should be OR'ed with P_FTEXT.

Except for the use of P_FTEXT in place of P_FSTREAM or P_FSTREAM_TEXT, the parameters and returns are as for opening a binary file, described in the previous section.

p_close — Close a text file channel

```
INT p_close(VOID *pfcb);
```

Close the text file channel pfcb and return zero if successful.

Since text written to the file is buffered in the device driver, p_close may give rise to an E_FILE_WRITE error. It is therefore advisable to call p_iow(P_FFLUSH) before calling p_close. Otherwise, the behaviour and returns are as for closing a binary file, described in the previous section.

p_read — Read from a text file channel

```
INT p_read(VOID *pfcb, VOID *buf, UINT len);
```

Read the contents of the current record (ie excluding any record terminators) from text file channel pfcb, writing up to len bytes to buf and, if successful, return the length of the record read (which is also the number of bytes written to buf). The channel is positioned to the next record.

If len is less than the length of the current record, the first len bytes from the record are read and p_read returns the negative E_FILE_RECORD. The channel is still positioned to the next record.

Note that p_read returns zero if it encounters a record of zero length.

After the last record is read, the channel is positioned to the end of the file. When the channel is positioned at the end of the file, p_read returns the negative E_FILE_EOF (nothing is written to buf).

Other negative error returns are:

E_FILE_ABORT	the SSD containing the file is or was previously not present and the channel is now in the abort state
E_FILE_READ	failed to read from the file (eg because of a CRC failure on a floppy disk)

Example

```
LOCAL_D VOID *fcb=NULL;

LOCAL_C INT CDECL SearchFile(TEXT *file, TEXT *pattern)
    {
    INT ret;
    TEXT line[P_FMAXRSIZE+2];

    if (ret=p_open(&fcb,file,P_FTEXT))
        panic("Failed to open file",ret);
    while ((ret=p_read(fcb,&line[0],P_FMAXRSIZE))>=0)
        {
        line[ret]=0;
        if (ret && p_smatchi(&line[0],pattern))
            p_printf(&line[0]);
        }
    p_close(fcb);
    if (ret!=E_FILE_EOF)
        panic("Failed to read file",ret);
    return(0);
    }
```

p_write — Write to a text file channel

```
INT p_write(VOID *pfcb, VOID *buf, UINT len);
```

Write a record of length len bytes (where len is zero to P_FMAXRSIZE inclusive) from buf to the text file channel pfcb. Returns zero if successful or one of the following negative error numbers:

E_FILE_FULL	not enough room on the medium for the data
E_FILE_RECORD	the record size exceeds P_FMAXRSIZE
E_FILE_RDONLY	the file channel is opened without P_FUPDATE being set in mode
E_FILE_ABORT	the SSD containing the file is or was previously not present and the channel is now in the abort state
E_FILE_WRITE	failed to write to the file (eg because of a CRC failure on a floppy disk)

Records are *always* written to the end of file.

The data between buf and buf+len should not include any record delimiters.

p_seek (or f_seek) — Position a text file channel

```
INT p_seek(VOID *pfcb, INT sense, LONG *ppos);
INT f_seek(VOID *pfcb, INT sense, LONG *ppos);
```

Set the current record position of text file channel pfcb to a position which depends on sense and *ppos, where sense is one of:

P_FREWIND	to position to the first record (the value of ppos is ignored)
P_FRSENSE	to get the position of the last record read or written
P_FRSET	to set the record position which was previously got with a call to p_seek(P_FRSENSE)

Returns zero if successful or the negative E_FILE_INV if the file was not opened with P_FRANDOM.

Some remote file systems may not be capable of supporting `p_seek(P_FRSET)` and `p_seek(P_FRSENSE)` and in this case `p_seek` will return `E_FILE_INV`.

The function `f_seek` is identical to `p_seek` except that, if there is an error, it calls `p_leave(err)` rather than return the negative error number `err`.

It is important to note that this function sets the current record position for read operations only. Records are always written at the end of file.

Example

```
while ((len=p_read(fcb,&buf[0],P_FMAXRSIZE))>0)
    {
    buf[len]=0;
    if (buf[0]==':')
        {
        p_seek(fcb,P_FRSENSE,&pos);
        StoreLabel(&buf[1],pos);
        }
    }
```

p_iow(P_FFLUSH) Flush internal file buffers

```
INT p_iow(VOID *pfcb, P_FFLUSH);
```

Flush all buffered written data to text file channel `pfcb` and write the file's modification date.

The behaviour and returns are as for flushing a binary file, described in the previous section. Unlike the binary file, however, data is buffered for all file systems and media.

p_iow(P_FSETEOF) Set end of text file

```
INT p_iow(VOID *pfcb, P_FSETEOF, ULONG *peof);
```

Set the logical end of file on channel `pfcb` to position `*peof`.

Behaviour and returns are as for flushing a binary file, described in the previous section.

p_iow(P_FCANCEL) Cancel an asynchronous file channel request

```
INT p_iow(VOID *pfcb, P_FCANCEL);
```

Cancel all asynchronous I/O requests on channel `pfcb` and return zero.

The behaviour and return values are as for cancelling a binary file request, described in the previous section.

At the time of writing, a preferable alternative is to simulate a cancel service, as illustrated in the earlier example, under the heading *Asynchronous file operations*.

CHAPTER 12

PROCESSES AND INTER-PROCESS MESSAGING

Processes

A process is a running program. It is normally created by loading an image file (also called an executable) using `p_execc`. After loading the program, the process creator normally calls `p_presume` to start the process running.

EPOC is a single-user multi-tasking operating system that rapidly switches contexts between a number of independent processes - creating, at times, the illusion of multiple processes running in parallel.

The maximum number of processes that may exist is `E_MAX_PROCESSES` (24).

At any particular time, one process is actually running. On a reschedule, EPOC runs the process with the highest priority that is ready to run. If there is only one ready process at the highest priority it will continue to run indefinitely (and any lower priority ready processes will wait indefinitely).

Preemptive multi-tasking means that the running process may be replaced at any time - it does not have to make a system call to yield the processor. A process which becomes ready will immediately run if it has the highest priority.

If there is more than one ready process with the highest priority then each process is made current for a fixed time period (4 system ticks) after which the operating system makes the next process of that priority current in what is called a "round robin" fashion. On a SIBO machine, the system "ticks" 32 times a second.

Because EPOC is a single-user system, the current process is comparatively rarely switched out by the system tick. It is more likely to stop running because an event occurred that made a higher priority process become ready or because the current process voluntarily gave up its ready status to wait for an event to occur.

In EPOC, a process consists of at least the following:

- a process control block (described below)
- a data segment, containing the processor stack, static variables and the heap (as described in the *Memory Allocation* chapter)
- a primary code segment (which is shared if there are one or more other processes of the same program)
- an I/O semaphore (as described in the chapter *Asynchronous Requests and Semaphores*)

On SIBO machines, a process that tries to write to a memory segment other than its own data segment (except via functions that explicitly allow such activity) is panicked with panic number 60.

A code segment may exist in ROM or it may be loaded into a RAM memory segment from an executable.

A process may acquire other resources during its lifetime - for example, I/O channels or additional code segments. The system automatically releases owned resources such as memory segments, I/O channels and semaphores when it terminates. Server processes are also designed to clean up client resources when a client process terminates.

System processes

When the operating system initialises itself, it creates a number of system processes some of which are essential to the operation of EPOC.

After a system reset on a typical machine running EPOC, the system processes (by process name) are:

`SYS$NULL.$01`	the zero priority null process runs when no other process is ready to run and switches the machine off (to reduce power consumption) after a period of inactivity
`SYS$MANG.$02`	the supervisor has a higher priority than any other process and performs many critical system functions including memory segment moving and resource clean-up when a process terminates
`SYS$FSRV.$03`	the file server (described in the *Files* chapter) has the second highest priority and performs all file related operations, including loading an image to create a process
`SYS$WSRV.$04`	the window server (described in the *Window Server* manual) provides shared access to the screen, keyboard and, if present, the digitiser (or mouse on a PC)
`SYS$SHLL.$05`	the shell process provides a user interface that allows other processes to be started (on a custom system it might provide a "turnkey" environment)

The extension of the process name gives the process number (assigned by the operating system). The structure of process names is described later in this chapter.

The functions (described in this chapter) which change the process name, change the process priority and suspend a process will fail when applied to `SYS$NULL`, `SYS$MANG` and `SYS$FSRV`.

On systems with a digitiser (or mouse), the window server creates a subsidiary process, sharing its data segment, to draw the mouse icon. In EPOC, a subsidiary process that shares the data segment of its owner is called a *task* and the window server task has a name such as `SYS$WSRV.@05`.

The notifier service `p_notify` (described in the chapter *Error Handling*) may be provided by the window server itself or it may be provided by a client of the window server. If a separate notifier exists, it has a process name such as `SYS$NTFY.$07`.

Process ID and process control block

Each process is identified by its process ID - a positive 16-bit number containing two bit fields:

- The least significant 12 bits is the offset of the process control block in the operating system data segment (also called the process slot).
- The most significant 4 bits contains a value in the range 0-7 (note that a valid process ID is positive). This value is incremented modulo 8 each time a process slot is used so that a process ID may be rejected after a process has terminated.

The function `p_getpid` returns the process ID of the caller.

If a system function is passed a process ID that is zero or negative, or in which the least significant 12 bits is outside the range of the process table, the caller is panicked with panic number 7 (invalid process ID).

The structure of the process control block is defined by the E_PROC struct, defined in *epoc.h* as:

```
typedef struct e_proc
    {
    struct e_proc *next;
    struct e_proc *prev;
    WORD queKey;
    WORD queData;
    UBYTE deltaType;
    UBYTE addressTrap;
    UBYTE status;
    UBYTE sstatus;
    UBYTE priority;
    UBYTE priorityH;
    UBYTE ramOrRom;
    UBYTE isTask;
    UBYTE name[E_MAX_NAME+1];
    UBYTE active;
    UWORD semaphore;
    UBYTE *semHead;
    UBYTE *memBasePtr;
    UWORD memGrowBy;
    UBYTE *mCtrlPtr;
    UWORD minHeap;
    HANDLE fServer;
    HANDLE dataSeg;
    HANDLE codeSeg;
    UBYTE *saveSP;
    UBYTE *saveBP;
    UBYTE notify;
    UBYTE sndSem;
    UWORD magic;
    UWORD checkSum;
    UWORD terminate;
    } E_PROC;
```

A copy of a process control block may be obtained by calling p_getosd (where E_PIDMASK is used to mask out the address portion from the process ID) as follows:

```
E_PROC pcb;

p_getosd(&pcb,(VOID *)(pid&E_PIDMASK),sizeof(pcb));
```

Many of the fields of E_PROC are described in the course of this chapter. The remaining fields are reserved for system use. As an aside, and for the reader's interest, some miscellaneous fields (which would not otherwise be described) are described below. It must be emphasised that none of these fields should be modified directly by any application code.

sstatus	TRUE if the process is waiting to be suspended (for example, if suspended while on the time delta queue - on leaving the queue it will be suspended rather than entering the ready queue)
ramOrRom	TRUE if the process code segment is in RAM, FALSE if the code segment is in ROM
isTask	TRUE if the process is a task (a subsidiary process that shares the data segment of its creator)
active	TRUE if the running of this process will stop the machine from switching off - as set by p_marka and p_unmarka
semaphore	the handle of the process I/O semaphore
memBasePtr	the address of the start of the heap (4 bytes before that obtained from p_allspace)
memGrowBy	the heap granularity in paragraphs as set by p_hgran
mCtrlPtr	contains the address of the message control block as set up by p_minit or zero if p_minit has not been called
minHeap	the minimum heap size in paragraphs

fServer	the client ID as given by the file server or zero if the process has not connected to the file server
dataSeg	the handle of the process data segment
codeSeg	the handle of the process code segment if the code is in RAM or the paragraph address of the code segment if the code is in ROM
notify	the notifier state as set by p_sentnotify (and sensed by p_getnotify)
sndSem	TRUE if the process is waiting on the sound semaphore, to indicate the need to call p_signal on termination of the process
magic	the top 4 bits of the process ID, used to reject a process ID of a process that no longer exists - even when another process has been created and re-uses the same process slot
terminate	contains the termination message type as set by p_onterminate or zero if p_onterminate has not been called

Process states

Each process is in one of the following states (as stored in pcb.status):

E_PROC_CURRENT	the process that is currently running (at any time, one process is in this state)
E_PROC_READY	the process is waiting for a chance to run
E_PROC_SEMAPHORE	the process is waiting on a semaphore, most likely its I/O semaphore pcb.semaphore (described in the chapter *Asynchronous Requests and Semaphores*)
E_PROC_DELTA	the process is waiting in the timer delta queue (described in the chapter *Time, Timers and Dates*)
E_PROC_SUSPENDED	the process exists but will not run until it is resumed by calling p_presume

Process queues

Processes that are in the READY, SEMAPHORE or DELTA state are in a doubly-linked queue (using pcb.next and pcb.prev - see queues in the chapter *Characters, Strings, Buffers and Queues*).

The READY queue is ordered by the process priority as stored in pcb.priority.

Each semaphore heads a SEMAPHORE queue. This queue may be empty, or may contain one or more processes in a first-in, first-out order.

The DELTA queue is a special kind of doubly-linked queue (called a delta queue, as described in *Characters, Strings, Buffers and Queues*). It stores the time interval in system ticks between timer entries.

These queues are described in more detail in the chapter *Asynchronous Requests and Semaphores*.

Process priorities

When there is more than one process that is ready to run, the operating system runs the process with the highest priority (an unsigned byte value in the range 1 to 255). Lower priority processes are blocked indefinitely.

If there is more than one ready process with the same highest priority, they take it in turns to run every four system ticks.

Applications should set their priority in the range E_MIN_PRIORITY (64) to E_MAX_PRIORITY (192) inclusive (the operating system reserves the values outside this range).

The initial process priority is normally taken from a value stored in the program file or image (and is generated by the tool used to build the image) but it may subsequently be changed using p_setpri (p_getpri returns the priority of a process).

Interactive application processes that are clients of the window server are normally created at the priority E_PRIORITY_FORE (128) and subsequently leave it to the window server to change their priority depending on whether the process is receiving user input or not. See the *Window Server* manual.

The supervisor runs at priority 248 and the file server at priority 240. A hardware interrupt runs at the same priority as the process it interrupts. Critical sections of interrupt code are protected by switching off pre-emption.

Preemptive scheduling

It is possible for a process to run for as long as it wants to and, in practice, this often happens. In this case, the current process eventually gives up its CURRENT state by changing its state to:

SEMAPHORE	by calling p_iowait to wait on the I/O semaphore (or p_wait to wait on any semaphore)
DELTA	by calling p_sleep, p_sleept or p_sleepa
SUSPENDED	by calling p_psuspend on its own process ID

However, many events can cause a reschedule in which the current process is *preempted* by a higher priority process without completing its course. Such events include:

- the fourth consecutive system tick
- a semaphore being signalled (eg as a result of user input) which releases a higher priority process
- the expiry of a higher priority process in the timer DELTA queue

The current process may, by its own action, cause itself to be preempted by:

- signalling the I/O semaphore of a higher priority process (by calling p_iosignalbypid or, for example, by sending it an inter-process message)
- calling p_presume to release a higher priority process from the SUSPENDED state (especially after loading a process from an image)
- raising the priority of a another process by calling p_setpri
- lowering its own priority by calling p_setpri

Process names

A process name takes the following form:

```
<name><ext>
```

where the <name> component contains between one and eight characters and the <ext> component consists of a period, a $ and a two digit number of the form 01, 02, 03 .... This number gives the index (starting from 1) of the process slot.

If the process is a *task* (a subsidiary process that shares the same data segment), the $ is replaced by a @ where, except for the @, the process name is otherwise the same as the creator of the task.

The maximum length of a process name is E_MAX_NAME (12), excluding the zero terminator (buffers normally allow E_MAX_NAME+2 bytes to include the zero terminator and to keep following variables on an even address).

When a process is created by loading an image using p_execc, the process is created with a <name> taken from the file name of the image. For example, if a program called LOC::\B:\UTILS\SORT.IMG is loaded twice the process names might be:

```
SORT.$07
SORT.$11
```

The <ext> component of a process name guarantees that the process name is unique. Note that the process data segments are given the same names (see the chapter *Memory Allocation*) but that these segment names are held independently of the process name.

To convert a process name into a process ID, you use p_pidfind. Since a program cannot anticipate the <ext> component, p_pidfind allows wild card characters in a match string. For example, to get the process ID of a database server process that was loaded from *db$serv.img*, you would use p_pidfind("DB$SERV.*").

When there is the prospect of more than one process of the same <name> (because a program was loaded more than once), p_pfind may be used to get the process IDs of all instances.

In the following example, the general purpose function ProcsOfThisProg returns the number of processes having the same process name (excluding the extension) as this process (that is, it returns the number of processes of the calling program).

```
GLDEF_C INT ProcsOfThisProg(VOID)
    {
    TEXT *pp;
    HANDLE h;
    INT count;
    TEXT mm[E_MAX_NAME+2];
    TEXT bb[E_MAX_NAME+2];

    p_pname(p_getpid(),&mm[0]);
    pp=(&mm[p_slocr(&mm[0],'.')+1]);
    *pp++='*'; *pp=0;
    for (h=0,count=0;(h=p_pfind(h,&mm[0],&bb[0]))>=0;count++);
    return(count);
    }
```

The function gets the name of this process using `p_pname` and `p_getpid` and builds a match string in `mm[]` by replacing the extension with the `'*'` wildcard. It then uses this match string to count the number of matching processes.

This function could be used to stop more than one process of a program (especially a server) from being executed where, for example, if `ProcsOfThisProg` returns more than one, the server program could panic.

Reserved statics

The reserved statics (otherwise known as magic statics) are variables with fixed, known, addresses, existing in the process data space between addresses 0x00 and 0x40. They are 'magic' in the sense that they are accessible from all parts of the process code, even from dynamic library code (which does not have any data space and therefore may not, normally, access statics).

Some of these variables are used by the operating system as part of the process context. Others are used by system code such as the window server, and the graphics user interface libraries. Such usage differs from machine to machine in the EPOC range; you will find a description of any such usage in the *Programming Guide* for the appropriate machine or in user interface library documentation.

The remaining reserved statics are freely available for use by the process code. A common use is to provide access from dynamic library code to application-specific data, without the need for it to be passed in function parameters.

`0x00 DatWordDead`
The word at this address contains the value `0xDEAD`. This value must not be changed by application code. Many operating system calls check for this value and will panic the process with panic reason code `PanicDead0` if it has changed. A change in this value is symptomatic of a common software bug; the unintentional use of a `NULL` pointer.

`0x02 DatHandNext`
`0x04 DatHandPrev`
The data at these two addresses are used as pointers to a queue of wait handler function descriptors. This data should not be modified by application code.

`0x06 DatCountrySeg`
This holds the segment handle of the data segment containing the country and language specific fold tables for the process. This data should not be modified by application code.

`0x08 DatClassHandle`
`0x0a DatClassPtr`
In applications which use any of the object oriented programming calls (this includes use of Hwif programs) these locations hold data required by the operating system to work out how to send a message to an object's superclass. It is recommended that application code does not modify the contents of these locations.

`0x0c DatEClassHandle`
`0x0e DatEClassPtr`
In applications which use any of the object oriented programming calls (this includes use of Hwif) these locations hold data required by the operating system to work out how to perform a `p_exactsend`. It is recommended that application code does not modify the contents of these locations.

`0x10 DatEnterFramePtr`
The enter and leave mechanism provided by the operating system uses this address to store the pointer to the last enter frame generated on the stack. When a leave occurs this pointer is used to unwind the stack and restore the register set. This location should not be modified by application code.

`0x12 w_ws`
In applications that use system user interface libraries this location is assumed to hold the object handle of an instance of the 'wserv' object. The application is responsible for ensuring that this location is set up correctly, normally by calling system code on application start-up.

0x14 w_am	In applications that use the application manager object (all object oriented programs and Hwif programs) this location is assumed to hold the object handle of an instance of the application manager object. The application is responsible for ensuring that this location is set up correctly, normally by calling system code on application start-up.
0x16 wClientData 0x18 wserv_channel	These locations are used by the window server process to store information about your application. If you use any graphics functions you should not modify the contents of these locations.
0x1a T	This location is used by the OPL language translator. If your application does not use OPL then this location is free for use by application code.
0x1c r	This location is used by the OPL language runtime code. If your application does not use OPL then this location is free for use by application code.
0x1e DatOsFramePtr	This location contains a pointer to the last member of a linked list of operating system calling frames. Following this linked list will show which functions called which operating system services. This location should not be modified by application code.
0x20 DatATFlag	This byte location contains the current address trap status for the process. Certain operating system calls cause address trapping to be turned on or off and the operating system sets or clears this flag to record the current address trap hardware status. Just setting the flag to zero will *not* disable address trapping. This location should not be modified by application code.
0x21 DatHeapLocked	This byte location contains the current state of the heap locked flag. While the heap is being modified (for example, being resized, due to a memory allocation request) the heap is locked. This prevents the operating system from compressing the segment while the data structures used by the operating system to run the heap are in an inconsistent state. This location should not be modified by application code.
0x22 DatProcessNamePtr	System user interface library code assumes that this location contains either NULL or a pointer to a zero terminated string that the application wishes to be displayed as its name. Otherwise it is free for use by application code.
0x24 DatCommandPtr	This location contains a pointer, set up by the operating system. It points to an alloc cell that contains the full path name, as a zero terminated string, of the *.img* or *.app* file from which the process was loaded. Immediately following the zero terminator, the alloc cell contains leading byte counted initial command line data.
0x26 DatTest	Psion's test system library code assumes that this location contains the object handle of the test code. Otherwise it is free for use by application code.
0x28 DatApp1 0x2a DatApp2 0x2c DatApp3 0x2e DatApp4 0x30 DatApp5 0x32 DatApp6 0x34 DatApp7	These locations are free for application code to use.
0x36 DatDialogPtr	System user interface library code may assume that this location contains a pointer to the current dialog structure. Otherwise it is free for use by application code.
0x38 DatGate	System user interface library code may assume that this location contains an object handle. Otherwise it is free for use by application code.
0x3a DatLocked	System user interface library code may assume that this location contains a flag indicating whether the application is capable of receiving termination or switch files messages. Otherwise it is free for use by application code.
0x3c DatStatusNamePtr	System user interface library code may assume that this location contains a pointer to a zero terminated string that will appear in a status window. Otherwise it is free for use by application code.

0x3e DatUsedPathNamePtr	System user interface library code may assume that this location contains a pointer to a fully parsed file name. Otherwise it is free for use by application code.

Shared code segments

When a second or subsequent process of the same program is loaded, the existing code segment (which has segment name <name>.$SC) is shared and not loaded from the image file (however, the initialized static data values are still loaded into the created process data segment).

The system checks that the contents of the existing code segment matches the code in the image file by comparing pcb.checkSum with the checksum stored in the image header (the checksum in the image header is also used to check the integrity of the code when it *is* loaded). If the checksums do not match, the load fails. It follows that you cannot run different programs of the same name (or different versions of the same program) at the same time.

Image files

An image file is a form of executable (program) file that is run by calling p_execc. It normally has a .IMG file name extension. Application files (with a .APP extension) are a particular type of image file.

An image file is created by applying the *emake.exe* tool to a DOS executable. This is normally done automatically as part of the build process that generates an EPOC application program (for an example of the use of *emake.exe*, see *\ts\sys\tsprj.txt*).

The creation process adds an ImgHeader struct (defined in *epoc.h*) to the front of the file and may optionally concatenate a number of other files into the image file.

The ImgHeader struct is effectively defined as:

```
#define SignatureSize 16
#define MaxAddFiles 4

typedef struct
    {
    UINT offset;
    UINT length;
    } ADDFILE;

typedef struct
    {
    UBYTE Signature[SignatureSize];
    UINT ImageVersion;
    UINT HeaderSizeBytes;
    UINT CodeParas;
    UINT InitialIP;
    UINT StackParas;
    UINT DataParas;
    UINT HeapParas;
    UINT InitializedData;
    UINT CodeCheckSum;
    UINT DataCheckSum;
    UINT CodeVersion;
    UINT Priority;
    ADDFILE Add[MaxAddFiles];
    UINT DylCount;
    ULONG DylTableOffset;
    UINT Spare;
    } ImgHeader;
```

The meanings of the elements of this struct are as follows:

Signature	contains the string "ImageFileType**".
ImageVersion	the version number of the software tools used to create the image file. At the time of writing the version is 2.00F (0x200f)
HeaderSizeBytes	the offset of the start of the executable code within the image file
CodeParas	the required size, in (16 byte) paragraphs, of the memory to be reserved for the code segment

`InitialIP`	the initial instruction pointer offset in the executable code - determined by the linker, (and normally 0)
`StackParas`	the size, in (16 byte) paragraphs of the stack - determined by which PLIB C startup module is linked into the program (see the discussion of startup modules in the *Introduction* chapter)
`DataParas`	the total size, in (16 byte) paragraphs, of the declared static data (including the initialised data - see below)
`HeapParas`	the required size, in (16 byte) paragraphs, of the initial - and minimum - alloc heap, user definable via a parameter to *emake.exe* (see below)
`InitializedData`	the size, in bytes, of the initialised static data
`CodeCheckSum`	internally generated and verified checksum on the code
`DataCheckSum`	internally generated and verified checksum on the data
`CodeVersion`	the version number of the executable code, user definable via a parameter to *emake.exe* (see `DbfVersion` in the chapter *Database Files* for the form of a version number)
`Priority`	the start-up priority of the process that will be created from this image file, user definable via a parameter to *emake.exe* (default value is `0x80`)
`Add`	an array of four ADDFILE structs, describing up to four additional files included within the image file (common included files, in .APP files, are an icon graphic and a resource file)
`DylCount`	the number of object dynamic libraries (DYLs) concatenated into the image file. See also the *Object Oriented Programming* chapter of this manual.
`DylTableOffset`	the offset within the file of the start of an array of `DYLENTRY` structs (defined in *epoc.h*) with `DylCount` entries, giving the names and file offsets of the included DYLs
`Spare`	reserved

The contents of this header may be displayed by applying the *edump.exe* tool to an image file.

For efficiency, the value of `HeapParas` should be adjusted to be equal to, or slightly larger than, the heap space used by the running process immediately after it has started. Setting a smaller value means that the heap will have to be grown one or more times during the initialisation of the process. Growing the heap may involve moving large amounts of in-memory data and may, therefore, significantly increase the process start-up time (see also *The heap allocator* in the chapter *Memory Allocation*).

Process termination

The subject of process termination (with associated functions) is described in detail in the chapter *Error Handling*. The functions described include those that terminate a process:

`p_pterminate` or `p_pkill`	to terminate another process (typically in response to a user request)
`p_ppanic`	to panic another process (normally following unreasonable behaviour from the process being terminated)

and those functions that request notification of the termination of other processes:

`p_logona`	to be signalled when the specified process terminates
`p_logon`	to receive an inter-process message when the specified process terminates (convenient for server processes to keep track of their clients)
`p_watchall`	to receive an inter-process message when any process terminates (only one process can call `p_watchall` - normally the Shell to monitor the termination of all processes)

When a process terminates, pcb.status in the process control block (which normally holds the process state) is set to E_PROC_FREE to mark it as available for use by a new process. When a newly created process re-uses the slot, pcb.magic is incremented modulo 8, as described above.

Creating a process

p_execc — Load an image

```
HANDLE p_execc(TEXT *pName, VOID *pCommand, INT length);
```

Create a suspended process from the image file described by the zero terminated file specification pNname and, if successful, return the positive process ID of the created process.

The created process is suspended so that the creator can perform any initialisation (eg to set the process priority) before allowing the process to run by calling p_presume. If the created process has a higher priority than the creator, the call to p_presume may not return for some time.

The file specification pName is parsed with a related name of ".IMG".

The function allocates a command cell from the heap of the created process. This cell is loaded with the following sequence:

- a copy of the zero terminated full file specification resulting from the parse of pName
- a byte containing length
- a copy of the length bytes from pCommand

where length must be between zero and E_MAX_COMMAND_BUFFER (127) inclusive. If pCommand is NULL, length is taken to be zero, whatever is passed.

The address of this command cell is written to the reserved static DatCommandPtr in the created process data segment. This variable is accessed from C by declaring:

```
GLREF_D UBYTE *DatCommandPtr;
```

Note that there is no restriction on the data in pCommand; it may include binary data structures as well as text. For example, it is often useful to pass the process ID of the caller in the command line.

The name of the created process is taken from the file name component (excluding any extension) of pName. If a process, loaded from pName, already exists (as determined by a name and checksum match), the created process shares the loaded code segment. In this case, the code is not reloaded from pName (however, the initialized static data values are still loaded into the created process data segment).

The function returns a positive process ID if successful or a negative error number if it failed. As well as the p_fparse error numbers that may be returned if the parse of pName fails, p_execc can return the following error numbers:

E_GEN_NOMEMORY	insufficient free system memory
E_FILE_DEVICE	the device in pName does not exist
E_FILE_NOTREADY	the device in pName does not contain a medium
E_FILE_DIR	the directory in pName does not exist
E_FILE_NXIST	the file in pName does not exist
E_FILE_EXIST	a process of a different program (ie with a different checksum) with the same name already exists
E_GEN_IMAGE	the file is not a valid image or if the image is corrupt
E_GEN_NOPROC	there are no more free process slots (the maximum process limit has been reached)
E_GEN_ARG	the total size of the data, stack and heap exceeds 0xFFE0 bytes or length exceeds E_MAX_COMMAND_BUFFER

The loading of an image is performed by the file server (see the *Files* chapter) and the caller must have connected to the file server before calling p_execc (otherwise the caller is panicked with panic number 41). The standard PLIB library connects to the file server before calling main.

Example

```
#include <plib.h>

LOCAL_C VOID Exec(TEXT *name, TEXT *cmd, INT len)
    {
    INT ret;
    TEXT bb[E_MAX_ERROR_TEXT_SIZE];

    ret=p_execc(name,cmd,len);
    if (ret<0)
        {
        p_errs(&bb[0],ret);
        p_printf("Failed to start %s (%s)",name,&bb[0]);
        }
    else
        {
        p_printf("Process ID is %x",ret);
        p_presume(ret);
        }
    }

GLDEF_C INT main(VOID)
    {
    TEXT *p1,*p2;
    TEXT bb[64];

    while (p_getl("Enter <img> <cmd> ? ",&bb[0],64))
        {
        p1=p_skipwh(&bb[0]);
        if (!*p1)
            continue;
        p2=p_skipch(p1);
        if (*p2)
            {
            *p2++='\0';
            p2=p_skipwh(p2);
            }
        Exec(p1,p2,p_slen(p2)+1);
        }
    return(0);
    }
```

The program solicits a line of input and parses out the name of the image to run and a text string to pass to the created process (which is passed with the zero terminator). If this program is used to run *dummy.img* in the default path (which happens to be LOC::D:\) by entering:

```
dummy fred
```

and the code for *dummy.img* is:

```
#include <plib.h>

GLREF_D UBYTE *DatCommandPtr;

LOCAL_C VOID PrintCommandPtr(VOID)
    {
    UBYTE *p,*pe;
    UINT lcell;

    if (!DatCommandPtr)
        p_panic(0);
    lcell=p_alen(DatCommandPtr);
    p_print("%d [",lcell);
    for (p=DatCommandPtr,pe=p+lcell;p<pe;p++)
        p_print(p_isprint(*p)?"%c":"<%02x>",*p);
    p_printf("]");
    }

GLDEF_C INT main(VOID)
    {
    PrintCommandPtr();
    p_getch();
    return(0);
    }
```

then *dummy.img* prints:

```
24 [LOC::D:\DUMMY.IMG<00><05>fred<00>]
```

p_execcasync — Load an image asynchronously

```
INT p_execcasync(TEXT *pName, VOID *pCommand, INT length, WORD *pStatus, HANDLE *pPid);
```

This is the asynchronous version of p_execc, which may return before the operation is complete. See the chapter *Asynchronous Requests and Semaphores* for an explanation of asynchronous requests.

See p_execc above for the meaning of the parameters pName, pCommand and length, the behaviour of the operation and its possible error returns.

The function p_execcasync returns zero if the asynchronous request was successful or a negative error number if the operation failed to start (eg E_FILE_NAME if pName failed to parse).

When the load completes, the completion status is written to *pStatus and the process I/O semaphore is signalled. If the load completes successfully, *pStatus contains zero and *pPid contains the process ID of the created process (this corresponds to the value returned by p_execc). If the load completes unsuccessfully, *pStatus contains a negative error number (eg E_GEN_NOMEMORY).

If pName refers to a file on the LOC:: file system, the load will actually have completed by the time p_execcasync has returned because the file server runs at a higher priority than any of its clients. However, if the image is on a REM:: file system where the connection is via an RS232 cable, p_execcasync will return well before the load is complete.

p_pcreate — Create a process

```
HANDLE p_pcreate(E_CPB *pBlock);
```

Create a process from the information in the E_CPB struct pointed to by pBlock and, if successful, return the positive process ID of the created process.

This is the primitive process creation service that does not involve the file server. In the vast majority of cases, it is more convenient to use p_execc, which will eventually call this service.

The E_PCB struct is defined in *epoc.h* as:

```
typedef struct
    {
    UWORD codeParagraphs;
    UWORD initialIp;
    UWORD stackParagraphs;
    UWORD dataParagraphs;
    UWORD heapParagraphs;
    UBYTE *commandLine;
    UWORD checkSum;
    UWORD minHeap;
    UBYTE priority;
    UBYTE ramOrRom;
    UBYTE name[E_MAX_NAME];
    } E_CPB;
```

The function creates a process of initial priority pBlock->priority where the process name is taken from the zero terminated <name> (up to 8 characters) in pBlock->name.

If pBlock->ramOrRom is TRUE and a process of the same name already exists, its checksum is compared with pBlock->checkSum and, if they match, the existing code segment is shared (in which case pBlock->codeParagraphs is ignored).

If pBlock->ramOrRom is TRUE and a process of the same name does not already exist, p_pcreate creates a code segment of length pBlock->codeParagraphs paragraphs (and pBlock->checkSum is stored as the code segment checksum). In this case, the caller must deposit the code into the created code segment before resuming the process.

If pBlock->ramOrRom is FALSE, pBlock->codeParagraphs is taken to be the paragraph address of the code segment in the ROM.

When the process is resumed, the initial instruction pointer is set to pBlock->initialIp.

A data segment is created, of sufficient size to include the stack (of length pBlock->stackParagraphs) a static data space (of length pBlock->dataParagraphs) and a heap (of length pBlock->heapParagraphs). The total size of the data, stack and heap must not exceed 0xFFE paragraphs. The pBlock->dataParagraphs area in the data segment is zero filled. The minimum heap size subsequently allowed for the created process is pBlock->minHeap paragraphs.

If pBlock->commandLine is not NULL it should point to a zero terminated string, immediately followed by a leading byte count buffer where the leading byte is less than or equal to E_MAX_COMMAND_BUFFER (as described in p_execc, above). The whole pBlock->commandLine data structure is copied into a heap cell that is allocated in the created process.

The function returns a positive process ID if successful or on of the following negative error numbers:

E_GEN_NOMEMORY	insufficient free system memory
E_FILE_EXIST	a process of a different program (ie with a different checksum) with the same name already exists
E_GEN_NOPROC	there are no more free process slots (the maximum process limit has been reached)
E_GEN_ARG	the total size of the data, stack and heap exceeds 0xFFE0 bytes or the leading byte count length in the pBlock->commandLine data structure exceeds E_MAX_COMMAND_BUFFER
E_FILE_NAME	pBlock->name is invalid

Operations on the current process

See the chapter *Error Handling* for the functions (p_exit and p_panic) that terminate the current process.

p_getpid — Get this process ID

```
HANDLE p_getpid(VOID);
```

Return the process ID of the caller.

For example:

```
TEXT ProcessName[E_MAX_NAME+2];

p_pname(p_getpid(),&ProcessName[0]);
```

writes the name of this process as a zero terminated string to `ProcessName`.

p_unmarka — Mark this process as non-active

```
VOID p_unmarka(VOID);
```

Mark the calling process as non-active such that the running of the process will not keep the machine switched on.

The zero priority null process switches the machine off (to reduce power consumption) after a period of inactivity. A context switch to a process marked as non-active does not count as activity.

When a process is created it is initially marked as active.

General purpose server processes should mark themselves as non-active since not to do so would disable their clients from effectively calling `p_unmarka`. For example, if the file server was marked as active, a file request by a non-active client would cause the file server to run and reset the inactivity timer. As it is, the file server is marked as non-active and it is left to the clients of the file server to reset the inactivity timer or otherwise.

Applications that respond to a continuously restarted short interval relative timer (as in, for example, a clock program) should also call `p_unmarka`. Otherwise, the machine would never switch off while the program is running.

Interactive programs that do call `p_unmarka` do not need to take any measure to reset the inactivity timer when responding to user input (a key press or the use of a pointing device if there is one) since the system (by one means or another) guarantees to reset the inactivity timer on user input. To reset the inactivity timer in response to an event other than user input, call `p_tickle`, described next.

Note that a compute-bound process (such as a game program that is computing its best next move) will not allow the machine to switch off, simply because the null process never gets an opportunity to run. Such a process should assume responsibility for allowing the machine to switch off by calling `p_allowoff` from time to time.

p_tickle — Register activity

```
VOID p_tickle(VOID);
```

Reset the auto-switch-off inactivity timer.

It is used by a process that is marked as inactive, but wishes to stop the system switching off. An example would be to keep the machine going if serial data is received.

You don't need to call `p_tickle` in response to user input since the system automatically registers activity in this case.

p_marka — Mark this process as active

```
VOID p_marka(VOID);
```

Mark the calling process as active such that the running of the process will stop the machine from switching off.

The zero priority null process switches the machine off (to reduce power consumption) after a period of inactivity. A context switch to a process marked as active resets the inactivity timer.

When a process is created it is initially marked as active and `p_marka` does not need to be called unless `p_unmarka` has previously been called.

Operations on any process

See the chapter *Error Handling* for functions (p_pterminate, p_pkill and p_ppanic) that terminate a process.

p_getpri — Get a process priority

```
INT p_getpri(HANDLE pid);
```

Return the positive priority of process pid, or the negative E_FILE_NXIST if the process does not exist.

p_setpri — Set a process priority

```
INT p_setpri(HANDLE pid, INT nPriority);
```

Set the priority of process pid to nPriority (between E_MIN_PRIORITY and E_MAX_PRIORITY inclusive) and return zero if successful or one of the following negative error numbers:

E_GEN_RANGE	nPriority is outside the range E_MIN_PRIORITY to E_MAX_PRIORITY
E_FILE_NXIST	process pid does not exist
E_GEN_FAIL	attempted to change the priority of the null process, the supervisor, or the file server

Calling p_setpri causes a reschedule.

p_presume — Resume a process

```
INT p_presume(HANDLE pid);
```

Resume process pid and return zero if successful or one of the following negative error numbers:

E_GEN_ARG	pid is not suspended
E_FILE_NXIST	process pid does not exist

If process pid has a higher priority than the caller, the call to p_presume may not return for some time.

If you actually want to wait for the resumed process to terminate before continuing, you can use p_logona (described in the chapter *Error Handling*) as follows:

```
p_logona(pid,&stat);
p_presume(pid);
p_waitstat(&stat);
```

p_psuspend — Suspend a process

```
INT p_psuspend(HANDLE pid);
```

Suspend process pid and return zero if successful or one of the following negative error numbers:

E_FILE_NXIST	process pid does not exist
E_GEN_FAIL	attempted to suspend the null process, the supervisor, or the file server

A process that is in the READY queue or is currently running (ie the calling process) is suspended immediately. A process that is waiting on the SEMAPHORE queue or the DELTA queue is marked (in pcb.sstatus) as requiring suspension and is subsequently placed in the SUSPENDED state when it would otherwise have been transferred to the READY queue (unless the process is resumed before this happens).

p_pname — Get a process name by ID

```
INT p_pname(HANDLE pid, TEXT *pName);
```

Write the name of process pid as a zero terminated string to pName (which should be big enough to receive E_MAX_NAME+2 bytes) and return zero if successful or E_FILE_NXIST if the process does not exist.

The process name written to pName includes the process slot extension as in, for example, SYS$NULL.$01.

p_prename — Rename a process

```
INT p_prename(HANDLE pid, TEXT *pNewName);
```

Rename process pid to the zero terminated pNewName and return zero if successful.

The process name pNewName should not include a process slot extension (this is supplied by the system) and should be between 1 and 8 characters long. No check is made that the new name is unique, but an invalid name will return the error E_FILE_NAME. The possible error returns are:

E_GEN_ARG	process pid is not suspended
E_FILE_NXIST	process pid does not exist
E_FILE_NAME	the new name is invalid

p_pidfind — Get a process ID by name

```
HANDLE p_pidfind(TEXT *pName);
```

Return the positive process ID of the first process having a name matching the zero terminated pName (which may contain wild card characters) or return E_FILE_NXIST if there is no matching process.

For example, to find the process ID of a process created by loading *db$serv.img*, use:

```
pid=p_pidfind("DB$SERV.*");
```

p_pfind — Find all processes

```
HANDLE p_pfind(HANDLE pid, TEXT *pMatch, TEXT *pName);
```

Called repeatedly to find all the processes that match the wild card string pointed to by pMatch, writing the process name as a zero terminated string into pName (which should be big enough to receive E_MAX_NAME+2 bytes). The first call should pass a zero pid.

Returns the positive process ID of the process found (which is also passed to the next find) or the negative E_FILE_NXIST if no more matching processes can be found.

The wild card string pMatch should remain the same between successive calls.

No memory is used by this service and it can be abandoned at any time without taking any further action.

For example, to print the names of all processes:

```
GLDEF_C INT main(VOID)
    {
    HANDLE h;
    TEXT b[E_MAX_NAME+2];

    for (h=0;(h=p_pfind(h,"*",&b[0]))>=0;p_printf(&b[0]));
    p_getch();
    return(0);
    }
```

p_getowner — Determine the owner of a process

```
HANDLE p_getowner(HANDLE pid);
```

This function is only available in EPOC version 2.17 or later.

Returns the process id of the process which last resumed process pid, that is, the last process to call p_presume(pid). This process is defined to be the owner of process pid.

Note that there is no guarantee that the owning process still exists.

Accessing a process data segment

The functions p_pcpyfr, p_piscpyfr and p_pcpyto copy data between a (normally different) process data segment and the caller's data segment (cf p_sgcopyfr and p_sgcopyto which copy data between any segment and the caller's data segment).

p_pcpyfr — Copy data from a process

```
INT p_pcpyfr(HANDLE pid, VOID *pSource, VOID *pTarget, UINT nBytes);
```

Copy nBytes bytes at offset pSource from the data segment of process pid to address pTarget (in the current process) and return zero if successful.

If pSource+nBytes exceeds the size of the processes data segment, the data up to the end of the segment is copied and zero is returned.

The system ensures that the copy is not interrupted by another process.

The function returns E_GEN_ARG if the process pid does not exist.

p_piscpyfr — Indirected string copy from a process

```
INT p_piscpyfr(HANDLE pid, VOID *pSourceAddr, VOID *pTarget, UINT nBytes);
```

This function is only available in EPOC version 2.14 or later.

Read the pointer at offset pSourceAddr in the data segment of process pid. Then copy the zero terminated string indicated by this pointer to address pTarget (in the current process) and return zero if successful.

If the pointer is NULL, a null string ("") is copied to pTarget.

If the string is longer than nBytes, only nBytes of data (plus a terminating zero) are copied and zero is returned.

If the implied length of the source string extends beyond the end of the processes data segment, the data up to the end of the segment (plus a terminating zero) is copied and zero is returned.

The system ensures that the copy is not interrupted by another process.

The function returns E_GEN_ARG if the process pid does not exist.

For example:

```
GLREF_D TEXT *DatProcessNamePtr;

LOCAL_C INT GetCalcName(VOID)
    {
    HANDLE h;
    TEXT buf[0x40];

    h=p_pidfind("calc.*");
    if (h<0)
        return(h);
    p_piscpyfr(h,&DatProcessNamePtr,&buf[0],0x40);
    return(0);
    }
```

fetches the process name of the calculator.

p_pcpyto — Copy data to a process

```
INT p_pcpyto(HANDLE pid, VOID *pTarget, VOID *pSource, UINT nBytes);
```

Copy nBytes bytes from pSource in the current process data segment to pTarget in the data segment of process pid and return zero if successful.

The system ensures that the copy is not interrupted by another process.

Address trapping is automatically switched off for the duration of the copy.

Returns the negative E_GEN_ARG if the process does not exist or if pTarget+nBytes exceeds the size of the target process data segment (in which case the data is not copied).

Inter-process messaging

Inter-process messaging in EPOC is designed for the efficient implementation of client-server relationships where a particular server may have multiple clients.

A server process is a commonly provided to share a resource amongst multiple client processes. The EPOC operating system starts up with three multi-client servers:

- the supervisor, which performs critical system functions and provides shared access to memory via the memory segment allocator
- the file server, which provides shared access to file storage devices
- the window server, which provides shared access to the screen, keyboard and, if present, a pointing device

A server may also be a client of another server. For example, the window server is a client of the file server (since, for example, it loads bitmap files) and all processes are implicitly clients of the supervisor.

In order that the relative priorities of client processes have their intended effect, a multi-client server process should run at a higher priority than any of its clients. A client's priority is effectively lowered to that of the server while waiting for completion of a service provided by a low priority server.

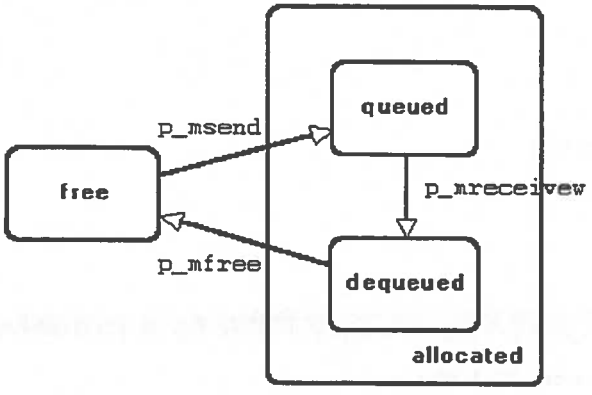
Message slots

A process that wishes to receive messages must first call p_minit to initialise the message system.

Calling p_minit(nMess,lMess) allocates nMess message slots (from the heap) where each message slot contains an E_MESSAGE struct header followed by a buffer of length lMess. The E_MESSAGE struct is defined in *epoc.h* as:

```
typedef struct message
    {
    struct message *next;
    UBYTE *status;
    UINT type;
    HANDLE pid;
    } E_MESSAGE;
```

The structure of the lMess bytes of data following the E_MESSAGE header is defined by the receiver of the message and is normally limited to a few words. For example, the file server and the supervisor both use an lMess of 8. Note that the sender has no control over the length of the message sent (a later version of a server could increase the message length to provide additional services while maintaining upward compatibility).



When the sender calls p_msend (or a variant) the arriving message is copied into a previously free message slot, which is also placed in the receiver's message queue.

From this time the message slot is allocated, in the sense that it cannot be overwritten by another incoming message. It contains the message type (as specified by the sender) in type and the process ID of the sender in pid, followed by lMess bytes of data from the sender. (The fields next and status in the E_MESSAGE header are used internally by the message system.)

The message slot is removed from the queue when the receiver calls, for example, p_mreceivew (which returns the address of the message slot). The content remains safe from being overwritten until the receiver calls p_mfree to indicate that it has completed processing the message. Calling p_mfree returns the message slot to its free state, available to receive another incoming message.

What the server does

A server is always a passive process that waits for messages to arrive from a client process. It is not expected that a server will ever initiate a transaction by sending a message to a client.

As described earlier, the server first calls p_minit to initialise the message system. The server then waits for a message to arrive by calling `p_mreceivew` (an asynchronous version, `p_mreceive`, also exists). When the message arrives, the function returns the address of the message slot.

The server always removes messages from the front of the queue. Arriving messages are normally inserted at the end of the queue but if the client's priority is `0x80` or more, the arriving message is inserted at the front, overtaking any existing messages, regardless of their sender's priority.

When it is necessary to send more than the amount of information allowed for by the (typically short) message length, the message contains the data by reference (ie by address and length). The server then uses `p_pcpyfr` to copy the data from the client's data segment.

The message may also contain the address or addresses of buffers to receive data from the server when the service has been completed. Here, the server uses `p_pcpyto` to copy the data to the client's data segment.

When the message has been processed, the server frees the message slot using `p_mfree`. If the message was sent in such a way that the sender is expecting an acknowledgment of some kind (ie the message was sent using `p_msendreceivew` or `p_msendreceivea` rather than `p_msend`), the call to `p_mfree` also writes back a completion status value and signals the sender's I/O semaphore.

Servers request to be informed of the termination of a client by calling `p_logon(pid,nType)` to receive a message of type `nType` when client `pid` terminates. Process termination and `p_logon` are described in the chapter *Error Handling*.

What the client does

The client process requests a service of the server by sending it typed fixed length messages (where the message length is determined by the server) using:

`p_msend`	to send the message "blind", returning when the message has been deposited (but not necessarily processed)
`p_msendreceivew`	to send the message and wait for the server to reply by calling `p_mfree`
`p_msendreceivea`	to send the message, returning when the message has been deposited (as for `p_msend`) having set up an asynchronous request for a reply

If the server does not have any empty message slots, the message sending function waits on a mutual exclusion semaphore until a message slot becomes free. It can therefore be blocked indefinitely, even if using the asynchronous `p_msendreceivea`. Multi-client servers avoid this prospect by allocating a slot for each potential client (by allocating say `E_MAX_PROCESSES` minus the number of known processes).

The only preparation required by the client is to obtain the process ID of the server. Examples of ways in which this is done are:

- the client gets the ID via the process name using `p_pidfind` (this is normally what is done for multi-client servers)
- the client knows the ID because it created the server using `p_execc`
- the server created the client and passed its process ID as a command parameter to `p_execc` (as in the above example)

For multi-client servers, the client typically sends an opening message to connect to the server.

An example of a server

```
typedef struct
    {
    E_MESSAGE mess;
    TEXT *bofs;
    UWORD len;
    } MESS;
```

```
LOCAL_C VOID RunServer(VOID)
    {
    MESS *pmsg;
    TEXT buf[256];

    FOREVER
        {
        p_mreceivew(&pmsg);
        if (pmsg->mess.type)
            {
            p_mfree(pmsg,0);
            break;
            }
        p_pcpyfr(pmsg->mess.pid,pmsg->bofs,&buf[0],pmsg->len);
        p_printf("%*s",pmsg->len,&buf[0]);
        p_mfree(pmsg,0);
        }
    }

LOCAL_C VOID Exec(TEXT *name)
    {
    HANDLE pid;
    TEXT bb[E_MAX_ERROR_TEXT_SIZE];

    pid=p_getpid();
    pid=p_execc(name,(UBYTE *)&pid,sizeof(pid));
    if (pid<0)
        {
        p_errs(&bb[0],pid);
        p_printf("Failed to start %s (%s)",name,&bb[0]);
        }
    else
        {
        p_printf("Process ID is %xd",pid);
        p_logon(pid,TRUE);
        p_presume(pid);
        RunServer();
        }
    }

GLDEF_C INT main(VOID)
    {
    TEXT bb[P_FNAMESIZE];

    p_minit(1,sizeof(MESS)-sizeof(E_MESSAGE));
    while (p_getl("Enter <img>? ",&bb[0],P_FNAMESIZE))
        Exec(p_skipwh(&bb[0]));
    return(0);
    }
```

In the above example, `main` solicits a file specification of an image to run. This image is then (in `Exec`) loaded using `p_execc` (passing the created process the process ID of its creator) and then resumed using `p_presume`.

While the image is running, the program acts as a server to it where messages of type `FALSE` are taken to contain a buffer by reference. The data from this buffer is copied from the process using `p_pcpyfr` and printed using `p_printf`.

Before resuming the process in `Exec`, the server logs on to the process by calling `p_logon`. When the process terminates, the server receives a type `TRUE` message. This causes the program to return from `RunServer` and solicit another image file specification.

Corresponding client code example

The following example shows how to write that part of the client side code that corresponds to the above example of server code.

```
GLREF_D UBYTE *DatCommandPtr;

LOCAL_D HANDLE pid=0;
LOCAL_D TEXT bb[256];

GLDEF_C VOID printf(TEXT *pfmt,...)
    {
    struct
        {
        TEXT *pbuf;
        UINT len;
        } msg;

    msg.len=p_atob(msg.pbuf=&bb[0],pfmt,&pfmt+1);
    if (!pid)
        pid=*(HANDLE *)(DatCommandPtr+p_slen(DatCommandPtr)+2);
    p_msendreceivew(pid,FALSE,&msg);
    }
```

The `printf` function behaves in the same way as `p_printf` except that the printing is performed by the creator of the process in its console window. Note that the call to `p_msendreceivew` does not return until the server calls `p_mfree`.

This example does not show the sending of the message with a type `TRUE` on termination of the client process.

Asynchronous messaging

The server-client example described above uses synchronous messaging in both the server and the client. While this may be sufficient in simple cases, there are situations where such an implementation will prove inadequate.

A client that sends messages synchronously is effectively suspended from the moment it calls `p_msendreceivew` until the server completes processing the message and calls `p_mfree`. If completion is dependent on an external event, such as the expiry of a timer or the arrival of serial data, the client may be suspended indefinitely. This will, in general, be unacceptable behaviour (particularly if the client must remain responsive to other events, such as user input) and in such a case the client should use asynchronous messaging.

A server does not, in general, have a user interface. If its only task is to receive and process messages from its clients, then synchronous server code may be perfectly acceptable. As in the example server code given earlier, the server is effectively suspended until a message is received and returns to the suspended state as soon as it has finished processing the message. If the server also has to respond to other events, such as the expiry of a timer, then the receipt of messages must be handled asynchronously.

Suppose a client needs to use asynchronous messaging because completion of the processing of the message may be delayed indefinitely by an external event. This implies that the server itself must respond to at least two events (the receipt of a message and the external event) and so must also handle events asynchronously.

The sequence of events that occur during messaging between asynchronous client and server is as follows:

- the server makes a call to `p_minit`
- the server then calls `p_mreceive`, passing the address of its messaging status word, and later makes a call to `p_iowait` (usually in its main event-handling loop)
- the client calls `p_msendreceiva`, passing the address of a status word, and later calls `p_iowait` (usually in its main event-handling loop)
- the server is signalled that a message has been received, by noting that its messaging status word is no longer `E_FILE_PENDING` on a return from `p_iowait`, and commences processing the message
- on completion of the processing the server calls `p_mfree` which notifies the client of the completion
- the client is signalled that processing is complete, by noting that the relevant status word is no longer `E_FILE_PENDING` on a return from `p_iowait`

If the processing of the message itself involves an asynchronous request, the server may process any number of synchronous messages from any of its clients (including the one sending the asynchronous

message) while the request is pending. It is therefore quite possible for a synchronous message to complete before an earlier asynchronous message from the same client.

On the assumption that clients may send messages asynchronously because they do not want to wait, the server will generally need more than one message slot to avoid the client having to wait for a message slot to become free. How many slots to provide depends on many factors, such as:

- the maximum expected number of clients
- the maximum number of outstanding messages allowed per client (rarely more than two)
- whether it is acceptable for any client ever to be suspended while waiting for a free slot

Message processing order

A message sent by a client to a server is placed in a message queue, normally at the end of the queue. The server receives a message by removing the one at the front of the queue (into one of its message slots). Thus, in normal circumstances, messages are processed in the order in which they are sent. As mentioned in the previous section, it is possible for synchronous messages to 'overtake' asynchronous messages from the same client.

The window server has a requirement, particularly in an overlapping window environment, to give priority to messages from the foreground task. For example, when an area of the screen covering all or part of a number of task windows needs redrawing (after, say, the disappearance of a dialog) the foreground task should be redrawn first. The following scheme has been adopted to satisfy the window server's requirement without causing an unacceptable performance penalty.

In all cases, messages from a client process with a priority of `0x80` or above (the foreground task normally has a priority of `0x80`, which is higher than the priority of background tasks) are placed at the front of the queue. They therefore overtake all other messages in the queue (including any earlier messages from the same client) irrespective of the priorities of their sending processes.

This has the added advantage of reducing the risk that switching to the shell task (which always runs at a priority higher than the foreground task) can be blocked by a 'rogue' task.

There is, however, one problem, which can be illustrated as follows. Suppose a client with a priority of `0x80` sends an asynchronous message to a server. This message is inserted at the front of the queue. Before the server removes any messages from the queue, the same client sends a cancel message, to cancel its previous request. This second message is again inserted at the front of the queue, overtaking the message it is supposed to be cancelling.

For this situation to arise, all the following conditions must be satisfied:

- the client must have a priority of `0x80` or above
- the client must use asynchronous messaging
- the client must be sensitive to the order in which the server processes its messages
- the server must not have removed the first message from the queue by the time the second message is queued

The last of these conditions is rarely satisfied, since a server normally runs at a higher priority than its clients. As soon as the server is signalled that a message has been queued the server pre-emptively suspends the client. In most cases the client will not resume until the server has completed processing the message and is waiting for another message. It is rare that a server has more than one message in the message queue.

The main exception is when the processing of the message requires the server to call (directly or indirectly) `p_iowait`, for example, to perform file I/O.

In such a case you should consider whether the client really needs to use asynchronous messaging. Alternatively you can set the client's priority to be less than `0x80`. In this case the mechanism by which the window server adjusts the priority of background and foreground tasks should be disabled (see the description of `wConnect` in the *Window Server* manual).

Server functions

All the server functions except `p_minit` call `p_panic` if messages have not been initialised and `p_minit` itself calls `p_panic` if it is called a second time.

p_minit — Initialise for message reception

```
INT p_minit(INT nMess, INT lMess);
```

Initialise a queue of `nMess` message reception slots of length `lMess` and return zero if successful or one of the following negative error numbers:

`E_GEN_NOMEMORY`	there is insufficient free memory to allocate the message queue
`E_GEN_NOSEM`	there are no more free semaphores (`p_minit` creates a mutual exclusion semaphore, initialised with `nMess`)

The message length `lMess` excludes the `E_MESSAGE` structure that is at the front of all messages. The message queue is allocated as a single cell from the heap. Since this cell remains allocated for the lifetime of the process, programs calling this service should do so early in their initialisation (before any cell has been freed) to avoid heap fragmentation. All message slots are of the same size and the total size of the message queue is `nMess*(lMess+sizeof(E_MESSAGE))` bytes.

To avoid senders being blocked by message sending (quite different from waiting for a reply that can be handled asynchronously), multi-client servers should allocate a message slot for each potential client. The constant `E_MAX_PROCESSES` contains the total number of processes that can be supported by the system. In practice this can be reduced by at least four, for the null process, supervisor, file server and window server.

Only the least significant byte of `nMess` is significant, limiting the number of message slots to 255.

Calls `p_panic` if `p_minit` has already been called or if the least significant byte of `nMess` is zero.

p_mreceivew — Wait for message reception

```
VOID p_mreceivew(VOID *pMess);
```

Return when a message has been received, where the address of the message slot containing the message is written to `*pMess`.

The data in the message slot is protected from being overwritten by the receipt of another message until the message slot is freed from the queue using `p_mfree`.

Calls `p_panic` if messages have not been initialised or if an asynchronous message receive request is pending.

Example

```
typedef struct
    {
    E_MESSAGE mess;
    TEXT *bofs;
    UWORD len;
    } MESS;

    MESS *pmsg;

    p_mreceivew(&pmsg);
```

p_mreceive — Asynchronous message reception

```
VOID p_mreceive(WORD *pStatus, VOID *pMess);
```

Make an asynchronous request to receive a message and return immediately without waiting for a message to be received.

While the request is pending (and the message queue is empty), `*pStatus` contains `E_FILE_PENDING`.

When a message is received (or if there is already a received message in the queue) `*pStatus` is set to zero, the address of the message slot containing the message is written to `*pMess` and the process I/O semaphore is signalled.

The data in the message slot is protected from being overwritten by the receipt of another message until the message slot is freed from the queue using `p_mfree`.

Calls `p_panic` if messages have not been initialised or if an asynchronous message receive request is already pending.

Example

```
typedef struct
    {
    E_MESSAGE mess;
    TEXT *bofs;
    UWORD len;
    } MESS;

    MESS *pmsg;
    WORD status;

    p_mreceive(&status,&pmsg);
```

p_mcancel — Cancel a message receive request

```
VOID p_mcancel(VOID);
```

Cancel any pending asynchronous request to receive a message (which was previously requested using the p_mreceive service).

It is not considered an error to call this service if no request is pending (since the request may complete at any time). If the cancel is processed before a message is received, the associated status word *pStatus is set to E_FILE_CANCEL.

Note that this service does not cancel a pending p_msendreceivea (which is a client function). A server may support a cancel service but this is invoked, like any other service, by sending the server a message.

p_mfree — Free a message

```
VOID p_mfree(VOID *pMess, INT nReply);
```

Free the message slot with address pMess (as returned from p_mreceivew or p_mreceive) and, if the message was sent using p_msendreceivew or p_msendreceivea, complete the request by signalling the sending client's I/O semaphore and copying nReply to the client's status word (which is returned by p_msendreceivew). A negative value of nReply normally indicates an error.

If the message was sent with p_msend, the message slot is just freed and nReply is ignored.

The server should not read the contents of pMess after calling p_mfree.

If messages are not freed from the message queue then in due course sending processes will be blocked waiting on the message queue mutual exclusion semaphore.

Client functions

p_msend — Send a message

```
INT p_msend(HANDLE pid, UINT mType, VOID *pMessage);
```

Send message pMessage of type mType to process pid and return zero when the message has been deposited or the negative E_GEN_RECEIVER if pid does not exist or if process pid has not called p_minit.

The message is deposited into a free message slot in the server where the member type in the E_MESSAGE header is set to mType and the message body is copied from pMessage (where the length of the message was specified by the server as a parameter to p_minit).

If pid does not have a free message slot, p_msend waits (on a mutual exclusion semaphore) until it does.

This function is only really suitable for sending messages where the whole information fits in the message. That is, the data at pMessage should not contain the address of further data to be copied (using p_pcpyfr) because there is no way of knowing when the server will copy the data so that pMessage may be re-used. Where the message does reference further data, p_msendreceivew or p_msendreceivea should be used.

A client should certainly use p_msendreceivew or p_msendreceivea if the service is to return information (eg success or failure).

p_msendreceivew Send a message and wait for a reply

```
INT p_msendreceivew(HANDLE pid, UINT mType, VOID *pMessage);
```

Send message pMessage of type mType to process pid and wait for the server to reply (which it does by calling p_mfree(pid,nReply)) and return the reply nReply.

Return immediately with the negative E_GEN_RECEIVER if pid does not exist or if process pid has not called p_minit.

The message is deposited into a free message slot in the server where the member type in the E_MESSAGE header is set to mType and the message body is copied from pMessage (where the length of the message was specified by the server as a parameter to p_minit).

If pid does not have a free message slot, p_msendreceivew waits (on a mutual exclusion semaphore) until it does.

p_msendreceivea Asynchronous send message and get reply

```
INT p_msendreceivea(HANDLE pid, UINT mType, VOID *pMessage, WORD *pStatus);
```

Send message pMessage of type mType to process pid and asynchronously request a reply from the server. Return zero if the message was successfully sent or the negative E_GEN_RECEIVER if pid does not exist or if process pid has not called p_minit.

When the server completes the request and calls p_mfree(pid,nReply), *pStatus is set to nReply and the client's process I/O semaphore is signalled. While the reply is pending, *pStatus contains E_FILE_PENDING.

The message is deposited into a free message slot in the server where the member type in the E_MESSAGE header is set to mType and the message body is copied from pMessage (where the length of the message was specified by the server as a parameter to p_minit).

If pid does not have a free message slot, p_msendreceivea waits (on a mutual exclusion semaphore) until it does.

CHAPTER 13

GENERAL SYSTEM SERVICES

System information

p_version — Get the operating system version

```
UINT p_version(VOID);
```

Return the operating system version number.

The returned version number should be interpreted as a 4-digit hexadecimal number of the form:

```
X.YYZ
```

where `X` is the major release number, `YY` is the minor release number and `Z` is normally the hexadecimal digit `F` (internal releases use `A` and `B` to designate alpha and beta releases, respectively).

For example, a return of `0x123F` is interpreted as 1.23F.

p_romversion — Get the ROM version

```
UINT p_romversion(VOID);
```

Return the ROM version number.

The ROM contains the operating system and other system components such as, for example, the window server. The tool used to build the ROM requires a version number to be specified and this is the version number that is retrieved by this function.

See `p_version` above for the interpretation of the version number.

p_getres — Get the cause of the last system shut-down

```
INT p_getres(VOID);
```

Return a number indicating the cause of the last system shut-down, as follows:

`E_IS_A_COLD_START`	The system started up for the first time (or after a period during which all power had been removed, including the Lithium back-up).
`E_IS_A_POWERFAIL_START`	The hardware forced a shut-down because the voltage got too low. This should not happen because the system software gets a non-maskable interrupt if the voltage drops below a certain threshold (but not low enough for the hardware to force a shut-down) and this interrupt code automatically switches the hardware off. However, if the clean-up takes too long because of a poorly designed device driver, the hardware will force the machine off before the interrupt completes - in which case `p_getres` returns `E_IS_A_POWERFAIL_START`. The environment variables and the contents of `M:` are preserved.

E_IS_A_RESET_START	The machine has been reset by pressing the recessed reset button. The environment variables and the contents of M: will have been preserved (a soft reset). If the Esc key is held down while pressing the reset button, this causes a hard reset. In this case both the environment variables and the contents of M: will have been cleared. Note that this value will not occur on a Work*about*, since this machine does not have a reset button.
E_IS_A_KERNEL_FAULT	The system was reset because a serious fault occurred while executing in the operating system kernel. This could result from: (1) a bug in the operating system or a system process; (2) a program bug that managed to overcome the operating system's defences or (3) a hardware problem such as a RAM fault. The environment variables and the contents of M: will be preserved (unless the system detects a memory corruption). On the Work*about*, this value is also returned if the system is reset by pressing Psion-Ctrl-Del. This is the equivalent of a soft reset on other machine types and preserves both the environment variables and the contents of M:.
E_IS_A_NEW_OS_START	The system was reset after programming a new operating system into the Flash ROM (normally after running the *repro* program). The environment variables and the contents of M: have been cleared. On the Work*about*, this value is also returned if the system is reset by pressing Shift-Psion-Ctrl-Del. This is the equivalent of a hard reset on other machine types and clears both the environment variables and the contents of M:.

p_getosd — Get operating system data

```
VOID p_getosd(VOID *pTarget, VOID *pSource, UINT length);
```

Copy `length` bytes from offset `pSource` in the operating system data space to `pTarget`.

Operating system handles (such as memory segment handles and semaphore handles) are actually addresses in the operating system data space of the appropriate control entry.

If you `AND` a process ID with `E_PID_MASK` you get the address in operating system space of the corresponding process control entry - as described in the chapter *Processes and Inter-Process Messaging*.

p_getpsu — Get power supply type

```
INT p_getpsu(VOID);
```

Return one of the following values, defined in *epoc.h*, to indicate the power supply type on a SIBO machine:

```
E_PSU_OLD
E_PSU_MAXIM
E_PSU_S3
E_PSU_S3_A9
```

Apart from this service EPOC hides the differences between the various power supplies.

Machines in the MC GI range of SIBO computers, for example, may use either the `E_PSU_OLD` or the `E_PSU_MAXIM` power supply variants, each of which requires a slightly different variant of the EPOC operating system. The REPRO program that blows a new operating system into the Flash ROM of the MC GI range uses this service to blow the appropriate variant of EPOC.

Language and country

SIBO machines are produced in a number of language variants, differing in the following respects:

- the language code
- the language of the text used by the ROM-based software (for example, the error messages returned by `p_errs` and the month names returned by `p_nmmon`)

- character type and conversion tables as described in the chapter *Characters, Strings, Buffers and Queues*
- the keyboard layout
- the default country and country-dependent data

A particular language variant will always have a different language code and text but may not differ in all the above.

The system was designed to be produced in a variety of languages and, as far as the system services are concerned, all the above variations are encapsulated in a single configuration file in the ROM called *ROM::SYS$CTRY.CFO*. Depending on the machine, there may be further files (for example, "resource files") containing language-dependent data for higher level system components.

Unlike the language-dependent data, the country-dependent data may be altered from the language-dependent defaults.

p_getlanguage — Get the language code

```
INT p_getlanguage(VOID);
```

Return the language code from the ROM configuration file.

The language code can be used by applications that contain the text for more than one language to determine which language to present. The language codes, defined in *p_config.h*, are as follows:

```
1 = English
2 = French
3 = German
4 = Spanish
5 = Italian
6 = Swedish
7 = Danish
8 = Norwegian
9 = Finnish
10 = USA
11 = Swiss french
12 = Swiss German
13 = Portuguese
14 = Turkish
15 = Icelandic
16 = Russian
17 = Hungarian
18 = Dutch
19 = Belgian Flemish
20 = Australian
21 = New Zealand
22 = Austrian
23 = Belgian french
```

p_gettext — Get operating system text

```
INT p_gettext(INT n, TEXT *pBuffer);
```

Get the `n`th string from the ROM configuration file and write it to `pBuffer`.

Returns zero if successful or the negative `E_GEN_ARG` if `n` is outside the range of the text strings in the configuration file.

This function is called by specific text retrieval functions such as `p_errs` and `p_nmmon`.

Applications only need to use `p_gettext` when retrieving text associated with a higher level of system software (in which case the documentation of the higher level software will list appropriate values of `n`).

p_getctd — Get country-dependent data

```
VOID p_getctd(E_CONFIG *pcfg);
```

Write a copy of the system E_CONFIG struct to pcfg where the E_CONFIG struct is defined in *p_config.h* as:

```
typedef struct
    {
    UWORD countryCode;
    WORD gmtOffset;
    UBYTE dateType;
    UBYTE timeType;
    UBYTE currencySymbolPosition;
    UBYTE currencySpaceRequired;
    UBYTE currencyDecimalPlaces;
    UBYTE currencyNegativeInBrackets;
    UBYTE currencyTriadsAllowed;
    UBYTE thousandsSeparator;
    UBYTE decimalSeparator;
    UBYTE dateSeparator;
    UBYTE timeSeparator;
    UBYTE currencySymbol[9];
    UBYTE startOfWeek;
    UBYTE summerTime;
    UBYTE clockType;
    UBYTE dayAbbreviation;
    UBYTE monthAbbreviation;
    UBYTE workDays;
    UBYTE units;
    UBYTE spare[9];
    } E_CONFIG;
```

The countryCode specifies a country by its international dialling code.

See the description of p_getctd in the chapter *Time, Timers and Dates* for a description of the time-related fields: gmtOffset, dateType, timeType, dateSeparator, timeSeparator, startOfWeek, summerTime, clockType, dayAbbreviation, monthAbbreviation and workDays.

See the description of p_getctd in the chapter *Floating Point* for a description of the fields that are related to the display of floating point numbers and currency: currencySymbol, currencySymbolPosition, currencySpaceRequired, currencyDecimalPlaces, currencyNegativeInBrackets, currencyTriadsAllowed, thousandsSeparator and decimalSeparator.

p_setctd — Set country-dependent data

```
VOID p_setctd(E_CONFIG *pcfg);
```

Sets the country-dependent data from the E_CONFIG structure pointed to by pcfg.

When changing a particular field or fields you would normally:

- use p_getctd to get a copy of the E_CONFIG struct
- modify the field or fields, as required
- use p_setctd to write back the modified E_CONFIG struct

Switching on and off

By default, the auto-switch-off period is set to 300 seconds. The auto-switch-off period may be sensed and set by calling p_getauto and p_setauto respectively. Auto-switch-off when a mains adaptor is connected may be disabled by calling p_setautomains, and the corresponding state sensed by p_getautomains.

The zero priority null process automatically switches the machine off to reduce power consumption after the system has been inactive for the auto-switch-off period. In situations where the null process does not get an opportunity to run, a process can assume responsibility for allowing the machine to switch off by calling p_allowoff.

Some processes are marked as not being significant when it comes to determining what constitutes activity. For example, a continuously running clock program should not stop the system from automatically switching off in the absence of any significant activity. See p_urmarka and p_marka in the chapter *Processes and Inter-Process Messaging* for additional details on how to avoid keeping the machine switched on by continuously running applications.

p_off Switch off

```
VOID p_off(UINT uTime);
```

Switch off the machine indefinitely or for any time up to approximately 4.5 hours and return when the machine switches on.

If uTime is 0xffff the machine switches off indefinitely and will stay off until an outstanding absolute timer expires or the user switches on the machine. Equivalent to the machine automatically switching off or being switched off by the user.

If uTime is greater than 8, the machine switches off for up to uTime 1/4ths of a second (although it will still switch on if an outstanding absolute timer expires or the user switches on the machine). (If uTime is less than or equal to 8, calling p_off has no effect.)

On the IBM PC version of EPOC, calling p_off has no effect.

p_getauto Get the auto-switch-off period

```
INT p_getauto(VOID);
```

Return the current auto-switch-off period in seconds.

If the auto-switch-off period is 0xffff, the system does not automatically switch off.

p_setauto Set the auto-switch-off period

```
VOID p_setauto(INT n);
```

Set the auto-switch-off period to n seconds.

Passing an n of -1 (0xffff) stops the system from automatically switching off.

Calling p_setauto with n less than 15 is equivalent to calling p_setauto(15).

p_getautomains Get switch-off state when mains is present

```
INT p_getautomains(VOID);
```

This function is only available in EPOC version 3.18 or later.

Return TRUE if auto-switch-off is disabled when mains is present, otherwise return FALSE.

p_setautomains Disable/enable switch-off if mains is present

```
VOID p_setautomains(INT flag)
```

This function is only available in EPOC version 3.18 or later.

Disable or enable auto-switch-off if mains is present.

If flag is TRUE, auto-switch-off is disabled while mains is present and if flag is FALSE, auto-switch-off is enabled.

Even if enabled, the machine will not switch off when mains is absent if switch-off has been stopped by use of p_setauto.

p_allowoff Allow auto switch off

```
VOID p_allowoff(VOID);
```

Allow the machine to switch off if the auto-switch-off period has expired.

A compute-bound process which has marked itself as non-active (by calling p_unmarka) does not allow the machine to switch off since the null process will never get an opportunity to run. Such a process should call p_allowoff from time to time. There is no particular advantage in calling p_allowoff more frequently than at intervals of 15 seconds - the shortest auto-switch-off period.

On the IBM PC version of EPOC, calling p_allowoff has no effect.

p_setonevent — Enable/disable the ON key event

```
VOID p_setonevent(INT state)
```

This function is only available in EPOC version 2.28 or later.

Enables or disables the event that is sent to the window server when the ON key is pressed.

If state is FALSE, the event is disabled, any other value enables the event.

Power supply

This section describes functions to:

- determine the presence or absence of the main battery, the Lithium backup battery or the mains adaptor (p_supplyinfo)
- get the voltage level of the main battery (or mains adaptor, if present) and the Lithium backup battery (p_supply)
- determine whether the mains adaptor is connected (p_supply)
- get the nominal maximum voltages of the main battery and the Lithium backup battery (p_wsupply)
- get the recommended low voltage warning levels for the main battery and the Lithium backup battery (p_wsupply)
- get the time and date of insertion of the main battery, and information about main battery usage (p_supplyinfo)
- sense and set the main battery type (p_getbat and p_setbat respectively)

The main battery type is only significant on machines that can take more than one battery type (such as the MC GI range) and affects the voltages returned by p_wsupply.

The main battery types are as follows:

E_BATTERY_UNKNOWN	the battery type is initially set to this value when EPOC starts up (equivalent in its effect to E_BATTERY_ALKALINE)
E_BATTERY_ALKALINE	the battery is an Alkaline
E_BATTERY_NICAD_600	the battery is a 600 mA hour NiCd rechargeable
E_BATTERY_NICAD_1000	the battery is a 1000 mA hour NiCd rechargeable

If the hardware is unable to identify automatically the battery type, it is the responsibility of the user to set the battery type to match that actually fitted. The E_BATTERY_UNKNOWN value is intended to trigger the system into prompting the user to identify the battery type.

p_supply — Get power supply status

```
VOID p_supply(E_SUPPLY *pValue);
```

Write the status of the various supplies to the E_SUPPLY struct at pValue where E_SUPPLY is defined as:

```
typedef struct
    {
    UWORD mainBatteryReading;
    UWORD lithiumBatteryReading;
    WORD mainsPresent;
    } E_SUPPLY;
```

where:

`mainBatteryReading`	is the main battery voltage in millivolts (or the mains adaptor voltage if the mains adaptor is present)
`lithiumBatteryReading`	is the Lithium backup battery voltage in millivolts
`mainsPresent`	is negative if the mains adaptor status cannot be determined (because the SSD pack doors are open); zero if the mains adaptor is not present; one if the mains adaptor is present

p_supplyinfo — Get additional power supply data

```
VOID p_supplyinfo(E_SUPPLY_INFO *pValue);
```

This function is only available in EPOC version 3.18 or later.

Write information concerning the various power supplies to the `E_SUPPLY_INFO` struct at `pValue`.

The `E_SUPPLY_INFO` is defined in *epoc.h* as:

```
typedef struct
    {
    UBYTE mainBatteryLevel;
    UBYTE mainBatteryStatus;
    UBYTE backupBatteryLevel;
    UBYTE dcLevel;
    UWORD warningFlags;
    ULONG insertionDate;
    ULONG ticksInUseBattery;
    ULONG ticksInUseDc;
    ULONG maTicks;
    } E_SUPPLY_INFO;
```

where:

`mainBatteryLevel`	is the present status of the main battery. It describes the voltage level as being in one of four discrete states: a) `E_MBAT_GOOD` battery voltage is good b) `E_MBAT_LOW` battery voltage is low c) `E_MBAT_VERY_LOW` battery voltage is very low d) `E_MBAT_ZERO` there is no battery present! No precise figures for the actual voltage levels corresponding to these states are given.
`mainBatteryStatus`	is the same as `mainBatteryLevel` described above except that it records the *lowest* level that the battery has reached. It is reset when the main batteries are removed from their housing.
`backupBatteryLevel`	is `TRUE` if the Lithium backup battery is present and `FALSE` if the backup battery level is low or the battery is not present
`dcLevel`	is `TRUE` if the mains adaptor is present *and* powered up

warningFlags	is a set of flags which can be one of: a) E_SUPPLY_SYSTEM_TIME_CHANGED this flag only has meaning when the battery insertion date changes. If set, it means that the system time has changed; if not set, it means that the main battery has been changed. b) E_SUPPLY_SOUND_WARNING this flag is set if the battery power level is too low to operate sound. Its setting implies that the user has been warned at least once before of the failure of an attempt to generate sound. c) E_SUPPLY_FLASH_WARNING this flag is set if the battery power level is too low to operate a flash SSD. Its setting implies that the user has been warned at least once before of the failure of an attempt to write to flash. These warning flags are intended to be used only for supplying information to a user. The last two, in particular, do not necessarily imply that an attempt to generate sound or to write to flash will definitely fail. Even if either or both of these warning flags are set, the attempt may succeed - for example, either because the battery voltage has recovered since an earlier failure, or because the machine is now connected to mains power. An application is not expected to test either of these flags before attempting the corresponding operation - failures should simply be handled by standard error-recovery techniques.
insertionDate	is the system time when the present main battery was inserted
ticksInUseBattery	is the total length of time in 'ticks' (1/32 second) for which the machine has been switched on *and* powered by the main battery. Any period during this time when the machine has been powered by the mains adaptor is excluded from the total.
ticksInUseDc	is the length of time in 'ticks' (1/32 second) for which the machine has been switched on *and* powered using the mains adaptor.
maTicks	is the cumulative current delivered by the present main battery; it is a product of current x time and is measured in units of milliamps x 'ticks' where a 'tick' is 1/32 second

On machines that use the ASIC1 chip (that is, the Series 3 classic and HC) this information is not available, even if the machines contain version 3.18 or later of EPOC. In such a case, p_supplyinfo writes zero values to all members of the E_SUPPLY_INFO struct.

p_wsupply — Get battery warning and maximum levels

```
VOID p_wsupply(E_SUPPLY_WARNINGS *pValue);
```

Write the recommended low voltage warning level and the nominal maximum voltage of the main battery and the Lithium backup battery to pValue.

The E_SUPPLY_WARNINGS struct is defined as:

```
typedef struct
    {
    UWORD mainBatteryWarning;
    UWORD lithiumBatteryWarning;
    UWORD mainBatteryMax;
    UWORD lithiumBatteryMax;
    } E_SUPPLY_WARNINGS;
```

where:

mainBatteryWarning	is the recommended voltage at which to warn the user of a low main battery
lithiumBatteryWarning	is the recommended voltage at which to warn the user of a low Lithium backup battery
mainBatteryMax	is the nominal maximum voltage of the main battery
lithiumBatteryMax	is the nominal maximum voltage of the Lithium backup battery

All voltages are provided in millivolts.

If the host machine can take more than one battery type, the voltages will depend on the battery type set with p_setbat.

p_getbat — Get the battery type

```
INT p_getbat(VOID);
```

Return the current battery type.

The battery types of the form E_BATTERY_xxx are described at the beginning of this section.

On machines whose hardware does not support detection of the battery type, a call to this function should be preceded by a call to p_setbat.

p_setbat — Set the battery type

```
INT p_setbat(INT nType);
```

Set the battery type to nType and return zero if successful or E_GEN_NSUP if battery type nType is not supported by the hardware.

The battery types of the form E_BATTERY_xxx are described at the beginning of this section.

This function has no practical effect on machines, such as the Workabout, whose hardware supports the detection of the battery type.

Keyboard

p_getscancodes — Get the state of all keys

```
INT p_getscancodes(UWORD *pScan);
```

This function is only available in EPOC version 3.18 or later.

Write the current state of all the keys on the keyboard to the array of ten words pointed to by pScan.

Each key corresponds to a bit in the word array. In the case of the Series 3a, the lowest eleven bits in each of the first eight words are used. All other machines in the SIBO range use the lowest eight bits in all ten words. If a key is depressed the corresponding bit will be set, otherwise it is clear.

The mapping between keys and bits in the array varies from machine to machine. In all cases, however, if no key is depressed all ten words in the array will contain zero. The following example, to detect if any key is pressed, is suitable for use on all SIBO machines:

```
GLDEF_C IsKeyDown(VOID)
    {
    UWORD *p;
    UWORD scans[10];

    p=&scans[0];
    p_getscancodes(p);
    for (;p<=&scans[9];p++)
        {
        if (*p)
            return(TRUE);
        }
    return(FALSE);
    }
```

The mapping between keys and the bits within the array for the Series 3a keyboard is given in in the description of the EPOC HwGetScanCodes service, in the *Hardware Management* chapter of the *EPOC O/S System Services* manual.

Display

p_getlcd — Get the system display type

```
INT p_getlcd(VOID);
```

Return the (positive) system display type, where the display types are defined in *epoc.h*.

The function returns -1 if the host is a PC that has an unknown display type.

On a SIBO machine the display type is a good way of identifying the model. The appropriate constants are defined in *epoc.h*. Examples are:

E_LCD_640_400	a 640x400 pixel display as on the MC 400
E_LCD_640_200_SMALL	a 640x200 pixel display as on the MC 200
E_LCD_160_80	a 160x80 pixel display as on the HC
E_LCD_240_80	a 240x80 pixel display as on the Series 3
E_LCD_480_160	a 480x160 pixel display as on the Series 3a
E_LCD_240_100	a 240x100 pixel display as on the Workabout

On a PC, the display types are:

E_PC_HERC	Hercules graphics adaptor
E_PC_CGA	CGA graphics adaptor
E_PC_MDA	MDA display adaptor
E_PC_EGA_MONO	EGA monochrome graphics adaptor
E_PC_EGA_COLOUR	EGA colour graphics adaptor
E_PC_VGA_MONO	VGA monochrome graphics adaptor
E_PC_VGA_COLOUR	VGA colour graphics adaptor

If p_getlcd returns -1 (which can only happen on a PC), and you choose not to fail, it is recommended that you proceed as if it had returned E_PC_VGA_MONO.

p_lcdcontrastdelta — Change the LCD contrast

```
VOID p_lcdcontrastdelta(INT nDelta);
```

Step the LCD contrast up or down depending on whether nDelta is positive or negative respectively (the magnitude of nDelta is ignored.

The LCD contrast is changed through a sequence of values in a circular fashion. That is, stepping the LCD contrast up when it is already at its maximum value sets it to its minimum value and vice versa.

The sequence of values that may be set varies from model to model. On a particular model, the values can be ascertained by using p_getlcdcontrast, described below.

p_getlcdcontrast — Get the current LCD contrast

```
INT p_getlcdcontrast(VOID)
```

Return the current LCD contrast setting.

p_backlight — Switch the backlight on or off

```
INT p_backlight(INT mode);
```

Switch the backlight on or off, depending on mode as follows:

E_BACKLIGHT_OFF	switch the backlight off (if it is not already off)
E_BACKLIGHT_ON	switch the backlight on (if it is not already on) and reset the auto-switch-off timer
E_BACKLIGHT_TOGGLE	switch the backlight on and reset the auto-switch-off timer if it is currently off or switch the backlight off if it is currently on
E_BACKLIGHT_QUERY	does nothing and is used just to get the current backlight on/off state

The function always returns the on/off state (TRUE if on, FALSE if off) as it was as the function was entered.

p_setbacklight — Set the backlight control value

```
VOID p_setbacklight(UINT flag);
```

Enable or disable the backlight key and set the backlight auto-switch-off interval in ticks.

If the most significant bit of flag is set (as given by the bit mask E_BACKLIGHT_DISABLE), the operating system will not respond to the backlight key. (However, this does not stop the backlight from being switched by a program calling p_backlight.)

The lower 15 bits of flag specifies the backlight auto-switch-off interval in ticks (1/32ths of a second). If this value is zero, the backlight is not automatically switched off by a backlight timer and it will remain on until the machine switches off.

For example:

```
p_setbacklight(96);
```

sets the backlight auto-switch-off interval to 3 seconds and:

```
p_setbacklight(E_BACKLIGHT_DISABLE|(32*5));
```

sets the backlight auto-switch-off interval to 5 seconds and disables the backlight key.

p_getbacklight — Get the backlight enablement

```
UINT p_getbacklight(VOID);
```

Return the backlight control value as set by p_setbacklight, described above.

Sound

Most SIBO machines contain a piezo-electric buzzer in addition to a loudspeaker.

This section describes how to use the piezo to make a sound, and how to manipulate the set of flags used to control the sound produced by the system.

The piezo uses very little power and is an easy way of generating sound, although it is fairly quiet. Consider using the SND: device driver, which drives the loudspeaker, if greater sound complexity or a louder sound is required. The SND: device driver is described in the *I/O Devices Reference* manual.

The bit masks for the flags controlling sound output are:

E_SOUND_KEYBOARD	keyboard clicks are silenced if clear
E_SOUND_BUZZER	the piezo sound system is silenced (except for keyclicks) if clear
E_SOUND_DEVICE	the SND: device driver is silenced if clear
E_SOUND_LOUD	the piezo will sound louder if set
E_SOUND_DISABLE	all sound in the system is silenced if set

p_sound — Make a sound with the piezo

```
VOID p_sound(UINT nDuration, UINT nPitch);
```

Make a sound through the piezo for nDuration system ticks and at pitch nPitch (where the pitch has frequency 512/nPitch KHz).

Shared access to this piezo service is controlled by first waiting on a mutual exclusion semaphore that has been pre-counted with 1 (mutual exclusion semaphores are described in the chapter *Asynchronous Requests and Semaphores*). The effect of the mutual exclusion semaphore, assuming the piezo is not already in use, is that the first call to p_sound will complete immediately but subsequent calls will wait until the current operation completes. The result is that there could be an indefinite wait before the sound is made.

If nDuration is passed as a negative value the call will always complete immediately, but will fail to make a sound if the piezo is currently in use.

For example:

```
p_sound(5,320);
```

makes a short beep, suitable for accompanying an error notification. The call:

```
p_sound(-5,320);
```

will make the same sound, provided the piezo is not in use.

p_getsnd — Get the sound flags

```
INT p_getsnd(VOID);
```

Return the current setting of the sound flags.

The flags are described at the beginning of this section.

p_setsnd — Set the sound flags

```
VOID p_setsnd(INT nFlag);
```

Set the sound flags to nFlag.

The flags are described at the beginning of this section.

Sound on the Series 3a

The Series 3a machine does not contain a piezo-electric buzzer, so all sounds are made via the loudspeaker. The Series 3a operating system does, however, contain a buzzer emulator (using the SND: device driver) that supports the sound services described in the previous section.

This does, of course, mean that the p_sound service uses more power than in the case of machines that contain a piezo-electric buzzer. Since all sounds are produced via the SND: device driver, sounds that, in other machines, use the buzzer (keyclicks, for example) are disabled while the Series 3a is playing a sound, such as an alarm.

Sound files

Series 3a sound files are files that contain a 32-byte header and a byte stream of A-Law encoded digital sound. Such files normally have a *.wve* extension.

During recording 13-bit (a sign bit plus 12 magnitude bits) sound samples is converted to an 8-bit data stream at 8000 bytes per second by CODEC hardware, using A-Law encoding. During playback the byte stream is sampled at 8000 bytes per second and decoded to 13-bit sound by the CODEC.

In C, the file header is represented by the following struct (defined in *epoc.h*):

```
#define SignatureSize 16
#define ALawSignature "ALawSoundFile**"

typedef struct
    {
    TEXT Signature[SignatureSize];
    UWORD Version;
    ULONG Samples;
    UWORD SilenceInTicks;
    UWORD Repeats;
    UWORD Spare[3];
    } SndFile;
```

This header is written and read by the Series 3a sound services described below. The meanings of the items in the SndFile struct are:

Signature — the 16-byte (including the zero terminator) string "ALawSoundFile**".

`Version`	the Series 3a sound file version number as a 4-digit hexadecimal number of the form `XYYZ`, where `X` is the major release number, `YY` is the minor release number and `Z` is normally the hexadecimal digit `F`. See, for example, the description of `p_version`.
`Samples`	the number of bytes following the header. This must always be size of file less the 32 bytes for the header. Dividing this by 8000 gives the duration of the sound in seconds.
`SilenceInTicks`	the number of system ticks of silence appended to each repeat on playback (in practice, you get at least 2 ticks between repeats).
`Repeats`	the number of times to repeat the sound on playback (0 and 1 are the same).
`Spare`	reserved for future use.

A system tick is a 1/32th of a second, equivalent to 250 samples.

The program *wav2wve.exe*, supplied with the SDK (it is installed to the *\sibosdk\sys* directory) converts *.wav* sound files to the *.wve* format.

The A-Law encoding scheme

The encoding scheme compresses signed, 12-bit magnitude (13 bits in all) samples into an 8-bit representation.

A-Law encoding uses a logarithmic compression to reduce the size of sound files whilst preserving the overall sound quality. The use of a logarithmic scale ensures that the low amplitude signals (which contain most of the information in speech signals) are stored and reproduced with a minimal loss of fidelity.

The A-Law encoding and decoding schemes are illustrated in the following sections.

Encoding

If the input data is represented by a normal two's complement signed value, it must first be converted into a 12-bit magnitude, plus a 13th sign bit. For example, a value of -1 (represented by the 16-bit two's complement value of `0xffff`) must be converted to the 13-bit binary representation `(1)000000000001`, where brackets indicate the sign bit.

The following table represents the range of possible 13-bit inputs. In this table an `x` character indicates either a one or a zero (a "don't care" state) and an `s` character represents the sign bit:

s	0	0	0	0	0	0	0	a	b	c	d	x
s	0	0	0	0	0	0	1	a	b	c	d	x
s	0	0	0	0	0	1	a	b	c	d	x	x
s	0	0	0	0	1	a	b	c	d	x	x	x
s	0	0	0	1	a	b	c	d	x	x	x	x
s	0	0	1	a	b	c	d	x	x	x	x	x
s	0	1	a	b	c	d	x	x	x	x	x	x
s	1	a	b	c	d	x	x	x	x	x	x	x

A-Law compression of the above 13-bit inputs leads to the following range of 8-bit output values:

s	0	0	0	a	b	c	d
s	0	0	1	a	b	c	d
s	0	1	0	a	b	c	d
s	0	1	1	a	b	c	d
s	1	0	0	a	b	c	d
s	1	0	1	a	b	c	d
s	1	1	0	a	b	c	d
s	1	1	1	a	b	c	d

The compressed data is computed according to the following rules:

- the most significant bit preserves the sign bit of the original data item
- the following three bits represent the magnitude of the original data item, by recording the position of the most significant non-zero bit (but note that the smallest magnitude group is treated exceptionally)

- the remaining four bits record the next four most significant bits of the original data (in all but the first group, these are the four bits that *follow* the most significant non-zero bit)

The compressed data is then manipulated to invert bits 0, 2, 4 and 6. This manipulation brings the ratio of ones to zeroes closer to 50:50 and thus improves analog transmission of the data.

Examples of A-Law encoded data are given in the following table, where brackets identify the sign bit.

signed input	13-bit data	compressed data	encoded output
-1	(1)000000000001	(1)0000000	(1)1010101
+2	(0)000000000010	(0)0000001	(0)1010100
-371	(1)000101110001	(1)1000111	(1)0010010
+2074	(0)100000011010	(0)1110000	(0)0100101

A-Law encoding can be performed most simply and rapidly by means of a look-up table. Note that a 2048-element table is sufficient, since the least significant bit of the input data has no effect on the encoded value.

Where speed is not of the essence, an algorithmic method may be used, such as that illustrated by the following code:

```
#define MAGICXOR 0x2a
#define ONE  1
#define BIT8 0x80
#define MASK2 0x03
#define MASK3 0x70
#define MASK4 0x0f
#define MASK8 0xff

GLDEF_C INT Compress(INT x)
/*
Compress input 16-bit 2's complement integer (13-bit magnitude) to
8-bit signed compressed number using A-law.
*/
    {
    INT p,s,y;

    /* convert 2's complement to sign bit and magnitude */
    p=BIT8; /* p is the (inverted) sign bit */
    if (x&0x1000)
        {
        x= -x;
        x&=0xFFF;
        p=0;
        }
    if (x&(MASK4<<8)) /* Find leading '1' using binary search */
        {
        if (x&(MASK2<<10))
            s=(x&(ONE<<11)) ? 7 : 6;
        else
            s=(x&(ONE<<9)) ? 5 : 4;
        }
    else
        {
        if (x&(MASK2<<6))
            s=(x&(ONE<<7)) ? 3 : 2;
        else
            s=(x&(ONE<<5)) ? 1 : 0;
        }
    if (s==0)
        y=x>>1;
    else
        y=((x>>s)&MASK4)|(s<<4);
    return((~((y|p)^MAGICXOR))&MASK8);
    }
```

Note that the final bit manipulation is done by adding in the (inverted) sign bit, performing an exclusive or with `MAGICXOR` (`0x2a` - inverting bits 1, 3 and 5) and then complementing the result. This exactly corresponds with the formal definition of A-Law compression and also has the effect of restoring the sign bit.

It would be marginally more efficient to replace the definition of MAGICXOR with:

```
#define ALTXOR 0xD5
```

and replace the last line of the function with:

```
return(((y|p)^ALTXOR)&MASK8);
```

Decoding

Decoding an A-Law compressed data value broadly consists of reversing the process that is described in the previous section.

The bit manipulation is first reversed, by re-inverting bits 0, 2, 4 and 6. This results in the range of possible 8 bit values as follows:

s	0	0	0	a	b	c	d
s	0	0	1	a	b	c	d
s	0	1	0	a	b	c	d
s	0	1	1	a	b	c	d
s	1	0	0	a	b	c	d
s	1	0	1	a	b	c	d
s	1	1	0	a	b	c	d
s	1	1	1	a	b	c	d

Decompressing these 8 bit inputs using A-Law decoding leads to the following 13 bit outputs:

s	0	0	0	0	0	0	0	a	b	c	d	1
s	0	0	0	0	0	0	1	a	b	c	d	1
s	0	0	0	0	0	1	a	b	c	d	1	0
s	0	0	0	0	1	a	b	c	d	1	0	0
s	0	0	0	1	a	b	c	d	1	0	0	0
s	0	0	1	a	b	c	d	1	0	0	0	0
s	0	1	a	b	c	d	1	0	0	0	0	0
s	1	a	b	c	d	1	0	0	0	0	0	0

The decompressed data is computed according to the following rules:

- the most significant bit preserves the sign bit of the compressed data item
- the following three bits of the compressed data are used to determine the position of the most significant non-zero bit of the decompressed data (but note that the smallest magnitude group is treated exceptionally)
- the remaining four bits of the compressed data are used as the next four most significant bits of the decompressed data (in all but the first group, these are the four bits that *follow* the most significant non-zero bit)
- the next most significant bit of the decompressed data is set to one and any remaining trailing bits are set to zero. The resultant value is thus set to the mean of all the possible values which, on compression, give the value that is being decoded

If the output data is required in 16-bit two's complement format, then the sign bit must be extracted from the data and, if it is set, the decoded value should be negated.

Examples of decoding are given in the following table, where brackets identify the sign bit.

encoded data	compressed data	13-bit data	signed output
(1)1010101	(1)0000000	(1)000000000001	-1
(0)1010100	(0)0000001	(0)000000000011	+3
(1)0010010	(1)1000111	(1)000101111000	-376
(0)0100101	(0)1110000	(0)100001000000	+2112

The starting values in this table are the results from the encoding examples given in the previous section. Notice that compression followed by decompression loses some of the information, as is expected.

As for encoding, A-Law decoding can be performed by means of a look-up table. In this case, a 256-element table is sufficient. It is, however, simple to perform the decoding algorithmically, as illustrated in the following code:

```
#define XORMASK 0x55
#define BIT8 0x80
#define MASK3 0x70
#define MASK4 0x0f
#define MASK8 0xff

GLDEF_C INT ALawDecode(INT x)
/*
Expand input 8-bit signed compressed number to 16-bit
2's complement integer (13-bit magnitude) using A-law.
*/
    {
    INT s,y;

    x=(x^XORMASK)&MASK8;
    s=(x&MASK3)>>4;
    y=0;
    if (s)
        {
        y=0x10;
        s-=1;
        }
    y=(((y+(x&MASK4))<<1)+1)<<s;
    if (x&BIT8)
        y= -y;
    return (y);
    }
```

Series 3a sound system services

p_recordsounda — Record a sound asynchronously

```
VOID p_recordsounda(TEXT *name, UINT len, WORD *stat);
```

This function is only available in EPOC version 3.18 or later.

Asynchronously record sound to a file.

The parameter `name` points to a zero terminated string. This should be the filename of the file to which sound is to be recorded (any existing file is replaced). The service will fail if the file is on a Flash SSD.

The value of `len` specifies, in units of 2048 bytes, the maximum number of bytes to be recorded. This figure excludes the 32-byte header. Before recording starts, a file of length `32+len*2048` bytes is created (and there must actually be room on the disk for a file of that length).

While recording is taking place, the word pointed to by `stat` contains `E_FILE_PENDING`. On completion of recording, the completion status is written to `*stat`. The completion status will be zero if recording completed successfully, `E_FILE_CANCEL` if recording was terminated by a call to `p_recordsoundcancel`, or a (negative) error number. See the chapter *Asynchronous Requests and Semaphores* for a general description of asynchronous services.

The service will fail with the error `E_GEN_FAIL` if sound is disabled. You can only record to *M:* or to a RAM SSD. Although you can not record to a Flash SSD, you can copy a recorded file to a Flash SSD and play it back from there.

p_recordsoundcancel — Cancel sound recording

```
VOID p_recordsoundcancel(VOID);
```

This function is only available in EPOC version 3.18 or later.

Cancel the recording of sound that was initiated by `p_recordsounda`. This service is functionally identical to `p_playsoundcancel` - the two services may be used interchangeably.

The sound file is truncated to the actual length that was recorded before recording was cancelled. The completion status of `p_recordsounda` will be `E_FILE_CANCEL`.

p_recordsoundw — Record a sound synchronously

```
INT p_recordsoundw(TEXT *name, UINT len);
```

This function is only available in EPOC version 3.18 or later.

Synchronously record sound to a file.

The parameter `name` points to a zero terminated string. This should be the filename of the file to which sound is to be recorded (any existing file is replaced). The service will fail if the file is on a Flash SSD.

The value of `len` specifies, in units of 2048 bytes, the maximum number of bytes to be recorded. This figure excludes the 32-byte header. Before recording starts, a file of length `32+len*2048` bytes is created (and there must actually be room on the disk for a file of that length).

The function returns when the recording is complete, and returns the completion status. This will be zero if the recording completed successfully, otherwise it is a (negative) error number.

The service will fail with the error `E_GEN_FAIL` if sound is disabled. You can only record to *M:* or to a RAM SSD. Although you can not record to a Flash SSD, you can copy a recorded file to a Flash SSD and play it back from there.

p_playsounda — Play back a sound asynchronously

```
VOID p_playsounda(TEXT *name, UINT duration, UINT volume, WORD *stat);
```

This function is only available in EPOC version 3.18 or later.

The parameter `name` points to a zero terminated string. This should be either the file specification of the sound file to be played, or a * followed by just the name component of the sound file. If the string starts with a *, the extension *.WVE* is assumed and the service automatically hunts *ROM::* and the *\wve* directories of *M:*, *A:* and *B:* (in that order). Note that the Series 3a *ROM::* sound files have names *sys$al01.wve*, *sys$al02.wve* ...

The time that the sound file will play, in system ticks, is specified by `duration`. If this is shorter than the natural duration of the specified sound file then playback is truncated. If `duration` is negative, in addition to truncating longer files, short files are padded with trailing silence to the specified duration. If `duration` is zero, the file is played without truncation or padding. Note that the natural duration includes any trailing silence and number of repeats that are specified in the file header.

(The Series 3a alarm server passes a parameter of -480 to truncate or pad out to 15 seconds.)

The loudness of playback is determined by `volume`, which may be a number between 0 and 5 inclusive, with 0 being the loudest. On the Series 3a there are only four actual volume levels: 1, 2, 3 and 4. Setting a level of 0 has the same effect as setting level 1, and setting a level of 5 has the same effect as setting level 4.

While playback is taking place, the word pointed to by `stat` contains `E_FILE_PENDING`. On completion of playback, the completion status is written to `*stat`. The completion status will be zero if playback completed successfully, `E_FILE_CANCEL` if playback was terminated by a call to `p_playsoundcancel`, or a (negative) error number. See the chapter *Asynchronous Requests and Semaphores* for a general description of asynchronous services.

The service will fail with the error `E_GEN_FAIL` if sound is disabled.

p_playsoundcancel — Cancel sound playback

```
VOID p_playsoundcancel(VOID);
```

This function is only available in EPOC version 3.18 or later.

Cancel the playback of sound that was initiated by `p_playsounda`. This service is functionally identical to `p_recordsoundcancel` - the two services may be used interchangeably.

After a call to `p_playsoundcancel` the completion status of `p_playsounda` will be `E_FILE_CANCEL`.

p_playsoundw — Play back a sound synchronously

```
INT p_playsoundw(TEXT *name, UINT duration, UINT volume);
```

This function is only available in EPOC version 3.18 or later.

The parameter `name` points to a zero terminated string. This should be either the file specification of the sound file to be played, or a * followed by just the name component of the sound file. If the string starts with a *, the extension *.WVE* is assumed and the service automatically hunts *ROM::* and the *\wve* directories of *M:*, *A:* and *B:* (in that order). Note that the Series 3a *ROM::* sound files have names *sys$al01.wve*, *sys$al02.wve* ...

The time that the sound file will play, in system ticks, is specified by `duration`. If this is shorter than the natural duration of the specified sound file then playback is truncated. If `duration` is negative, in addition to truncating longer files, short files are padded with trailing silence to the specified duration. If `duration` is zero, the file is played without truncation or padding. Note that the natural duration includes any trailing silence and number of repeats that are specified in the file header.

(The Series 3a alarm server passes a parameter of -480 to truncate or pad out to 15 seconds.)

The loudness of playback is determined by `volume`, which may be a number between 0 and 5 inclusive, with 0 being the loudest. On the Series 3a there are only four actual volume levels: 1, 2, 3 and 4. Setting a level of 0 has the same effect as setting level 1, and setting a level of 5 has the same effect as setting level 4.

The function returns when playback is complete, and returns the completion status. This will be zero if the playback completed successfully, otherwise it is a (negative) error number.

The service will fail with the error `E_GEN_FAIL` if sound is disabled.

Exit the system

p_hwexit — Exit to DOS

```
VOID p_hwexit(VOID);
```

Exit EPOC and return to DOS (only effective on the IBM PC version of EPOC).

Calling `p_hwexit` has no effect on a SIBO version of EPOC.

CHAPTER 14

DATABASE FILES

Overview of database files

Database files (DBFs) 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 database files. The data files created and manipulated by OPL are also examples of database files.

DBFs are designed to be *Flash-friendly*, that is, they may be stored and manipulated in Flash SSDs (or any other EPROM medium). A DBF 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.

A freshly formatted SSD has, apart from a short header, all bytes set to 0xff, that is, all bits are set to one. Writing to an SSD consists of selectively clearing bits to zero. On a Flash SSD it is not possible (except by reformatting the whole SSD) to overwrite a zero with a one. Data can be overwritten, provided the new value can be derived from the old one solely by clearing bits to zero. DBFs take advantage of this fact by reserving record type zero to represent a deleted record.

This has a number of implications for DBFs. The most fundamental is that deleting a record does not reduce the size of the file, since all that happens is that the record type is overwritten with zero. A DBF containing deleted records can be reduced in size by:

- calling `DbfCompress`, provided the file is stored on a compressible medium
- using the `DbfCopyFile` service to copy the file record by record, since deleted records will not be copied.

Furthermore, updating a record can only be performed by deleting the original record and appending the modified version. Thus, updating a record must always move it to the end of the file.

The DBF service functions described in this chapter enhance access to database files by providing:

- an option to access records via a sparse index, with an index entry for every sixteenth record
- read-ahead buffering with, typically, a 4k buffer
- optimised searching, to locate a record by content.

The optional index table resides in a separate segment (with segment name *DBF$nnnn.INX*, where *nnnn* is a 4 digit hexadecimal number derived from the open DBF channel number) so as not to use any of the application's data space. It enables fast access to a record by its record number and allows for fast backwards scanning of the file. The index table consists of a 4 byte address for every sixteenth record. Addresses are appended to the table as necessary when records are added to the file. Deleting a record causes the addresses to be adjusted as necessary so that they continue to point to every sixteenth record.

A read call to the file server from a DBF service will fill the read-ahead buffer, typically reading many records. This reduces the number of separate calls to the file server during a sequential scan of the file and increases the speed of operation of many of the DBF services.

In general, DBF services may overwrite the buffer contents. The next read of a DBF record following a modification of the buffer contents will cause the entire buffer to be read in again, inevitably resulting in a loss of performance. Since this process assumes a knowledge of the buffer contents, it is essential that all modifications to the buffer contents are either performed via DBF services or are accompanied by a call to `DbfTrash`.

In this respect it is worth noting that the services in the following list are guaranteed not to alter the buffer:

```
DbfFlush
DbfVersion
DbfAppend
DbfSense
DbfCount
```

The file header

Database files start with a 22 byte standard header containing the following information:

Byte offset in header	Information
0 - 15	Zero terminated file signature.
16, 17	Version of DBF software used to produce the file.
18, 19	Offset from the start of the file to the first record.
20, 21	Minimum version of DBF software required.

All 16 bytes of the file signature are used for verification, not just the zero terminated string. It is therefore essential that all file signatures fill the whole 16 bytes. If necessary, you should pad out the signature string with trailing zeros.

See the `DbfVersion` service for the format of the version numbers.

The file offset to the first record permits the use of an extended header, with additional application-specific information following the standard header. If an extended header is not used, the value should be 22.

Records

The records are of variable length, with the type and length contained in a leading word header. In memory a record occupies a `DbfRecord` struct, defined in *p_dbf.h* as:

```
typedef struct
    {
    UWORD header;     /* Used for record header word */
    UBYTE data[2];    /* Data to be written...       */
    } DbfRecord;
```

The most significant four bits of `header` contain the record type, in the range 0 to 15 (0xf). The remaining twelve bits store the record length. DBF records are restricted to a maximum length of 4094 bytes, which is one byte less than the theoretical maximum of 4095 (0xfff) bytes.

The record types are classified as follows:

0	*Deleted record*. These records are ignored by all DBF services. In particular, they are never copied by the `DbfCopyFile` service.
1	*Standard data record*, containing a number of fields corresponding to the field sequence specified by the field information record, described below. Most DBFs will contain, apart from deleted records, only type 1 records, one field information record and, optionally, one descriptive record (described below).
2	*Field information record*, used to store the field structure used by other records. There must be a field information record in each file and it must be the first record in the file. Any subsequent type 2 records will be ignored. The content of this record is described below.

3	*Descriptive record*. A DBF may optionally contain a record of this type, containing file-wide application-specific data (such as the screen font to use). The content of such a record consists of one or more variable length sub-records, with a word header containing the type and length, exactly as for the main records. The sub-record types are specific to the creating application. There is further information about descriptive records in the descriptions of the `DbfDescRecordRead` and `DbfDescRecordWrite` services.
4 - 7	*Application-specific records* that are copied to a new file, but not appended to an existing file by the `DbfCopyFile` service.
8 - 13	*Application-specific records* that are both copied to a new file and appended to an existing file by the `DbfCopyFile` service.
14	*Reserved* for voice records, containing information that is generated and interpreted by a voice device driver.
15	*Reserved* for internal use - not to be used by applications.

The field information record contains up to 32 bytes, each indicating the type of the corresponding field in the data records (it follows that a data record may contain a maximum of 32 fields, but there is an exception, described later). The possible values for each byte are:

0	Word
1	Long
2	Double
3	String
4 - 255	Reserved

The file opening services open a DBF in such a way that only one record type (usually type 1) is visible to the DBF services. There is no requirement for all record types to conform to the structure specified in the field information record, but it is expected that, for normal use, type 1 records will do so. The only service that assumes the record structure matches the content of the field information record is `DbfFindRead`.

Records are not restricted to contain the same number of fields as listed in the field information record. They may contain fewer fields, provided that only trailing fields are omitted. If a record contains more than the number of fields specified in the field information record, it is assumed that the additional ones are string fields.

A record that contains only string fields is not restricted by the normal maximum of 32 fields; it may contain any number of fields, subject to the overall 4094 byte limit on the record length.

An application that uses several record types may:

- open the file for one record type at a time, closing the file and reopening it to access records of a different type
- open the file for one record type and handle the reading and writing of records of other types independently of the DBF services

String fields

String fields contain leading byte counted text, and thus a normal string field may not contain more than 255 characters.

However, longer strings may be stored by making use of *continuation sub-fields*. In such a case, the first 254 characters of the string and a terminating byte of value `0x14` are stored in an otherwise normal string field, with a count byte containing the value 255. The terminating `0x14` character, coupled with a length byte of 255, indicates that further string characters are contained in an immediately following string field. This following field is considered as a continuation sub-field of the previous one.

The same mechanism may be used in a continuation sub-field to extend the string text into a further continuation sub-field. Subject to the overall restriction that a record may not exceed 4094 bytes, there is thus no limit on the length of text that may be stored in a single database string field.

Number of records

The DBF services are restricted to files containing a maximum of 65534 records, numbered from 0 to 65533. Since only one record type is visible via the DBF services, a DBF may contain more than this maximum, provided there are not more than 65534 records of any one type.

A file containing more than the maximum number of (visible) records will, on opening, be logically truncated to contain the maximum number of records.

End of file record

When any of the record services attempts to read past the end of the file, the error `E_FILE_EOF` will be returned. The current record number (the record number returned by `DbfSense`) will then be the number of the last record plus 1. This (fictitious) record is known as the end of file record.

Attempting to read before the first record in the file, with either the `DbfBackRead` service or the `DbfFindRead` service, will also result in an `E_FILE_EOF` error. In this case the current record number will be zero. This will normally refer to the first record in the file, but may, if the file contains no records, refer to the end of file record.

If the file contains no records, `DbfSense` will always return zero, again referring to the end of file record.

At any time that the current record number refers to the end of file record, those services that operate on the current record, such as `DbfEraseRead` or `DbfUpdate`, will do nothing to the record and return `E_FILE_EOF`.

Database files and OPL

OPL data files, created and manipulated by the OPL data file commands, are database files. They are created with a file signature string of `"OPLDatabaseFile"` and contain, in addition to the leading field information record, only standard data (type 1) records.

OPL can open and manipulate database files created by other applications, with the following restrictions:

- The file must have a file signature string of `"OPLDatabaseFile"`
- Records other than the leading field information record and standard data (type 1) records are ignored by OPL

DBF functions

DbfOpen — Open a database file

```
INT DbfOpen(INT *pstate, VOID **pFcb, TEXT *fName, UINT mode, DbfHeader *pHead, UBYTE *pbuffer,
            UINT len, UINT type);
```

Open a channel to the database file specified by the zero terminated file specification `fName` and, if successful, return zero and write the channel to `*pFcb` (`pFcb` is not written to if the open fails).

See also `DbfQuickOpen`.

The file specification `fName` is parsed with a `NULL` related name (see `p_fparse`). If this fails, the open fails and returns the return value from `p_fparse`.

The mode in which the file is opened is selected by `mode`, which should contain one (and only one) of:

```
P_FOPEN
P_FCREATE
P_FREPLACE
P_FAPPEND
P_FUNIQUE
```

optionally `OR`ed with one of:

```
P_FUPDATE
P_FSHARE
```

For the meanings of these flags, see the description of `p_open(P_FSTREAM)` in the *Files* chapter. The DBF services make no distinction between files opened with either `P_FOPEN` or `P_FAPPEND`. All other required

mode flags are supplied automatically. Opening a DBF with mode equal to P_FUNIQUE will write the unique name to fName.

The value of *pstate may be one of:

DbfStateDisabled	Opens the file with a sparse index. The file is open and the index is fully built when the call to DbfOpen returns, but the call may take an extended time to return.
DbfStateOpenNoIndex	Opens the file without an index. The call to DbfOpen returns much faster than for the previous case. Not all DBF services may be used on a file opened without an index. See the descriptions of the individual services for further details.
DbfStateStart	Opens the file with a sparse index. The open process may not be complete when the call to DbfOpen returns, depending on the value written back to *pstate. DbfOpen must be called repeatedly, passing the value written to *pstate by the previous call to DbfOpen, until the value written to *pstate is DbfStateStart. This should be used in cases (such as the need to remain responsive to user input) where an extended time to return is unacceptable.

The parameter pHead is a pointer to a DbfHeader struct, defined in *p_dbf.h* as:

```
typedef struct
    {
    UBYTE fileType[DbfHeaderNameSize]; /* 16 byte file signature */
    UWORD createVersion;     /* software version used to create file */
    UWORD dataStart;         /* offset in file of first record */
    UWORD needVersion;       /* minimum software version needed to handle this file */
    UWORD firHeader;
    UBYTE fir[DbfMaxFirLength];
    } DbfHeader;
```

When creating a new file or replacing an existing file, all elements of this struct should be pre-filled with the header and field information record data described earlier. Note that there is no gap between the header and the field information record, even if an extended header is required. The file itself, however, will contain a gap for the extended header, the length of which is 22 bytes less than the value in the dataStart field.

When opening an existing file, the fileType field must be pre-filled with the file signature. The remainder of the DbfHeader struct will be filled in with the relevant data read from the file. All 16 bytes of the file signature will be verified against the signature in the file and E_FILE_INVALID is returned if the two signatures are not identical.

The following checks are made in all cases, regardless of whether the information is provided by the user or read from an existing file.

- The needVersion field is checked against the DBF software version number (returned by the DbfVersion service). The major version number in needVersion must not exceed the current DBF software major version number (see the DbfVersion service for the format of version numbers). It is the application's responsibility to perform any further validation of the version number.
- The field information record data is checked to be the correct type and of a length not exceeding the maximum length (32). An E_FILE_INVALID error is returned if any of these checks fail.

The address and length of a user-supplied read-ahead buffer are passed in pbuffer and len respectively. Each read call to the file server from a DBF service will read len bytes into this buffer. In general, the buffer will contain more than one record. A DBF read service to access a record that, as a result of an earlier read, is already in the buffer will simply locate the record within the buffer.

The buffer length, in len, must be in the range 512 to 16384. Any value outside this range will cause DbfOpen to fail with an E_FILE_RECORD error.

In addition, the buffer should be at least as large as the largest record in the file. Opening a file with a sparse index and a buffer which is smaller than the largest record will cause DbfOpen to fail with an E_FILE_RECORD error (this error will not be reported when opening a DBF without an index). A buffer of 4096 bytes is guaranteed to be sufficient for all database files.

Only records of type equal to type are visible. In most cases, type will be 1 but could, exceptionally, be in the range 4 to 14 inclusive. No check is made on the value of type, but the results of opening a file will be unpredictable if type is 0, 2, 3 or greater than 14.

Immediately after opening the file, the current record number (as returned by DbfSense) will be 0, so a call to DbfNextRead would read record number 1 and DbfEraseRead would erase record 0. To read record 0 you should call DbfFirstRead.

No error is returned if the opened file contains more than the maximum number (65534) of records of any one type. The DBF services will treat such a file as if it contained the maximum number of records.

Apart from the errors explicitly mentioned above, DbfOpen may fail with any of the errors returned by p_open(P_FSTREAM), p_seek or p_read.

DbfQuickOpen — Open a database file

```
INT DbfQuickOpen(INT *pstate, DbfOpenArgs *pargs, UBYTE *pbuffer, UINT len, UINT type);
```

Open a channel to a database file, as for DbfOpen, except that a number of the parameters are passed in a DbfOpenArgs struct, defined in *p_dbf.h* as:

```
typedef struct
    {
    VOID **pFcb;
    UBYTE *fName;
    UINT mode;
    DbfHeader *pHead;
    } DbfOpenArgs;
```

The meanings of the struct members and the remaining parameters are exactly as described for DbfOpen.

DbfQuickOpen returns zero if successful, otherwise it returns errors as for DbfOpen.

DbfQuickOpen should be used in preference to DbfOpen since it provides more efficient and shorter code. The DbfOpen service is retained for compatibility reasons.

DbfClose — Close a database file

```
INT DbfClose(VOID *pFcb);
```

Close a database file, returning zero for success (the negative error returns are as for p_close). The file channel is closed, even if DbfClose returns an error.

May be used on a DBF opened without an index.

Calls p_panic if pFcb is not a pointer to valid DBF file channel data, generated by an earlier call to DbfOpen or DbfQuickOpen.

DbfFlush — Flush a database file

```
INT DbfFlush(VOID *pFcb);
```

Flush all buffers, ensuring that all modified data is written to the DBF.

Returns zero for success (the negative error returns are as for p_write).

May be used on a DBF opened without an index.

Calls p_panic if pFcb is not a pointer to valid DBF file channel data, generated by an earlier call to DbfOpen or DbfQuickOpen.

DbfTrash — Notify that the DBF buffer has been overwritten

```
VOID DbfTrash(VOID *pFcb);
```

Inform the DBF services that the buffer of the file corresponding to the channel data in pFcb has been overwritten by the caller and that the contents of the buffer can not be relied on. See also DbfCopyDown.

May be used on a DBF opened without an index.

Calls p_panic if pFcb is not a pointer to valid DBF file channel data, generated by an earlier call to DbfOpen or DbfQuickOpen.

DbfCopyDown — Copy down a DBF record

```
INT DbfCopyDown(VOID *pFcb, UINT offset);
```

Copy a record at `offset` in the DBF buffer to the start of the buffer and sets a flag to signal that the buffer is no longer valid (i.e. there is no need to call `DbfTrash`). The length of the record is read from the buffer and is returned by the service.

It is assumed that `offset` is the position in the buffer of a valid record, given by an earlier call to `DbfAbsRead`, `DbfAbsReadSense`, `DbfNextRead`, `DbfBackRead`, `DbfFirstRead`, `DbfLastRead`, `DbfEraseRead` or `DbfFindRead`. The results will be unpredictable if this is not the case, or if the caller has written to the buffer since making one of the above calls.

May be used on a DBF opened without an index.

Calls `p_panic` if `pFcb` is not a pointer to valid DBF file channel data, generated by an earlier call to `DbfOpen` or `DbfQuickOpen`.

DbfCompress — Compress a database file

```
INT DbfCompress(UINT *pstate, VOID *pFcb);
```

Recover space used by deleted records (provided the file is stored on a compressible medium), returning zero for success. If the medium is not compressible, this service will do nothing but will still return zero.

After calling this service, the current record will be the end of file record (unless the medium was not compressible - in which case the current record is unchanged).

The value of `*pstate` may be one of:

`DbfStateDisabled`	The file compression is complete when the call to `DbfCompress` returns, but the call may take an extended time to return.
`DbfStateStart`	The file compression may not be complete when the call to `DbfCompress` returns, depending on the value written back to `*pstate`. `DbfCompress` must be called repeatedly, passing the value written to `*pstate` by the previous call to `DbfCompress`, until the value written to `*pstate` is `DbfStateStart`. This should be used in cases (such as the need to remain responsive to user input) where an extended time to return is unacceptable.

Should not be used on a file opened without an index.

Calls `p_panic` if `pFcb` is not a pointer to valid DBF file channel data, generated by an earlier call to `DbfOpen` or `DbfQuickOpen`.

DbfCopyFile — Copy a database file

```
INT DbfCopyFile(UINT *pstate, VOID *pFcb, TEXT *pTargetName, UINT targetMode, UINT type, INT dir);
```

Copy records (except deleted records) in either direction between the current file (specified by `pFcb`) and the file named in `pTargetName`, returning zero for success. This service may be used either to copy records to a new file or to append records to an existing file.

The value of `*pstate` may be one of:

`DbfStateDisabled`	The copy is complete when the call to `DbfCopyFile` returns, but the call may take an extended time to return.
`DbfStateStart`	The copy may not be complete when the call to `DbfCopyFile` returns, depending on the value written back to `*pstate`. `DbfCopyFile` must be called repeatedly, passing the value written to `*pstate` by the previous call to `DbfCopyFile`, until the value written to `*pstate` is `DbfStateStart`. This should be used in cases (such as the need to remain responsive to user input) where an extended time to return is unacceptable. An estimate of the number of calls required to complete the copy is to divide the source file size by the buffer size and add 2.
`DbfStateCopyAbort`	aborts a copy that was started with the state `DbfStateStart`.

The mode in which `pTargetName` is opened is specified by `targetMode`, with the same options as for `DbfOpen`. If `targetMode` is `P_FUNIQUE`, the unique file name is written to `pTargetName`. In all cases, this file is opened without an index.

The direction of the copy is determined by `dir`:

`DbfCopyFromHandle` copies records from the current file to the file specified by `pTargetName`. To copy the current file, `targetMode` should be `P_FCREATE`, `P_FREPLACE` or `P_FUNIQUE`. If appending records to an existing file, targetMode should be `P_FOPEN` or `P_FAPPEND`.

`DbfCopyToHandle` appends records from the file specified by `pTargetName` into the current file. In this case `targetMode` can sensibly only be `P_FOPEN`.

Appending records to the current file may be slower than a copy or append from the current file because of the need to update the index (if it exists).

Which records are copied is determined by `type`. This may specify either a single type (normally type 1) or (by passing the value `DbfRecordTypeAll`) all record types. There are special cases that depend on the nature of the copy:

- when copying to a new file (`targetMode` is `P_FCREATE`, `P_FREPLACE` or `P_FUNIQUE`) the field information (type 2) record is always copied to the new file, regardless of the value of `type`.
- when appending records to an existing file (`targetMode` is `P_FOPEN` or `P_FAPPEND`) records of type 2 to 7 inclusive (which therefore includes the field information record and the descriptive record) are never copied, regardless of the value of `type`.

If the copy is to a new file (`targetMode` is `P_FCREATE`, `P_FREPLACE` or `P_FUNIQUE`) the file header (including any extended header) is copied to the new file. If any error occurs during the copy the target file will be deleted, if possible.

If the copy appends records to an existing file (`targetMode` is `P_FOPEN` or `P_FAPPEND`) the signatures of the two files are verified and the field information records are checked to be compatible (either identical, or both containing only string fields). If either test fails the call to `DbfCopyFile` will return `E_FILE_INVALID`.

May be used on a DBF opened without an index.

Calls `p_panic` if `pFcb` is not a pointer to valid DBF file channel data, generated by an earlier call to `DbfOpen` or `DbfQuickOpen`. In addition to errors explicitly mentioned above, `DbfCopyFile` error returns are as for `p_open`, `p_read` and `p_write`.

WARNING: using `DbfCopyFile` to append records can result in a file containing more than 65534 records of a particular type. No error is given if this occurs.

DbfFileSize — Find the size of a database file

```
INT DbfFileSize(VOID *pFcb, ULONG *pSize);
```

Writes the size of an open database file to `*pSize`, returning zero for success.

May be used on a DBF opened without an index.

Calls `p_panic` if `pFcb` is not a pointer to valid DBF file channel data, generated by an earlier call to `DbfOpen` or `DbfQuickOpen`. Error returns are as for `p_seek`.

DbfExtHeaderRead — Read a DBF extended header

```
INT DbfExtHeaderRead(UINT cont, VOID *pFcb, VOID *buf, UINT len);
```

Read up to `len` bytes from the extended header of a database file and write the data to `buf`, returning the actual number of bytes read.

If there are fewer than `len` bytes left before the end of the extended header, the number of remaining bytes are read and returned. If the current position is already at the end of the extended header the negative number `E_FILE_EOF` is returned. All other error returns are as for `p_read` and `p_seek`.

A long extended header may be read in sections. A value of 0 for `cont` signifies an initial read of the extended header and resets the current file position to the start of the extended header before reading. For subsequent reads, `cont` should be set to 1. If the whole extended header is read in a single call to `DbfExtHeaderRead`, `cont` must be set to 0.

May be used on a DBF opened without an index.

Calls `p_panic` if `pFcb` is not a pointer to valid DBF file channel data, generated by an earlier call to `DbfOpen` or `DbfQuickOpen`.

DbfExtHeaderWrite — Write a DBF extended header

```
INT DbfExtHeaderWrite(UINT cont, VOID *pFcb, VOID *buf, UINT len);
```

Write up to `len` bytes of data from `buf` into the extended header, returning the actual number of bytes written.

If there are fewer than `len` bytes left before the end of the extended header, the number of remaining bytes are written and returned. If the current position is already at the end of the extended header the negative number `E_FILE_EOF` is returned. All other error returns are as for `p_write` and `p_seek`.

A long extended header may be written in sections. A value of 0 for `cont` signifies an initial write of the extended header and resets the current file position to the start of the extended header before writing. For subsequent writes, `cont` should be set to 1. If the whole extended header is written in a single call to `DbfExtHeaderWrite`, `cont` must be set to 0.

May be used on a DBF opened without an index.

Calls `p_panic` if `pFcb` is not a pointer to valid DBF file channel data, generated by an earlier call to `DbfOpen` or `DbfQuickOpen`.

DbfDescRecordRead — Read a DBF descriptive record

```
INT DbfDescRecordRead(VOID *pFcb);
```

Read the descriptive record to offset zero in the file's read-ahead buffer.

The descriptive record contains variable length sub-records, in the same format as main records, with the record types being defined by the creating application. The reader should ignore (and not delete) unrecognised sub-record types.

Returns the length of the descriptive record, if it exists, otherwise `E_FILE_EOF`.

Returns `E_FILE_INVALID` if the file was opened without an index.

Calls `p_panic` if `pFcb` is not a pointer to valid DBF file channel data, generated by an earlier call to `DbfOpen` or `DbfQuickOpen`. Other error returns are as for `p_seek` and `p_read`.

DbfDescRecordWrite — Write a DBF descriptive record

```
INT DbfDescRecordWrite(VOID *pFcb, UINT len);
```

Write a descriptive record from the data at offset zero in the file's read-ahead buffer, returning zero for success.

The data in the buffer be a `DbfRecord` structure, that is the record content must start at offset 2. The first two bytes are used to construct the header for the record (see `DbfAppend`). These two bytes should not be included in `len`, the length of the record.

Any existing descriptive record will be erased, so that there is no more than one descriptive record per file. The application should ensure that any unrecognised sub-record types are preserved from any previously existing descriptive record. If `len` is passed as zero, any existing descriptive record will be erased and no new one will be written out.

Returns `E_FILE_INVALID` if the file was opened without an index.

Calls `p_panic` if `pFcb` is not a pointer to valid DBF file channel data, generated by an earlier call to `DbfOpen` or `DbfQuickOpen`. Other error returns are as for `p_seek`, `p_read` and `p_write`.

DbfVersion — Get the DBF version number

```
UINT DbfVersion(VOID);
```

Return the version number of the DBF software. This will be a hexadecimal number in the form `XYYF` where:

`X`	is the major version number (4 bits)
`YY`	is the minor version number (8 bits)
`F`	is the release type, either A,B or F for Alpha, Beta or Final respectively (4 bits).

For example, if 110FH is returned, the DBF software version is 1.10F.

Note that only the major version number is used to determine whether or not the DBF file system can handle a particular file.

May be used on a DBF opened without an index.

DbfAbsRead — Read a specific DBF record

```
INT DbfAbsRead(VOID *pFcb, UINT recnum, UWORD *pOffset);
```

Seek to and read (into the read-ahead buffer) record number `recnum`, returning the length of the record or a negative error.

Record `recnum` becomes the current record and the offset of the record within the read-ahead buffer is written to `*pOffset`. This offset indicates the start of a `DbfRecord` struct (including the header) but the returned record length is the length of the record data, excluding the header.

If `recnum` is greater than the last record, the error `E_FILE_EOF` is returned, the current record is set to the end of file record and `*pOffset` is not valid.

May be used on a DBF opened without an index.

Calls `p_panic` if `pFcb` is not a pointer to valid DBF file channel data, generated by an earlier call to `DbfOpen` or `DbfQuickOpen`. Other error returns are as for `p_seek` and `p_read`.

DbfAbsReadSense — Read and sense a specific DBF record

```
INT DbfAbsReadSense(VOID *pFcb, UINT recnum, UWORD *pOffset, ULONG *pPos);
```

Seek to and read (into the read-ahead buffer) record number `recnum`, returning the length of the record or a negative error.

Record `recnum` becomes the current record and the offset of the record within the read-ahead buffer is written to `*pOffset`. This offset indicates the start of a `DbfRecord` struct (including the header) but the returned record length is the length of the record data, excluding the header. The file position of the header at the start of the record is written to `*pPos`.

If `recnum` is greater than the last record, the error `E_FILE_EOF` is returned, the current record is set to the end of file record and `*pOffset` is not valid.

May be used on a DBF opened without an index.

Calls `p_panic` if `pFcb` is not a pointer to valid DBF file channel data, generated by an earlier call to `DbfOpen` or `DbfQuickOpen`. Other error returns are as for `p_seek` and `p_read`.

DbfNextRead — Read the next DBF record

```
INT DbfNextRead(VOID *pFcb, UWORD *pOffset);
```

Seek to and read (into the read-ahead buffer) the next record, returning the length of the record or a negative error.

This record becomes the current record and the offset of the record within the read-ahead buffer is written to `*pOffset`. This offset indicates the start of a `DbfRecord` struct (including the header) but the returned record length is the length of the record data, excluding the header.

If the current record is already the last record, or the file contains no records of the current type, the error `E_FILE_EOF` is returned, the current record is set to the end of file record and `*pOffset` is not valid.

May be used on a DBF opened without an index.

Calls `p_panic` if `pFcb` is not a pointer to valid DBF file channel data, generated by an earlier call to `DbfOpen` or `DbfQuickOpen`. Other error returns are as for `p_seek` and `p_read`.

DbfBackRead — Read the previous DBF record

```
INT DbfBackRead(VOID *pFcb, UWORD *pOffset);
```

Seek to and read (into the read-ahead buffer) the previous record, returning the length of the record or a negative error.

This record becomes the current record and the offset of the record within the read-ahead buffer is written to `*pOffset`. This offset indicates the start of a `DbfRecord` struct (including the header) but the returned record length is the length of the record data, excluding the header.

If the current record is already the first record, or the file contains no records of the current type, the error `E_FILE_EOF` is returned, the current record is set to record number 0 (which may be the end of file record) and `*pOffset` is not valid.

May be used on a DBF opened without an index.

Calls `p_panic` if `pFcb` is not a pointer to valid DBF file channel data, generated by an earlier call to `DbfOpen` or `DbfQuickOpen`. Other error returns are as for `p_seek` and `p_read`.

DbfFirstRead — Read the first DBF record

```
INT DbfFirstRead(VOID *pFcb, UWORD *pOffset);
```

Seek to and read (into the read-ahead buffer) the first record, returning the length of the record or a negative error.

This record becomes the current record and the offset of the record within the read-ahead buffer is written to `*pOffset`. This offset indicates the start of a `DbfRecord` struct (including the header) but the returned record length is the length of the record data, excluding the header.

If the file contains no records of the current type, the error `E_FILE_EOF` is returned, the current record is set to record number 0 (which is the end of file record) and `*pOffset` is not valid.

May be used on a DBF opened without an index.

Calls `p_panic` if `pFcb` is not a pointer to valid DBF file channel data, generated by an earlier call to `DbfOpen` or `DbfQuickOpen`. Other error returns are as for `p_seek` and `p_read`.

DbfLastRead — Read the last DBF record

```
INT DbfLastRead(VOID *pFcb, UWORD *pOffset);
```

Seek to and read (into the read-ahead buffer) the last record, returning the length of the record or a negative error.

This record becomes the current record and the offset of the record within the read-ahead buffer is written to `*pOffset`. This offset indicates the start of a `DbfRecord` struct (including the header) but the returned record length is the length of the record data, excluding the header.

If the file contains no records of the current type, the error `E_FILE_EOF` is returned, the current record is set to record number 0 (which is the end of file record) and `*pOffset` is not valid.

Should not be used on a DBF opened without an index.

Calls `p_panic` if `pFcb` is not a pointer to valid DBF file channel data, generated by an earlier call to `DbfOpen` or `DbfQuickOpen`. Other error returns are as for `p_seek` and `p_read`.

DbfAppend — Append a DBF record

```
INT DbfAppend(VOID *pFcb, UINT len);
```

Append a record of the current type, and of length `len`, to the end of the file and make this the current record, returning zero for success.

The record to be appended must be stored at the start of the read-ahead buffer as a `DbfRecord` struct (defined in *p_dbf.h*) including the leading two byte `header`. `DbfAppend` uses these two bytes to construct the type and length header for the record. This header should not be included in `len`, which is the length of the data only.

The error `E_GEN_OVER` is returned if there are already 65534 records of the current type in the file. If the total length of the record (including the two byte header) is greater than the length of the read-ahead buffer then `E_FILE_RECORD` is returned. Other error returns are as for `p_seek` and `p_write`.

Should not be used on a DBF opened without an index.

Calls `p_panic` if `pFcb` is not a pointer to valid DBF file channel data, generated by an earlier call to `DbfOpen` or `DbfQuickOpen`.

DbfEraseRead — Erase a DBF record

```
INT DbfEraseRead(INT *pstate, VOID *pFcb, UWORD *pOffset);
```

Erase the current record and read (into the read-ahead buffer) the following record, returning the length of the record read, or a negative error.

This record becomes the current record and the offset of the record within the read-ahead buffer is written to `*pOffset`. This offset indicates the start of a `DbfRecord` struct (including the header) but the returned record length is the length of the record data, excluding the header.

There are two separate circumstances in which `DbfEraseRead` may return `E_FILE_EOF`:

- if the current record is already the "end of file" record (or there are no records)
- if the current record is the last record.

In the first case, the service does nothing. In the second case, the last record is erased and the current record becomes the end of file record.

It is the application's responsibility to distinguish, if necessary, between these two cases. This may be done either by checking if the current record is the "end of file" record (using `DbfSense` and `DbfCount`) before calling `DbfEraseRead`, or by using `DbfCount` before and after the call to determine if the record count has decreased.

The value of `*pstate` may be one of:

`DbfStateDisabled`	The erase and read process is complete when the call returns, but the call may take an extended time to return.
`DbfStateStart`	The erase and read process may not be complete when the call returns, depending on the value written back to `*pstate`. `DbfEraseRead` must be called repeatedly, passing the value written to `*pstate` by the previous call to `DbfEraseRead`, until the value written to `*pstate` is `DbfStateStart`. This should be used in cases (such as the need to remain responsive to user input) where an extended time to return is unacceptable.

Should not be used on a DBF opened without an index.

Calls `p_panic` if `pFcb` is not a pointer to valid DBF file channel data, generated by an earlier call to `DbfOpen` or `DbfQuickOpen`.

DbfUpdate — Update a DBF record

```
INT DbfUpdate(INT *pstate, VOID *pFcb, UINT len);
```

Erase the current record and append a new record, of length `len`, from the read-ahead buffer, making this the current record.

Returns zero for success, or a negative error.

The record to be appended must be stored at the start of the read-ahead buffer as a `DbfRecord` struct (that is, including a leading two bytes). `DbfAppend` uses these two bytes to construct the type and length header for the record. This header should not be included in `len`, which is the length of the data only.

Note that the current record is not erased until after the new record has been successfully appended.

`DbfUpdate` will do nothing and return `E_FILE_EOF` if there are no records of the current type, or if the current record is the end of file record.

The value of `*pstate` may be one of:

`DbfStateDisabled`	The update is complete when the call returns, but the call may take an extended time to return.
`DbfStateStart`	The update may not be complete when the call returns, depending on the value written back to `*pstate`. `DbfUpdate` must be called repeatedly, passing the value written to `*pstate` by the previous call to `DbfUpdate`, until the value written to `*pstate` is `DbfStateStart`. This should be used in cases (such as the need to remain responsive to user input) where an extended time to return is unacceptable.

Should not be used on a DBF opened without an index.

Calls p_panic if pFcb is not a pointer to valid DBF file channel data, generated by an earlier call to DbfOpen or DbfQuickOpen.

DbfFindReadField — Find a DBF record

```
INT DbfFindReadField(UINT *pstate, VOID *pFcb, VOID *pBuffer, UINT len, UINT findMode,
                     UINT nStrings, UWORD *pOffset, UINT startStr);
```

This function is only available in EPOC version 3.18 or later.

Match the wildcard text at pBuffer and of length len (which must not exceed 255 bytes), in the nStrings string fields starting at string field number startStr (a value of zero for startStr starts matching at the first string field). The match is attempted on each record, starting at the current record. Returns the length of the first record found to contain a match or, if no match is found, a negative error.

This service must only be used on records which conform with the content of the field information record. No error is reported if any particular record has fewer than nStrings string fields. If nStrings has the value DbfFindAllStrings the search for a match will continue through all string fields until the end of the record. The field information record is used to determine the types of up to the first 32 fields. Fields in excess of those defined in the field information record are assumed to be string fields.

findMode specifies the type of search and is made up of three parts which must be ORed together:

The maximum length over which the match is made in any one string field is passed in findMode. String fields are effectively truncated to this length before matching. The maximum length that may be specified is 255, implying no truncation.

The starting point and direction of the search is specified by ORing one of the following into findMode:

DbfFindForwards	Search forwards from current record to next match.
DbfFindBackwards	Search backwards from current record to previous match.
DbfFindFirst	Search to first match in file.
DbfFindLast	Search to last match in file.

The type of match that is made is specified by ORing either of the following into findMode:

DbfFindCaseIndependent	Case independent match.
DbfFindCaseDependent	Case dependent match.

If a match is found, the record containing the match becomes the current record and the offset of the record within the read-ahead buffer is written to *pOffset. This offset indicates the start of a DbfRecord struct (including the header) but the returned record length is the length of the record data, excluding the header.

If a match is not found, E_FILE_EOF is returned and *pOffset is no longer valid. The current record will then be either the first record if the search was backwards, or the end of file record if the search was forwards.

The value of *pstate may be one of:

DbfStateDisabled	The find is complete when the call returns, but the call may take an extended time to return.
DbfStateStart	The find may not be complete when the call returns, depending on the value written back to *pstate. DbfFindRead must be called repeatedly, passing the value written to *pstate by the previous call to DbfFindRead, until the value written to *pstate is DbfStateStart. This should be used in cases (such as the need to remain responsive to user input) where an extended time to return is unacceptable.

May be used on a DBF opened without an index, except for a findMode that specifies DbfFindLast, for which the result is unpredictable.

Calls p_panic if findMode is improperly constructed, or if pFcb is not a pointer to valid DBF file channel data, generated by an earlier call to DbfOpen or DbfQuickOpen. Other error returns are as for p_seek and p_read.

Finding across continuation sub-fields

As described above, a search using DbfFindReadField will not locate text that is contained, in whole or in part, in a continuation sub-field.

If it is possible that a database may contain string fields more than 255 characters in length, the values of the len and findMode parameters must be modified to force the search to extend into continuation sub-fields.

Firstly, the value 0x1400 must be ORed into len (whose unmodified value cannot exceed 255).

Secondly, the value of findMode must be constructed as follows:

- the length component of findMode, representing the maximum length over which the match is made in any one string field, must be set to 255
- the value 0x4000 must be ORed into findMode
- only case-independent matching is allowed, so findMode must be ORed with DbfFindCaseIndependent
- the starting point and direction of the search is specified, as before, by ORing one of DbfFindForwards, DbfFindBackwards, DbfFindFirst or DbfFindLast into findMode

DbfFindRead — Find a DBF record

```
INT DbfFindRead(UINT *pstate, VOID *pFcb, VOID *pBuffer, UINT len, UINT findMode,
                UINT nStrings, UWORD *pOffset);
```

Match the wildcard text at pBuffer and of length len (which must not exceed 255 bytes), with the first nStrings string fields of each record, starting at the current record. Returns the length of the record containing a match, if found, or a negative error.

Calling DbfDindRead is equivalent to calling DbfFindReadField with startStr set to zero.

This service must only be used on records which conform with the content of the field information record. No error is reported if any particular record has fewer than nStrings string fields. If nStrings has the value DbfFindAllStrings the search for a match will continue through all string fields until the end of the record. The field information record is used to determine the types of up to the first 32 fields. Fields in excess of those defined in the field information record are assumed to be string fields.

findMode specifies the type of search and is made up of three parts which must be OR'ed together:

The maximum length over which the match is made in any one string field is passed in findMode. String fields are effectively truncated to this length before matching. The maximum length that may be specified is 255, implying no truncation.

The starting point and direction of the search is specified by ORing one of the following into findMode:

DbfFindForwards	Search forwards from current record to next match.
DbfFindBackwards	Search backwards from current record to previous match.
DbfFindFirst	Search to first match in file.
DbfFindLast	Search to last match in file.

The type of match that is made is specified by ORing either of the following into findMode:

DbfFindCaseIndependent	Case independent match.
DbfFindCaseDependent	Case dependent match.

If a match is found, the record containing the match becomes the current record and the offset of the record within the read-ahead buffer is written to *pOffset. This offset indicates the start of a DbfRecord struct (including the header) but the returned record length is the length of the record data, excluding the header.

If a match is not found, E_FILE_EOF is returned and *pOffset is no longer valid. The current record will then be either the first record if the search was backwards, or the end of file record if the search was forwards.

The value of *pstate may be one of:

DbfStateDisabled	The find is complete when the call returns, but the call may take an extended time to return.
DbfStateStart	The find may not be complete when the call returns, depending on the value written back to *pstate. DbfFindRead must be called repeatedly, passing the value written to *pstate by the previous call to DbfFindRead, until the value written to *pstate is DbfStateStart. This should be used in cases (such as the need to remain responsive to user input) where an extended time to return is unacceptable.

May be used on a DBF opened without an index, except for a findMode that specifies DbfFindLast, for which the result is unpredictable.

Calls p_panic if findMode is improperly constructed, or if pFcb is not a pointer to valid DBF file channel data, generated by an earlier call to DbfOpen or DbfQuickOpen. Other error returns are as for p_seek and p_read.

Finding across continuation sub-fields

As described above, a search using DbfFindRead will not locate text that is contained, in whole or in part, in a continuation sub-field.

If it is possible that a database may contain string fields more than 255 characters in length, the values of the len and findMode parameters must be modified to force the search to extend into continuation sub-fields.

Firstly, the value 0x1400 must be ORed into len (whose unmodified value cannot exceed 255).

Secondly, the value of findMode must be constructed as follows:

- the length component of findMode, representing the maximum length over which the match is made in any one string field, must be set to 255
- the value 0x4000 must be ORed into findMode
- only case-independent matching is allowed, so findMode must be ORed with DbfFindCaseIndependent
- the starting point and direction of the search is specified, as before, by ORing one of DbfFindForwards, DbfFindBackwards, DbfFindFirst or DbfFindLast into findMode

DbfSense — Sense the current DBF record number

```
UINT DbfSense(VOID *pFcb);
```

Return the record number of the current record.

This will be the record number of the end of file record (0 if there are no records, otherwise the number of records plus one) if an immediately preceding DBF service call returned an E_FILE_EOF error.

May be used on a DBF opened without an index.

Calls p_panic if pFcb is not a pointer to valid DBF file channel data, generated by an earlier call to DbfOpen or DbfQuickOpen.

DbfCount — Count the number of DBF records

```
UINT DbfCount(VOID *pFcb);
```

Return the number of records of the currently visible type, without altering the current record number.

Should not be used on a DBF opened without an index.

Calls p_panic if pFcb is not a pointer to valid DBF file channel data, generated by an earlier call to DbfOpen or DbfQuickOpen.

CHAPTER 15

OBJECT ORIENTED PROGRAMMING

This chapter contains a complete reference description of EPOC's run-time support for object oriented programming (OOP). It does not describe the essential concepts of OOP, the compile-time tools used to build applications using OOP, or any system-supplied object class libraries.

This chapter is not suitable as an introduction to the object oriented programming environment on any SIBO machine.

Classes

A *class* is implemented as:

- a class descriptor (a data structure that resides in a code segment)
- a set of functions (normally written in C) that implement the class *methods*

The class descriptor resides in the same code segment as its associated method functions.

Class descriptor

The C structure for the class descriptor (which may be of use after calling `p_cpycat` to copy a class descriptor to the data segment) is defined as:

```
typedef
    {
    UWORD cat;
    struct p_class *super; /* superclass class */
    UWORD len; /* length of instance */
    UWORD base; /* base function number */
    UBYTE sig_6b; /* signature - should be 0x6b */
    UBYTE num; /* number of entries in vector table */
    UBYTE ncomp; /* number of component objects */
    } P_CLASS;
```

where the contents of a loaded and dynamically linked class descriptor is as follows:

`cat`	the handle of the code segment that contains the superclass class descriptor
`super`	the offset of the superclass class descriptor within segment `cat` or zero if this is a root class (which has no superclass)
`len`	the length of an instance of the class, including the lengths of inherited property (used by eg `p_new` and `p_newlibh` to create an instance)
`base`	the base method number corresponding to the first entry in the method table that follows the class descriptor
`num`	the number of entries in the method table
`ncomp`	the number of component objects to be automatically destroyed
`sig_6b`	a signature (which should be `0x6b`) to guard against a bad class reference

This header is followed by an array of num 16-bit code segment offsets (into the same segment that contains the descriptor) of the method functions. This table contains "holes" (that is, method numbers for which there is no method function) represented by zeros.

In the unlinked structure (for example, when the category descriptor resides in an executable file or a DYL), the first two fields cat and super, which identify the superclass, contain different values. These values are overwritten when the segment is loaded and dynamically linked. This is described below.

Object instances

An *instance* of a class is implemented as a cell in the heap and is created by calling p_new, f_new, f_newlibh, p_newlibh, f_newsend or f_newlibhsend.

The first two words of the cell contain the code segment handle and class descriptor offset of the class of which that object is an instance (in exactly the same way as for the superclass reference in a linked class descriptor).

The remainder of the cell contains the *property* (if any) of that class, including any property inherited from its superclasses.

A subclass contribution to the property comes after its immediate superclass contribution such that (given the restriction of single inheritance) the offset to a property contribution of a class is the same for that class as it is for any subclass of that class.

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

Object destruction

An object is normally destroyed by sending it a message of message number zero (sending a message is described later).

A root class (ie a class that has no superclass) normally contains a single method (corresponding to message number zero). This implements the default destroy method and is inherited by all other objects.

The default destroy method is designed to destroy the object and all its components (and the components of components and so on) as defined by the ncomp field in the class descriptor.

The destroy method function, root_destroy, is normally provided from the PLIB library.

Before calling p_free to free the heap cell that represents the object instance, root_destroy scans from the youngest class to the oldest looking for non-zero ncomp values in the corresponding class descriptors. If it finds a non-zero ncomp, it assumes that the property segment contributed by that class begins with an array of ncomp object addresses or NULLs (by convention, a NULL entry indicates an object that has either not yet been created or has already been destroyed). Each non-NULL entry is sent a zero destroy message.

Since an object is always initially zero-filled, a failure in a partially constructed compound object will have NULLs in all the right places and a single destroy method to the owner should perform the appropriate partial destroy.

Object classes which create resources that are not objects (eg an I/O channel, which needs to be closed) or which wish to destroy objects in a specific order, generally subclass the destroy method to clean up the resources introduced by the class in addition to "supersending" the destroy message to the superclass to continue the process.

Categories

Categories package a collection of classes into load modules and, when loaded, code segments.

There are two types of categories:

Image categories	which contain an entry point at offset zero and are used to implement programs. The name of a code segment that contains an image category has the extension .$SC. An image category code segment is created by loading an executable using p_execc (as described in the chapter *Processes and Inter-Process* messaging).
Dynamic library categories (DYLs)	which contain no entry point, but contain classes that are referenced from image categories and other DYLs. The name of a code segment containing a DYL has the extension .DYL. A DYL code segment is created by loading a DYL load module (which may be a separate file or a partition of an executable) using p_loadlib or p_loadfilelib (as described in this chapter).

The ROM typically contains a number (depending on the machine) of loaded and linked category code segments.

Category code segments are shared - there is only one copy of a particular category in memory, however many processes are executing it.

Once loaded into RAM and dynamically linked to external categories on which it depends, a category memory segment is read-only - as is any code segment. A process that accidentally tries to write to a code segment is panicked with panic number 60.

A straightforward small to medium sized application typically consists of a single image category that references the built-in ROM DYLs.

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

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

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

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

Category handles

A category handle identifies a category code segment, which may be in RAM or the ROM, as follows:

- if the category handle is positive, it is the handle of a moveable RAM-based code segment
- if the category handle is negative, it is the paragraph address of a ROM category code segment

Category numbers

A category code segment may also be identified by a category *number* which is known at compile time (category *handles* are only known at run time).

The local category (that is, the category containing the code that makes the category reference) always has the category number zero.

An external category number is the index (from 1) into an array of external category handles in the local category code segment.

A category number is mainly used to create an instance of an object class using `p_new`, `f_new` or `f_newsend` although it is also used by the more obscure functions `p_exactsend`, `p_reclass` and `p_cpycat`.

The value of an external category number depends on the composition and (arbitrary) order of the external category array in the local category and different categories will, in general, use different category numbers to refer to the same external category. Because of this fact, a category number should not be passed as a parameter to an external method (for example, to create a component of variable class). When there is a requirement to pass a category as a parameter, the category handle rather than the category number should be used. The category handle may always be obtained from the category number by calling `p_getlibh`.

Dynamic linkage

A reference to an external category by category number occurs when:

- a class from an external category is subclassed by the local category
- the local category contains code that references an external category by a category number (most likely to create an instance of an external class using `p_new`, `f_new` or `f_newsend` although it could also contain calls to `p_exactsend`, `p_reclass` and `p_cpycat`).

Before such references may be made, the category must be dynamically[1] linked to the external categories it references by category number.

The main image category is linked by calling `p_linklib(0)` and DYLs are linked either by the function that loads the DYL (either `p_loadlib` or `p_loadfilelib`) or subsequently (for reasons discussed below) by calling `p_linklib`.

External categories are referenced by their memory segment names and it follows that, when a category is dynamically linked, **all the referenced categories must be loaded**.

A category is dynamically linked shortly after loading it. Between loading and linking, it may be necessary to load other referenced DYLs.

For the predominant case where an application is implemented as a single image category referencing only ROM-based DYLs (which are already loaded), the image may be linked by calling `p_linklib` at any time (normally early in `main`).

When the application image category loads a DYL which references only those categories that are already loaded (for example, the ROM-based DYLs and the loading image category), the function that loads the DYL (either `p_loadlib` or `p_loadfilelib`) may be passed a parameter value that causes the function to link the DYL immediately after loading.

Referencing by category handle

It is possible for a category to reference an external category by category *handle* - most likely to create an instance of an external class using `p_newlibh`, `f_newlibh` or `f_newlibhsend` or to call the more obscure `p_reclassbyhandle`.

In this case, the handle is normally obtained independently of dynamic linkage by one of the following means:

- the category handle is passed as a parameter to a method
- from the segment name by calling `p_findlib` (emulating dynamic linkage)
- because the local category loaded the DYL using `p_loadlib` or `p_loadfilelib`

A category (whether an image category or a DYL) may reference any number of external categories (which may also be images or a DYLs). Typically, the following cases occur:

- an application image category references one or more DYLs (especially ROM-based DYLs)
- a DYL references another DYL
- an application-specific DYL references the application image category

Although technically possible, the case of an image category referencing another image category is unlikely to be useful.

DYLs

Like the main image category, the main DYL contains code that may be shared by multiple processes.

Also like the main category, DYLs are produced independently of any other category by a (static) linker.

In practice, DYLs are of one of the following types:

- genuine library DYLs, used by multiple applications (for example, the ROM-based DYLs)
- application-specific DYLs supplementing the application image category and containing classes which could, in principle, equally well be resident in the image category
- DYLs conforming to a common interface for use by one or more applications - for example to implement a number of different file transfer protocols with a common interface

DYLs that are used by more than one application should not access static data other than the *reserved static variables* that are allocated at the beginning of the data segment of all applications (the structure of the data segment is discussed in the chapter *Memory Allocation*).

---

[1] The term *dynamic linkage* is used because the link is made at run time - as opposed to normal (static) linkage between code modules, which occurs at compile time (and is used to produce a category load module - for example, an executable).

For example, the ROM-based DYL *OLIB.DYL* uses the reserved static:

```
GLREF_D VOID *w_am;
```

holding the address of the one and only instance of the application manager object, which schedules the running of multiple active objects (as described in the *OLIB Reference* manual).

Application-specific DYLs may use the seven static variables that are reserved for application programs (in the sense that the system either does not use them or it restores them if it does). These reserved statics (which are initialised by the system to zero) are:

```
GLREF_D VOID *DatApp1;
GLREF_D VOID *DatApp2;
GLREF_D VOID *DatApp3;
GLREF_D VOID *DatApp4;
GLREF_D VOID *DatApp5;
GLREF_D VOID *DatApp6;
GLREF_D VOID *DatApp7;
```

These names may be `#define`'d to a more descriptive name, depending on the usage, as in, for example:

```
#define PageLayout DatApp1
```

Note that the code in DYLs may not introduce static variables by using quoted strings in C. For example, the code:

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

is fine in an image category but can't be used in a DYL. DYLs tend to have code fragments such as:

```
WORD b[3];

b[0]=('T'<<8)+'I'; b[1]=('M'<<8)+':'; b[2]=0;
p_open(&tcb,(TEXT *)&b[0],-1);
```

which, although hard to read, does at least produce efficient code.

When writing application-specific DYLs it is technically possible to use static variables, provided all such variables are declared in a single module, included both in the link of the application image category and the DYL (analogous to FORTRAN COMMON blocks). At the time of writing, we were taking the view that this is a dangerous practice since the accidental introduction of static data - say a quoted string - would misalign the variables in the two links. The tool to build a DYL from the output of the linker fails if it detects any declared static data other than the reserved statics.

Category load modules

An image category is loaded from an executable by another process calling `p_execc` - as described in the chapter *Processes and Inter-Process Messaging*. The executable may have the extension *.IMG* or *.APP* but the loaded segment always has the extension *.$SC*.

The image category is not automatically dynamically linked by `p_execc` because the category might reference DYLs which must first be loaded. After loading any such DYLs, the program calls `p_linklib(0)` to link itself.

A DYL category may be loaded from one of two sources:

- from a dynamic library file (which normally has the extension *.DYL*) using `p_loadlib`
- from a file containing multiple DYLs (normally an executable with the extension *.APP*) using `p_loadfilelib` after having previously opened the file using `p_openlib`

In the former case, the DYL is identified by its file specification. This is appropriate for genuine library DYLs and replaceable DYLs such as, for example, a printer driver DYLs.

In the latter case, the DYL is identified by a number that indexes the DYLs embedded in the executable. This is appropriate for application-specific DYLs.

A DYL may either be linked by the function that loads the DYL (either `p_loadlib` or `p_loadfilelib`) or subsequently (if further referenced DYLs need to be loaded first) by calling `p_linklib`.

The structure of a loaded and linked category

A loaded category code segment contains the following:

- an external category table containing the segment handles of all externally referenced categories (to convert external category numbers into category handles)
- a class table containing the segment offsets of each class descriptor (to convert class numbers into the segment offset of the corresponding class descriptor)
- a class descriptor for each class
- the method functions and other local and global functions

Typically, the bulk of the code segment is filled with functions - like any other code segment.

The segment offset of the external category table is stored at offset 8 in the segment where the category table consists of:

- a word containing the number of entries in the following array ORed with 0x8000
- the array of external category handles

The segment offset of the class lookup table is stored at offset 6. The class lookup table is immediately followed by the external category table so the length of the class lookup table may be obtained from the difference between the values at offset 8 and 6.

In an image category, the entry point is at address zero.

The word at offset 4 in an image code segment contains the address in the data segment of the beginning of the uninitialised static variables (and the end of the initialised static variables).

The structure of an unlinked category

The unlinked category differs from the linked category in the following respects:

- there is an external category *name* table instead of an external category *handle* table
- the superclass category references in the class descriptors are by category number (zero for the local category)
- the external superclass class references in the class descriptors are by class number (references to classes in the local category are by segment offset)

The external category name table consists of the following:

- a word containing the number of entries in the following array (but not ORed with 0x8000)
- the array of external category names

What happens during dynamic linkage

Dynamic linkage consists of the following:

- converting the external category *name* table into the external category *handle* table (and ORing the number of entries word with 0x8000 to indicate that this conversion has taken place)
- converting local superclass references (which have category number zero) to the handle of the local segment
- using the external category table to convert external superclass references (which have category number greater than zero) to category handles and also to convert the class number to a segment offset (using the class table in the external segment)

Message passing

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

Method functions are called by their *method number*, which must be between zero and 255. The method number zero is normally reserved for the method that destroys the object (and its components, if any).

When called, the method function is passed the address of the object instance (a cell in the heap, as returned by say p_new or p_newlibh) as its first parameter with zero to three additional parameters, depending on the method.

The most common way of sending a message is to use p_send, which does the following:

- locate the class descriptor of which that object is an instance (using the category handle and class segment offset at the beginning of the instance)
- if the method number is in range of the method table that follows the class descriptor, and the corresponding entry has a non-zero value in it, call the corresponding method function
- otherwise locate the superclass class descriptor and repeat the above

If the process of trying to find a corresponding method in successive superclass class descriptors (sometimes called *superclass chaining*) fails, the sending function panics with panic number 48.

The send will also panic (with panic number 55) if the category handle and class segment offset at the beginning of the instance points to a class descriptor that does not have the correct signature. This catches, amongst other things, the sending of a message to an object that has already been destroyed.

As well as p_send, which is most commonly used within method functions to send a message to an object, there is:

p_supersend	which is used within a method function to send a message of the same method number to the same object but to be handled by a superclass method. It works like p_send except that the search for a method starts at the immediate superclass of the class associated with the method containing the call to p_supersend. It is typically used within a subclass method that adds further processing (before, after or around the call to p_supersend) to the method being replaced.
p_entersend	which works like a p_send that has been called with a p_enter - but more efficiently
p_exactsend	which can send a message to any method of any class (ignoring the category handle and class segment offset at the beginning of the instance). In OOP, it is normally used within a method function to send a message of the same method number to the same object but to be handled by a superclass method once removed - in effect a super supersend.

The message sending functions (p_send etc) represent the only mechanism for calling methods when:

- the method is *polymorphic* (where a particular send may call different method functions depending on the class of the instance to which the method is being sent)
- the method function is in an external category (for example, a ROM-based DYL)
- the method function contains a call to p_supersend

When writing the method functions of application-specific classes, a message send can be implemented as regular (near) function call when:

- the target method function is in the same category as the sending method
- the target method is *monomorphic*
- there are no calls to p_supersend in the target method

If the target monomorphic method function is in the same category as *all* actual and prospective sending methods, there is no reason to even include the address of the method function in the method table (which appears after the class descriptor).

When writing general-purpose library DYLs, one has to be more careful about calling a local method (rather than using a message sending function such as p_send) because direct calling removes any opportunity for subclassers to divert the send to a subclass method. However, in some cases it may be positively desirable to restrict subclassers.

Calling conventions for method functions

A method function that is the target of any of the message sending functions (eg p_send, p_supersend or p_entersend) must use one of the following two calling conventions:

CDECL	where the generated code will take the parameters off the stack

METHOD_CALL where the generated code will take the parameters from the registers (which is more efficient)

Note that if you call a method function directly, the prototype must be present and indicate the correct calling convention.

Recall from the *Error Handling* chapter that the target of a p_enter must use one of:

CDECL where the generated code will take the parameters off the stack

ENTER_CALL where the generated code will take the parameters from the registers

Since the ENTER_CALL convention is different from the METHOD_CALL convention, a method function that is a target of both p_send (or any other message sending function, including p_entersend) and p_enter must be declared as CDECL.

Performance of message sending

Message sending using p_send takes longer than a direct call for the following reasons:

- there is a far call via an 8086 software interrupt to get to the message sending code in the ROM
- it has to find the class and index a method table to find the address of the function to call (possibly more than once if the method is found after superclass chaining)
- it performs complex stack manipulation to take the parameters from the call to p_send to pass to the method (note that it removes the method number)
- it has to handle the fact that the target method may be in a moveable code segment (as well as the return to the caller which is also, in general, in a moveable code segment)

The following table of timings (in seconds) for a million calls to various library functions was produced from a test program running on an MC400 running at 8MHz.

```
Empty loop..................  8
p_dummy..................... 13
p_osdummy................... 24
p_isprint('A')..............  44
p_slen("")..................  74
p_send2..................... 134
p_send5..................... 145
p_send2 (subclass)..........  152
p_ioyield................... 165
p_iosignal+p_iowait.........  230
```

The following code segment indicates the basic mechanism that was used to obtain the above numbers.

```
LOCAL_D ULONG c1=1000000L;

GLDEF_C VOID Time(VOID (*call)(VOID), TEXT *name)
    {
    ULONG t1,t2;

    p_print("%-.30s",name);
    p_sleep(5L); /* to allow redraws to complete */
    t1=p_date();
    (*call)();
    t2=p_date();
    p_printf("%4ld",t2-t1);
    }

GLDEF_C VOID z_dummy(VOID)
    {
    ULONG c;

    for (c=c1;c--;p_dummy());
    }
```

where main contained calls of the form:

```
    Time(z_dummy,"p_dummy");
```

The function p_dummy consists only of a return. The 13 second result includes the 8 seconds for the ULONG loop overhead to call the function a million times.

The function p_osdummy calls the minimal ROM interrupt service, which just returns. The additional 11 seconds over the time for p_dummy represents the overhead of making a far call to the ROM and handling the return to a moveable code segment.

The test for p_iosignal also includes a call to p_iowait. The functions p_isprint, p_slen, p_ioyield, p_iosignal and p_iowait are all described in this manual. Note that the time taken by a p_ioyield or a p_iowait will depend on what wait handlers are installed.

The tests for message sending all send to a method that just returns. The relative difference between the time for p_send2 (which has no additional parameters) and the time for p_send5 (which has the maximum of three additional parameters) shows that there is little overhead to passing more parameters.

The test labelled p_send2 (subclass) shows the time taken for one iteration of superclass chaining. Here, the message was sent (with no additional parameters) to a instance of a class that relies on its superclass to provide the method.

These results show that message sending has about 20 times the overhead of a local function call. The results also show that calling any ROM-based service via a software interrupt has an overhead of 3 to 5 times that of a call to a local function.

At several thousand sends per second, the additional overhead of message sending is only going to degrade performance when it occurs in the innermost loops of an application. Where performance is key[2], speed critical sections of code should clearly avoid using p_send or any other far call to ROM code.

The main benefit of object oriented programming is in promoting well-designed programs. Since a well-designed program is presumably understood by its author, there should be no difficulty in identifying the critical code sections that need to be written efficiently (and where the avoidance of far calls is just one factor contributing to that efficiency).

It is most certainly possible to produce poorly-designed programs using object oriented programming and the above timings show that a program which bumbles along will go a lot slower using p_send rather than direct function calls.

An assertion along the lines of "look after the pennies and the pounds look after themselves" is a good rule provided it is not taken too far - it should not ruin the design and make future maintenance a nightmare. It should certainly not be the only basis on which performance is delivered, nor should it be considered as an alternative to understanding the program.

Using, where possible, direct function calls in place of message sending functions (such as p_send) does no harm to the design and can only improve performance.

The correct approach is to consider performance **where it is specified to be important** in the original design. The above timings are provided to guide that consideration.

DLLs

The functions in this chapter, described in the context of object oriented programming, may also be used to implement re-entrant dynamic link libraries (sometimes called DLLs on PCs) with the following benefits:

- application code can break the 64K code segment limit
- there is only one copy of the DLL in memory at a time however many processes are accessing it

DLLs may be loaded and linked using p_loadlib and p_linklib.

The DLL functions would be organised into one or more groups (root classes in OOP) of up to 255 functions (methods in OOP). The functions may be called (without having to create an object instance) using p_exactsend (where the object instance parameter has no special significance).

[2]In a professional development environment, this should be stated in a *Software Requirements Specification*.

Category functions

This section describes the following functions:

p_loadlib	which loads a DYL from a file (which normally has the extension *.DYL*).
p_openlib p_loadfilelib	which are used in conjunction to load DYLs that have been combined into a multiple DYL file (normally an executable)
p_unloadlib	which frees the memory occupied by a DYL (provided another process is not using it)
p_linklib	which is used by an image category to link itself and may also used to link DYLs which reference another DYL that has just been loaded
p_findlib	which is used to obtain the handle of a loaded category from its name
p_getlibh	which converts a category number to a category handle
p_cpycat, p_ccpy	which may be used to copy data from a category into the data segment

An application that uses a DYL to implement a transient subsystem should call p_unloadlib as soon as it has finished using it so that the memory is returned to the system.

Image categories are loaded by calling p_execc - described in the chapter *Processes and Inter-Process Messaging*.

An application process must connect to the file server before calling p_loadlib, p_openlib or p_loadfilelib. However, this is normally taken care of by the C startup module (the code that precedes main) supplied as standard for use with the PLIB library.

p_loadlib — Load a DYL

```
INT p_loadlib(TEXT *pName, HANDLE *pCatHandle, INT link);
```

Load (and optionally link) the DYL with the zero terminated file specification pName and, if successful, write the category handle to *pCatHandle and return zero.

If link is TRUE, the DYL is automatically linked after loading. In this case, all externally referenced categories must already have been loaded (otherwise p_panic is called).

You would normally only set link to FALSE if the DYL externally references another DYL which you have yet to load (where the DYL would be linked subsequently by calling p_linklib).

The file specification pName is parsed with a related name of ".DYL". The name component from pName is used to name the category segment - the segment name has the extension ".DYL" regardless of pName.

If the parse fails, p_loadlib returns with one of the negative error numbers returned by p_fparse. Other possible error returns are:

E_FILE_NXISTS	pName does not exist
E_GEN_IMAGE	pName is not a valid DYL
E_GEN_OPEN	pName has already been loaded by the caller
E_FILE_EXIST	a different DYL (ie with a different checksum) with the same name already exists

If the same DYL has already been loaded by another process, it is shared and not re-loaded.

A loaded DYL remains in memory until the process terminates or until the process unloads the DYL by calling p_unloadlib.

Note that DYLs that are in the ROM are deemed to be loaded by default. There is therefore never any need to call p_loadlib for such a DYL.

p_openlib — Open a file containing multiple DYLs

```
INT p_openlib(VOID **pfcb, TEXT *pName);
```

Open a channel to the file containing multiple DYLs as specified by the zero terminated pName and, if successful, write the file channel to *pfcb and return zero.

The file specification pName is parsed with a related name of ".IMG". If the parse fails, p_openlib returns with one of the negative error numbers returned by p_fparse.

Most commonly pName is an image file to which the multiple DYLs have been added using the *emake* program. If the file is not a valid multiple DYL file, the error E_GEN_IMAGE is returned.

The returned *pfcb is passed to subsequent calls to p_loadfilelib, described next, to load the DYLs.

The file channel *pfcb is obtained by an internal call to p_open and when access to the file is no longer required, it should be closed by calling p_close.

The fact that *pfcb is a regular binary file handle (opened with P_FRANDOM but not P_FUPDATE) may be exploited to read any other data from the file.

Any error that may be returned by p_open may also be returned by p_openlib.

p_loadfilelib — Load from a multiple DYL file

```
INT p_loadfilelib(VOID *fcb, UINT n, HANDLE *pCatHandle, INT link);
```

Load (and optionally link) the nth DYL from the multiple DYL file channel fcb and, if successful, write the category handle to *pCatHandle and return zero.

The number n selects the DYL to be loaded from the file in the order that DYLs were originally added by *emake*. To load the first DYL from the file, n should be zero.

As well as storing the DYLs themselves, the multiple DYL file stores the original DYL file names. These file names are used to name the segment in the same way as if the DYL had been loaded directly using p_loadlib.

If link is TRUE, the DYL is automatically linked after loading. In this case, all externally referenced categories must already have been loaded (otherwise p_panic is called).

You would normally only set link to FALSE if the DYL externally references another DYL which you have yet to load (where the DYL would subsequently be linked by calling p_linklib).

The file channel fcb is obtained by previously opening a multiple DYL file using p_openlib, described above.

If the DYL has already been loaded by another process, it is shared and not be re-loaded. If, however, the library has already been loaded by the caller, p_loadfilelib fails and returns the negative E_GEN_OPEN.

A loaded DYL remains in memory until the process terminates or until the process unloads the DYL by calling p_unloadlib.

p_unloadlib — Unload a dynamic library

```
INT p_unloadlib(HANDLE catHandle);
```

Unload DYL catHandle from memory and return zero if successful.

If the caller has not loaded the DYL, the negative error number E_GEN_NOTOPEN is returned.

A loaded DYL may be shared by multiple processes. Each time a process loads a particular DYL an access count is incremented. Unloading the library decrements the access count and if this takes the access count to zero, the DYL memory segment is deleted.

p_linklib — Link a loaded category

```
VOID p_linklib(HANDLE catHandle);
```

Link the loaded image category or DYL with handle catHandle or, if catHandle is zero, link the image category containing the call to p_linklib.

All externally referenced categories must already have been loaded (otherwise p_panic is called).

Calling this function is harmless if the category has already been linked. Since categories are shared, a category may have already been linked because another process has previously loaded and linked it.

Applications programs which reference external DYLs by category number must call `p_linklib(0)` to link themselves early in their initialisation (but after loading any referenced DYLs). Applications that do not access any external DYLs, or only access DYLs that are in the ROM, may contain the call to `p_linklib(0)` as the first line of their `main()` function.

In many cases, DYLs can be linked by the call to `p_loadlib` or `p_loadfilelib` that loads them - you only need to use `p_linklib` on a DYL in the relatively rare case where the link has to be deferred because the DYL references another DYL which has yet to be loaded.

p_findlib — Find a category handle

```
INT p_findlib(TEXT *pName, HANDLE *pHandle);
```

Write the category handle of the loaded category segment with the zero terminated name `pName` to `*pHandle` and return zero or, if no such category exists, return the negative error `E_FILE_NXISTS`.

The category may be in the ROM or in a RAM memory segment.

The name pointed to by `pName` is a category segment name, not a file specification. Thus *form.dyl* is a valid name, but *rom::form.dyl* is not. The name should include an extension - image categories have the extension *.$sc* and DYLs have the extension *.dyl*.

Although `p_findlib` will find a RAM-based DYL, it does not do anything to keep the DYL loaded. Normally, you would only use `p_findlib` to get the handle of a ROM-based DYL. To get the handle of a RAM-based DYL, you should use `p_loadlib` or `p_loadfilelib` which will not load the DYL if it is already loaded and will increment the usage count to keep it loaded until the program calls `p_unloadlib` (or until the program exits).

p_getlibh — Convert a category number to a handle

```
HANDLE p_getlibh(INT catNum);
```

Return the handle of the external category specified by the category number `catNum`.

If `catNum` is zero, `p_getlibh` returns the handle of the local category.

Any number greater than zero indexes the external category table to get the external category handle.

The function calls `p_panic` if `catNum` is outside the range of the external category table or if the local category has not been linked.

p_cpycat — Copy data from a category

```
VOID p_cpycat(UINT catNum, VOID *pTarget, VOID *pSource, UINT count);
```

Copy `count` bytes of data from offset `pSource` in the segment of the category specified by the category number `catNum`, to `pTarget` in the caller's data segment.

Calls `p_panic` if `catNum` is outside the range of the external category table or if the local category has not been linked.

p_ccpy — Copy data from the local category

```
VOID p_ccpy(VOID *pTarget, VOID *pSource, UINT count);
```

Copy `count` bytes of data from offset `pSource` in the code segment containing the call to `p_ccpy`, to `pTarget` in the caller's data segment.

Object functions

This section describes the following functions:

`p_new, f_new, p_newlibh, f_newlibh`	which create an instance of an object, given its class
`p_send, p_supersend, p_entersend, p_exactsend`	which send a message to an object instance, given its address
`f_newsend, f_newlibhsend`	which create an object and send it an initialisation message
`p_reclass, p_reclassbyhandle`	which change the class of an instance

p_new (or f_new) Create an object by category number

```
VOID *p_new(INT catNum, INT classNum);
VOID *f_new(INT catNum, INT classNum);
```

Create an instance of class `classNum` from the category specified by the category number `catNum`, returning the address of the object, or `NULL` if there is insufficient memory to allocate the instance from the heap.

This function converts `catNum` to a category handle by calling `p_getlibh` and then calls `p_newlibh` (described next).

The class number `classNum` specifies the class by indexing the class table in the category `catNum`.

Except for the instance header (which points to the specified class descriptor), the rest of the property is initialised to zero.

Calls `p_panic` if `catNum` is outside the range of the external category table, if the local category has not been linked or if `classNum` is outside the range of the class table.

The function `f_new` is identical except that it calls `p_leave(E_GEN_NOMEMORY)` rather than return `NULL`.

p_newlibh (or f_newlibh) Create an object by category handle

```
VOID *p_newlibh(HANDLE catHandle, INT classNum);
VOID *f_newlibh(HANDLE catHandle, INT classNum);
```

Create an instance of class `classNum` from the category specified by the category handle `catHandle`, returning the address of the object, or `NULL` if there is insufficient memory to allocate the instance from the heap.

The class number `classNum` specifies the class by indexing the class table in the category `catHandle`.

Except for the instance header (which points to the specified class descriptor), the rest of the property is initialised to zero.

Calls `p_panic` if `catHandle` is not the handle of a valid external category or if `classNum` is outside the range of the class table.

The function `f_newlibh` is identical except that it calls `p_leave(E_GEN_NOMEMORY)` rather than return `NULL`.

p_send (or f_send) Send a message to an object

```
INT p_send(VOID *pObject, INT methodNum, ...);
INT f_send(VOID *pObject, INT methodNum, ...);
INT p_send2(VOID *pObject, INT methodNum);
INT f_send2(VOID *pObject, INT methodNum);
INT p_send3(VOID *pObject, INT methodNum, VOID *p1);
INT f_send3(VOID *pObject, INT methodNum, VOID *p1);
INT p_send4(VOID *pObject, INT methodNum, VOID *p1, VOID *p2);
INT f_send4(VOID *pObject, INT methodNum, VOID *p1, VOID *p2);
INT p_send5(VOID *pObject, INT methodNum, VOID *p1, VOID *p2, VOID *p3);
INT f_send5(VOID *pObject, INT methodNum, VOID *p1, VOID *p2, VOID *p3);
```

Call the method function corresponding to methodNum of the object instance pObject, passing the function from zero to three additional parameters and return the value (which should be the size of an INT) returned by the selected method function.

You can either use p_send, which presents the stack-based CDECL calling convention, or one of the p_send? variants, which use a more efficient register calling convention. Each f_ variant is identical to the corresponding p_ variant except that, if the method returns a negative value err, it calls p_leave(err) rather than returning err.

The method function is called with the same parameters as passed in the p_send except that the methodNum parameter is removed. The method function must use either the stack-based CDECL or the more efficient register-based METHOD_CALL calling convention.

The following example illustrates the form of a method function declaration, using the METHOD_CALL calling convention, together with the corresponding p_send? method function call.

```
#pragma save
#pragma METHOD_CALL

GLDEF_C VOID myobject_mymethod_one(VOID *self,TEXT *buf,UINT len)
    {
    ...
    }

GLDEF_C VOID myobject_mymethod_two(VOID *self,TEXT *buf,UINT len)
    {
    ...
    p_send4(self,O_MYMETHOD_ONE,buf,len);
    ...
    }

#pragma restore
```

The symbol O_MYMETHOD_ONE, representing the method number of the method function myobject_mymethod_one, is generated by the category-building tools.

The search for a method function starts from the class pointed to by the object header (and which was used to create the object). If this class does not contain a method corresponding to methodNum, the search continues with the superclass and so on until a method is found. If the search fails (that is, it fails to find a method in the root class), p_send calls p_panic.

p_supersend Send a message to be handled by the superclass

```
INT p_supersend(VOID *pObject, INT methodNum, ...);
INT p_supersend2(VOID *pObject, INT methodNum);
INT p_supersend3(VOID *pObject, INT methodNum, VOID *p1);
INT p_supersend4(VOID *pObject, INT methodNum, VOID *p1, VOID *p2);
INT p_supersend5(VOID *pObject, INT methodNum, VOID *p1, VOID *p2, VOID *p3);
```

Behaves exactly as for p_send except that the search for a method function corresponding to methodNum starts at the superclass of the class of the method containing the call to p_supersend (ignoring the object header of pObject).

The class of the method is determined by retrieving (from the stack) the last class descriptor that provided the path to the method containing the call to p_supersend. This can go wrong if the calling method was reached via a direct function call. If there is a possibility of the method being called directly, consider

using p_exactsend as an alternative to p_supersend or, if possible, a direct function call (which is, in any case, better for performance).

Within reason, pObject must be the same as that passed to the method calling the p_supersend. In nearly all cases, methodNum is also the same as that which selected the calling method (and in many cases the remainder of the parameters, if any, are the same too).

This function is typically used when a method function wishes to call the method function corresponding to the same methodNum of a superclass (when a method function includes the superclass method function's processing in its own processing). Very rarely, it is used to call a different method number of a superclass.

p_entersend Send a message with an enclosing p_enter

```
INT p_entersend(VOID *pObject, INT methodNum, ...);
INT p_entersend2(VOID *pObject, INT methodNum);
INT p_entersend3(VOID *pObject, INT methodNum, VOID *p1);
INT p_entersend4(VOID *pObject, INT methodNum, VOID *p1, VOID *p2);
INT p_entersend5(VOID *pObject, INT methodNum, VOID *p1, VOID *p2, VOID *p3);
```

Behaves exactly as for p_send except that the method function called is entered as if it had been called with p_enter.

Note that, as for p_send, the target method function must use either CDECL or METHOD_CALL calling convention (and not ENTER_CALL as for functions entered via p_enter).

If p_leave(err) is called before the entered method function returns, the stack is unwound and the call to p_entersend returns err. The call to p_leave may occur in the entered function or in a sub-function and so on.

The enter and leave mechanism (which is commonly used to implement structured error recovery) is described in the chapter *Error Handling*.

p_exactsend Send a message to a specific class

```
INT p_exactsend(HANDLE catHandle, INT classNum, VOID *pObject, INT methodNum, ...);
```

Behaves as for p_send except that the search for a method function corresponding to methodNum starts at the class specified by catHandle and classNum (ignoring the object header of pObject).

Like p_supersend, this function is typically used to access a superclass method corresponding to the same methodNum. It is typically used for one of the following two reasons:

- to call a first generation method of a superclass (obscured by an intervening second generation method) from a third generation method
- to call a superclass method when the calling method may be called by a direct function call

f_newsend Create and initialise an object by category number

```
VOID *f_newsend(INT catNum, INT classNum, INT methodNum, ...);
```

Create and initialise an object by:

- creating an instance of class classNum from the category specified by the category number catNum
- calling the method function corresponding to methodNum of the created object, passing the function from zero to three additional parameters

The function returns the address of the created object.

Behaves as for f_new followed by a p_send of methodNum to the created object where methodNum is typically an initialisation method that calls p_leave if an error occurs.

If there was insufficient memory to create the object, it calls p_leave(E_GEN_NOMEMORY) - just like f_new.

What makes f_newsend more valuable than an apparently equivalent call to f_new followed by a p_send is its error handling following a successful f_new: if there is a call to p_leave with a negative parameter before methodNum returns, the partially initialised object is sent a destroy message and the call to p_leave is propagated. The net effect is that the function is either successful (in which case it returns the address of the created and initialised object) or it leaves having cleaned up the partially created object.

There is no requirement for the method function `methodNum` to return zero (as there normally is for entered functions) and the method function may be declared as a `VOID` or otherwise (any value returned by the method function is lost).

Calls `p_panic` if `catNum` is outside the range of the external category table, if the local category has not been linked or if `classNum` is outside the range of the class table.

f_newlibhsend Create and initialise an object by category handle

```
VOID *f_newlibhsend(HANDLE catHandle, INT classNum, INT methodNum, ...);
```

Behaves exactly as for `f_newsend`, described above, except that the category containing the class of the object to be created is identified by its category handle rather than its category number.

In fact, `f_newlibhsend` is more primitive than `f_newsend` (which calls `p_getlibh` to convert the category number to a category handle before calling `f_newlibhsend`).

p_reclass Reclass an object by category number

```
VOID p_reclass(INT catNum, INT classNum, VOID *pObject);
```

Change the class of which `pObject` is an instance to that specified by the category number `catNum` and `classNum`.

Within reason, the new class is a subclass or a superclass of the original class, with the same property. The function calls `p_panic` if you attempt to reclass to a class with a different property length.

Calls `p_panic` if `catNum` is outside the range of the external category table; if the local category has not been linked or if `classNum` is outside the range of the class table.

p_reclassbyhandle Reclass an object by category handle

```
VOID p_reclassbyhandle(HANDLE catHandle, INT classNum, VOID *pObject);
```

Change the class of which `pObject` is an instance to that specified by the category handle `catHandle` and `classNum`.

Within reason, the new class is a subclass or a superclass of the original class, with the same property. The function calls `p_panic` if you attempt to reclass to a class with a different property length.

Calls `p_panic` if `catHandle` is not the handle of a valid external category or if `classNum` is outside the range of the class table.

INDEX

#pragma 4
  call 6
  restore 6
  save 6
.$nn 64
.$SC 64
.APP 159
.DYL 64, 212
.IMG 158
.LDD 64
.PDD 64
.wav 189
.wve 188
80286 63
80386 63
80486 63
8086 1, 63
8087 emulator 39, 46
  avoiding 40, 46
A-Law
  decoding 191
  encoding 189
absolute timer 104
active objects 10
ADDFILE 159
alloc heaven 67
arccosine 45
architecture
  SIBO 1
arcsine 45
arctangent 45
array 23
  binary search 23
  sort 24
ASIC 1
asynchronous
  event 86
  file server operations 122
  image,load 162
  message 171
  sound,play back 193
  sound,record 192
  timer 105
asynchronous request 80
  building 82
  cancelling 82
  cancelling,simulation 122
  file,cancel 150
  I/O 90
  status word 81
  wait 82, 86
attached
  driver 91
  I/O devices 84
auto switch off 180
battery
  backup 182
  type 182
  voltage level 182
binary file
  access 140
  close 143
  flush 145
  open 142
  position 144
  read 144
  seek 144
  write 144
binary search 23
buffer 11
  align 13
  character,repeat 13
  checksum 14
  compare 18
  compare,case independent 19
  copy 11
  functions 11
  locate byte 19
  locate character,case independent 20
  locate sub-buffer 21
  locate sub-buffer,case independent 21
  multiple arguments,convert 30
  pattern match 21
  pattern match,case independent 22
  replicate 12
  searching 19
  swap 13
BYTE 5
C
  floating point 39, 89
  prototype 4
  startup module 7, 117
C compiler
  Microsoft 4
  TopSpeed 3
  Turbo C 4
calling convention 4, 6
  register_based 7
  stack_based 6
case conversion table 15
category 212
  data,copy from 222
  data,copy from local 222
  DYL 212, 214
  DYL,load 220
  DYL,unload 221
  dynamic library 212
  dynamic linkage 216
  external 213
  handle 213
  handle to number,convert 222
  handle,find 222
  handle,referencing 214
  image 212
  linkage 213
  load module 215
  loaded & linked,structure 216
  loaded DYL,link 221
  loaded,link 221
  multiple DYL,load 221
  multiple DYL,open 221
  number 213
  number,external 213

number,local 213
unlinked,structure 216
CDECL 6, 59, 218
cell 66
allocate 67
change contents 68
change size 68
free 68
get length 69
channel
cancel request 97
close 96
high speed serial 118
I/O functions 92
opening to device 92
operations 90
read from 96
services,file server 121
timer,close 106
timer,open 105
to device,opening 90
write to 97
character
alphabetic,test 15
alphanumeric,test 16
case conversion table 15
classification 14
classification table 14
control,test 16
conversion 14
fold 17
fold table 15
hexadecimal,test 16
lower case,convert 17
lower case,test 15
non-whitespace,skip 17
numeric,test 16
printable graphic,test 16
printable,test 16
punctuation,test 16
upper case,convert 17
upper case,test 15
whitespace,skip 16
whitespace,test 16
Clarion Software 3
class 10, 211
descriptor 211
instance,create 223
property 212
root 212
specific,message send 225
superclass 212
superclass chaining 217
time 109
classification table 14
CLIB
library 1, 8
client 169
file server 91
functions 174
message,asynchronous send & wait 175
message,send 174
message,send & wait 175
message,send and wait 175
client-server 168
clock 1, 118
real-time 1
code segments 2, 64
CON: 99
config.h 179, 180
connect file server 117
console 8, 9
arguments,convert and write 101
change mode 100
change size 100
character,get 101
character,write 101
device driver 99
I/O 99
redirecting writes 100
string,get 101
string,get with prompt 101
string,write 101
control blocks
process 152
coordinates 34
cosine 44
country code 178
data segments 2, 64
database 195
close 200
compress 201
continuation sub-fields 197
copy 201
descriptive record,read 203
descriptive record,write 203
extended header,read 202
extended header,write 203
file header 196
first record,read 205
flush 200
index 195
last record,read 205
next record,read 204
open 198
open quickly 200
OPL 198
overwritten buffer,notify 200
previous record,read 204
record 196
record number,sense 209
record type,application specific 197
record type,deleted 196
record type,descriptive 197
record type,field information 196
record type,standard 196
record,append 205
record,copy down 201
record,count 209
record,end of file 198
record,erase 206
record,find 207, 208
record,number 198
record,update 206
size,find 202
specific record,read 204
specific record,read & sense 204
version number,fetch 203
databse
string fields 197
DatApp1 157
DatApp2 157
DatApp3 157
DatApp4 157
DatApp5 157

DatApp6 157
DatApp7 157
DatATFlag 157
DatClassHandle 156
DatClassPtr 156
DatCommandPtr 157
DatCountrySeg 156
DatDialogPtr 157
date
  am/pm suffixes,get 110
  day in month suffixes,get 110
  day name abbreviation,get 110
  day name,get 109
  format string 112
  language dependent 110
  month name abbreviation,get 110
  month name,get 110
  preferences 111
  string generation 112
  text form 109
DatEClassHandle 156
DatEClassPtr 156
DatEnterFramePtr 156
DatGate 157
DatHandNext 156
DatHandPrev 156
DatHeapLocked 157
DatLocked 157
DatOsFramePtr 157
DatProcessNamePtr 157
DatStatusNamePtr 157
DatTest 157
DatUsedPathNamePtr 158
DatWordDead 156
DbfAbsRead 204
DbfAbsReadSense 204
DbfAppend 205
DbfBackRead 204
DbfClose 200
DbfCompress 201
DbfCopyDown 201
DbfCopyFile 201
DbfCount 209
DbfDescRecordRead 203
DbfDescRecordWrite 203
DbfEraseRead 206
DbfExtHeaderRead 202
DbfExtHeaderWrite 203
DbfFileSize 202
DbfFindRead 208
DbfFindReadField 207
DbfFirstRead 205
DbfFlush 200
DbfHeader 198, 199
DbfLastRead 205
DbfNextRead 204
DbfOpen 198
DbfOpenArgs 200
DbfQuickOpen 200
DbfRecord 196
DbfSense 209
DbfTrash 200
DbfUpdate 206
DbfVersion 203
delta queue 27, 79, 154
  add entry 27
  remove entry 27
device driver 89
  attached 91
  CON: 99
  delete 98
  external 89
  FIL: 90
  find 99
  floating point 89
  functions 98
  logical 89
  logical,load 98
  PAR: 90
  physical 89
  physical,load 98
  query units supported 98
  SND: 187
  TIM: 90
  TTY: 90
devices 126
  console 8, 9
  default 126
  formatting 131
  information 129
  list 128
  local SSD,direct read 133
  local,media information,read 132
  opening channel to 90, 92
  operations 90, 127
  parallel port 9
  serial port 9
  sound driver 9
digitiser 152
digitising pad 1
directory 126
  create 138
  default 126
  delete 137
  existence,test 136
  list 133
  operations 133
  rename 136
DLL 219
DOUBLE 5
  from string,convert 42
  random 46
  to string,convert 41
doubly linked queue 25
  add entry 26
  remove entry 27
drives
  flash EPROM 118, 119
  masked ROM 118
  once programmable ROM 118
  SSD 118
  static RAM 118, 119
DYL 212, 214
  ROM 213
E_BATTERY_ALKALINE 182
E_BATTERY_NICAD_1000 182
E_BATTERY_NICAD_600 182
E_BATTERY_UNKNOWN 182
E_CONFIG 43, 108, 111, 180
E_CPB 162, 163
E_CURRENCY_AFTER 43
E_CURRENCY_BEFORE 43
E_FILE_PENDING 81
E_FILE_xxx 56
E_GEN_xxx 56
E_IMPERIAL 44

E_MAX_ENV_SIZE 75
E_MAX_GROWBY 51
E_MAX_PRIORITY 154
E_MAX_PROCESSES 151
E_MESSAGE 168
E_METRIC 44
E_MIN_PRIORITY 154
E_NORMAL_EXIT 50
E_NOSPACE_BETWEEN 43
E_PANIC_EXIT 50
E_PROC 153
E_SEGMENT_DEVICE 51
E_SEGMENT_HIGH 51
E_SEGMENT_LOCKED 51
E_SEGMENT_LOW 51
E_SPACE_BETWEEN 43
E_SUPPLY 182
E_SUPPLY_INFO 183
E_SUPPLY_WARNINGS 184
E_TASK_PANIC_EXIT 50
edump.exe 159
EM$ 39
emake.exe 158
end of file 146
enter a function 58, 59
ENTER_CALL 59, 218
environment variables 63, 75
  delete 76, 77
  EM$ 39
  find 77
  get value 75, 76
  set value 76
EPOC 2, 63
  device drivers 89
  files 117
  I/O system 89
  multi-tasking 151
  operating system 2
  PC 2, 182
  program environment 2
  ROM 2
  single-user 151
  system services reference 9
  system tables 14
  timer and delta queue 27
epoc.h 50, 69, 104, 132, 153, 158, 159, 163, 168, 178, 183, 186, 188
EPROM
  flash 118, 119
error
  codes 50
  handling 49
  number to string,convert 56
  returning 56
event 9, 10, 79, 81, 85, 94, 105, 151, 155, 171, 182
  asynchronous 86
executable 151
exit to DOS 194
exponential 45
external device driver 89
external memory segments 3, 71
f_alloc 67
f_fparse 123
f_leave 60
f_new 223
f_newlibh 223
f_newlibhsend 226
f_newsend 225
f_open 92
f_read 96
f_realloc 68
f_seek 144, 149
f_send 224
f_write 97
FIL: 90
file 117
  .IMG 158
  asynchronous request,cancel 146, 150
  attributes,set 138
  binary 140
  binary,close 143
  binary,open 142
  binary,position 144
  binary,read 144
  binary,seek 144
  binary,write 144
  buffers,flush 145
  create date,set 139
  database 195
  delete 137
  end of,set 146
  image 158
  information,get 135
  label medium,set 138
  operations 133
  record text 147
  record text,open 148
  record text,position 149
  record text,read 148
  record text,seek 149
  record text,write 149
  rename 136
  shared access 140
  sound 188
  stream text access 146
  stream text,open 147
  text,close 148
  text,flush 150
  text,set end 150
file server 7, 91, 117, 168, 195
  asynchronous operations 122
  channel services 121
  client 91
  connect 117
  default device 126
  default directory 126
  default node 126
  default path 121
  file specification 120
  file specification,change directory 125
  file specification,manipulation 123
  file specification,parse 123
  non-channel services 121
  process default path,get 127
  process default path,set 126
  process-id default path,get 127
  shared access 140
  system default path,set 126
  unattended applications 118
file systems 117
  LOC:: 63, 71, 117, 128, 144, 146
  MSDOS 120, 144, 146
  REM:: 117

ROM:: 117
UNIX 146
Flash filing system 119
Flash-friendly 195
floating point
add 46
assignment 46
avoiding emulator 46
C 39
compare 47
divide 47
double to integer,convert 48
double to long,convert 48
emulator data space 65
integer part 48
integer to double,convert 48
long to double,convert 48
modulus 48
multiply 47
negate 47
subtract 47
fold table 15
format 131
formatting
device 131
dual density 131
functions
buffer 11
category 220
channel I/O 92
character classification 14
character conversion 14
client 174
database 198
DBF 198
device driver 98
enter 59
integer conversion 29
leave,on error 60
leave,standard 60
long integer 44
object 223
rectangle 34
scientific 44
semaphore,primitive 84
server 172
string 11
wait handlers 87
GLDEF_C 5
GLDEF_D 5
GLREF_C 5
GLREF_D 5
granularity 69
growing the heap 66
HANDLE 5
hardware interrupts 84, 154, 177
hardware protection 3
header files
config.h 179, 180
epoc.h 50, 69, 104, 132, 153, 158, 159, 163, 168, 178, 183, 186, 188
p_config.h 43, 111
p_date.h 106
p_dbf.h 196, 199, 200, 205
p_file.h 56, 81, 90, 105, 106, 107, 121, 123, 128, 129, 134
p_gen.h 32, 56
p_graf.h 35, 100
p_math.h 41
p_que.h 25
p_std.h 5
plib.h 4
heap 66
alloc heaven 67
allocate failure 66
allocator 66
cell 66
cell,allocate 67
cell,change contents 68
cell,change size 68
cell,free 68
cell,get length 69
fragmentation 67
granularity 69
growing 66
integrity,check 70
potential free space 70
set granularity 69
shrinking 66
structure 66
walk 69
high speed serial channel 118
hook notifier interface 58
I/O
asynchronous request 90
channel,cancel request 97
channel,close 96
channel,read from 96
channel,write to 97
console 99
device drivers 89
device,opening channel to 90
external device driver 89
LDD 89
logical device driver 89
operations on open channel 90
PDD 89
physical device driver 89
reference 9
semaphore 81, 85, 90
semaphore,signal 85
semaphore,signal process 85
semaphore,wait 86
start operation 93, 94
start operation & wait 95
system 89
identify machine type 186
image files 158
ImgHeader 158
inactive process 164
index table 65
INT 5
integer
ULONG random 44
integer conversion functions 29
INT to decimal char 29
LONG from signed decimal 32
LONG to decimal char 29
UINT to char 30
ULONG from string 33
ULONG to char 30
UWORD from string 32
WORD from signed decimal 32
integrated circuit 1
inter process messaging 58, 168
interrupt

disabling 3
hardware 84, 154
software 4, 218
vectors 63
introduction 1
ISDN combo sound system 1
keyboard 185
language code 178
LCD display 1
LDD 89, 91
leave
on error 60
standard 60
leave a function 58
libraries 1
CLIB 1, 8
object dynamic 10
PLIB 1
TopSpeed C 8
window server 1
WLIB 1, 9
LOC:: 63, 71, 117, 128, 144
changing 128
LOCAL_C 5
LOCAL_D 5
logarithm 45
logarithm,natural 45
logical device driver 89, 91
LONG 5
long integer functions 44
loudness 193
machine type 186
macros 41
magic statics 65, 156
main() 7
mains adaptor 182
manuals 8
MC400 10
media type 132
memory
allocate failure 66
allocation 63
available 71
heap 66
moving 3
RAM disk usage 71
segment 64, 71
segment,adjust size 74
segment,close 74
segment,copy from 73
segment,copy to 73
segment,create 72
segment,decrease usage count 75
segment,delete 73
segment,device 64
segment,dynamic 64
segment,find by name 74
segment,get size 74
segment,increase usage count 75
segment,lock 75
segment,open 73
segment,unlock 75
system usage 63, 71
message 168
free 174
processing order 172
queue 172
reception 173
reception,asynchronous 173
reception,cancel 174
reception,wait 173
send 174
send & wait 175
slots 168, 174
Microsoft C 4
monomorphic 217
mouse 152
multi-tasking 2, 7, 151
polling 84
waiting 84
mutual exclusion semaphore 80
name of process 155
natural logarithm 45
nodes 117, 126
default 126
information 128
list 127
operations 127
notify
error & response 58
hook interface 58
message & response 57
service 56
state get 58
state,set 58
unhook interface 58
null process 164, 180
number representation preferences 43
object
by category handle,create 223
by category handle,create & init 226
by category handle,reclass 226
by category number,create 223
by category number,create & init 225
by category number,reclass 226
class 10
classes 211
destruction 212
dynamic libraries 10
instance 212
message specific class,send 225
message to superclass,send 224
message,entersend 225
messages 216
message,send 224
messages,performance 218
method function 217
method function,convention 217
method number 216
monomorphic 217
polymorphic 217
property 212
reference 10
root class 212
superclass 212
object-oriented programming 10
panic 53
OLIB reference 10
OLIB.DYL 10
opening channel to device 90
operating system
data space 63
overview 2
operations on
any process 165
current process 163

devices 127
directories 133
files 133
nodes 127
open channel 90
OPL 15, 56, 157, 195
database 198
overview
system memory 63
p_absrec 37
p_acos 45
p_adjust 68
p_alen 69
p_allchk 70
p_alloc 67
p_allowoff 181
p_allspc 70
p_allwalk 69
p_asin 45
p_atan 45
p_atob 30
p_atos 31
p_backlight 186
p_bcmp 18
p_bcmpi 19
p_bcpy 11
p_bfil 13
p_bloc 19
p_bloci 20
p_bmatch 21
p_bmatchi 22
p_brep 12
p_bsrch 23
p_bsub 21
p_bsubi 21
p_bswap 13
p_ccpy 222
P_CD_PARENT 125
P_CD_ROOT 125
P_CD_SUBDIR 125
p_chdir 125
P_CLASS 211
p_close 96, 143, 148
p_config.h 43, 111
p_cos 44
p_cpycat 222
p_crc 14
p_date 103, 107, 115
p_date.h 106
p_dayinm 108
P_DAYSEC 106, 107, 108, 115
p_dbf.h 196, 199, 200, 205
P_DECLAREQ 25
p_delenv 76
p_delenviron 77
p_delete 137
P_DELTA 27
p_deque 27
p_dequed 27
p_devdel 98
p_devfnd 99
p_devqu 98
p_dinfo 129
p_ds2str 115
p_dstodt 107
p_dstost 107
p_dt2str 115
p_dtob 41
P_DTOB_EXPONENT 42
P_DTOB_FIXED 42
P_DTOB_GEN_LIM 42
P_DTOB_GENERAL 42
p_dttods 108
p_emprec 36
p_enque 26
p_enqued 27
p_enter 59
p_entersend 225
P_ENVMAX 75
p_errs 56
p_exactsend 225
p_execc 160
p_execcasync 162
p_exit 53
p_exp 45
P_FABS 144
P_FABSOLUTE 105
p_fadd 46
P_FADIR 134
P_FAHIDDEN 134, 138
P_FAMOD 134, 138
P_FAPPEND 143, 198
P_FASYSTEM 134, 138
P_FATEXT 134
P_FAVOLUME 134
P_FAWRITE 134, 138
P_FBLKSIZE 144
P_FCANCEL 90, 96, 97, 105, 146, 147, 150
P_FCLOSE 90
p_fcmp 47
P_FCREATE 142, 198
P_FCUR 145
p_fdate 139
P_FDEVICE 121, 128
P_FDIR 121, 133
p_fdiv 47
P_FEND 145
P_FFLUSH 91, 96, 145, 147, 150
P_FFORMAT 121, 131
p_file.h 56, 81, 90, 105, 106, 107, 121, 123, 128, 129, 134
p_findenviron 77
p_findlib 222
p_finfo 135
p_fld 46
P_FMAXRSIZE 148
P_FMAXSSIZE 140, 144
P_FMEDIA_COMPRESSIBLE 130
P_FMEDIA_DUAL_DENSITY 130
P_FMEDIA_DYNAMIC 130
P_FMEDIA_FLASH 130
P_FMEDIA_FLOPPY 129
P_FMEDIA_FORMATTABLE 130
P_FMEDIA_HARDDISK 129
P_FMEDIA_INTERNAL 130
P_FMEDIA_RAM 130
P_FMEDIA_ROM 130
P_FMEDIA_UNKNOWN 129
P_FMEDIA_WRITEPROTECTED 130
p_fmul 47
P_FNAMESIZE 121
p_fndenv 77
p_fneg 47
P_FNODE 121, 127
P_FOPEN 142, 198

p_fparse 123
p_frand 46
P_FRANDOM 143
P_FREAD 90, 93, 147
p_free 68
P_FRELATIVE 105
P_FREPLACE 142, 198
P_FREWIND 149
P_FRSENSE 149
P_FRSET 149
P_FSENSE 91, 96
P_FSET 91, 96
P_FSETEOF 146, 147, 150
P_FSHARE 140, 143, 198
P_FSTREAM 121, 142
P_FSTREAM_TEXT 121, 147
p_fsub 47
P_FSYSTYPE_FLAT 128
P_FSYSTYPE_HIER 128
P_FTEXT 121, 148
P_FUNIQUE 143, 198
P_FUPDATE 143, 198
P_FWRITE 90, 93, 147
p_gen.h 32, 56
p_getampmtext 110
p_getauto 181
p_getautomains 181
p_getbacklight 187
p_getbat 185
p_getch 101
p_getctd 43, 111, 180
p_getenv 75
p_getenviron 76
p_getl 101
p_getlanguage 179
p_getlcd 186
p_getlcdcontrast 186
p_getlibh 222
p_getnotify 58
p_getosd 178
p_getowner 166
p_getpid 164
p_getpri 165
p_getpsu 178
p_getpth 127
p_getpthbyid 127
p_getram 71
p_getres 177
p_gets 101
p_getscancodes 185
p_getsnd 188
p_getsuffixes 110
p_gettext 179
p_gltob 30
p_graf.h 35, 100
p_gtob 30
p_hgran 69
p_hwexit 194
P_INFO 133, 134, 135
P_INITQ 25
p_insrec 35
p_int 48
p_inti 48
p_intl 48
p_ioa 93
p_ioc 94
p_ioc(P_FABSOLUTE)
absolute timer 106
p_ioc(P_FRELATIVE)
relative timer 105
p_iosignal 85
p_iosignalbypid 85
p_iow 95
p_iow(P_FCANCEL) 97, 106, 146, 150
p_iow(P_FFLUSH) 145, 150
p_iow(P_FSETEOF) 146, 150
p_iowait 86
p_ioyield 86
p_isalnum 16
p_isalpha 15
p_iscntrl 16
p_isdigit 16
P_ISEMPTYQ 26
p_isgraph 16
p_islower 15
p_isprint 16
p_ispunct 16
p_isspace 16
p_isupper 15
p_isxdigit 16
p_itob 29
p_itof 48
P_JCENTRE 13
P_JLEFT 13
P_JRIGHT 13
p_jtob 13
p_lcdcontrastdelta 186
p_leave 60
p_linklib 221
p_ln 45
p_loadfilelib 221
p_loadldd 98
p_loadlib 220
p_loadpdd 98
p_locchg 128
p_locdevice 132
p_locreadpdd 133
p_log 45
p_logoff 55
p_logoffa 55
p_logoffx 55
p_logon 55
p_logona 54
p_longtof 48
p_ltob 29
p_marka 164
p_math.h 41
P_MAXSYSIO 101
p_mcancel 174
p_mfree 174
p_minit 173
p_mkdir 138
p_mod 48
p_mreceive 173
p_mreceivew 173
p_msend 174
p_msendreceivea 175
p_msendreceivew 175
p_new 223
p_newlibh 223
P_NINFO 127, 128
p_nmday 109
p_nmdaya 110
p_nmmon 110
p_nmmona 110
p_notify 57

p_notifyerr 58
p_notifyhook 58
p_notifyunhook 58
p_now2str 115
P_NSECDAY 106
p_off 181
p_offrec 35
p_onterminate 54
p_open 92
p_open("TIM:") 105
p_open(P_FDEVICE) 128
p_open(P_FDIR) 133
p_open(P_FFORMAT) 131
p_open(P_FNODE) 127
p_open(P_FSTREAM) 142
p_open(P_FSTREAM_TEXT) 147
p_open(P_FTEXT) 148
p_openlib 221
p_panic 53
p_pcpyfr 167
p_pcpyto 167
p_pcreate 162
p_pfind 166
p_pidfind 166
p_pinrec 36
p_piscpyfr 167
p_pkill 53
p_playsounda 193
p_playsoundcancel 193
p_playsoundw 193
p_pname 165
P_POINT 35, 100
p_pow 46
p_ppanic 54
p_prename 166
p_presume 165
p_print 101
p_printf 101
p_psuspend 165
p_pterminate 53
p_putch 101
p_puts 101
P_PWILD_ANY 124
P_PWILD_EXT 124
P_PWILD_NAME 124
p_qsort 24
P_QUE 25, 26, 27
p_que.h 25
p_rand 46
p_randl 44
p_read 96, 144, 148
p_realloc 68
p_reclass 226
p_reclassbyhandle 226
p_recordsounda 192
p_recordsoundcancel 192
p_recordsoundw 193
P_RECT 35, 100
p_rename 136
p_romversion 177
p_scap 18
p_scat 12
p_scatm 12
p_scmp 19
p_scmpi 19
p_sconf 17
p_scpy 11
p_scpyf 17
p_scpym 12
p_sdate 103
p_seek 144, 149
p_semcrt 84
p_semdel 84
p_send 224
p_setauto 181
p_setautomains 181
p_setbacklight 187
p_setbat 185
p_setctd 180
p_setdefaultpath 126
p_setenv 76
p_setenviron 76
p_setnotify 58
p_setpri 165
p_setpth 126
p_setsnd 188
p_sfstat 138
p_sgadjust 74
p_sgclose 74
p_sgcopyfr 73
p_sgcopyto 73
p_sgcreate 72
p_sgdelete 73
p_sgfind 74
p_sgfree 71
p_sglock 75
p_sgopen 73
p_sgramdisk 71
p_sgsize 74
p_sgunlock 75
p_signal 85
P_SIGNAL_DISABLE 87
P_SIGNAL_ENABLE 87
P_SIGNAL_UNUSED 87
p_signaln 85
p_signalnr 85
p_sin 44
p_skipch 17
p_skipwh 16
p_sleep 104
p_sleepa 105
p_sleept 104
p_slen 11
p_sloc 20
p_sloci 20
p_slocr 20
p_slocri 20
p_smatch 22
p_smatchi 22
p_sound 187
p_sqrt 46
p_srep 13
p_ssub 21
p_ssubi 21
p_st2str 115
p_std.h 5
p_stoa 33
p_stod 42
p_stog 32
p_stogl 33
p_stoi 32
p_stol 32
p_sttods 107
p_supersend 224
p_supply 182
p_supplyinfo 183

p_svecadd 87
p_sveccall 88
p_svecrem 88
p_tan 45
p_testpth 136
p_tickle 164
p_tofold 17
p_tolower 17
p_totalK 71
p_toupper 17
p_unirec 36
p_unloadlib 221
p_unmarka 164
p_version 177
p_wait 85
p_waitstat 86
p_watchall 56
p_weekno 108
p_wkday 108
p_write 97, 144, 149
p_wsupply 184
panic 50
  numbers 50
  object-oriented programming 53
  OLIB 53
  process,by ID 54
  window server 53
PAR: 90
paragraph 65
parallel port 9
parse file specification 123
PDD 89
physical device driver 89
piezo-electric 187
  sound,make 187
PLIB
  C startup module 7
  header files 4
  library 1
plib.h 4
polymorphic 217
power 46
power supply 178, 182
pragma 4
  call 6
  restore 6
  save 6
preemptive scheduling 151, 155
preferences 43
priority
  process 154
process 151
  active,mark 164
  activity,register 164
  control block 152
  creating 160, 162
  current 163
  data segments 64, 65, 167
  data,copy from 167
  data,copy to 167
  find all 166
  find owner 166
  heap 65
  ID 152
  ID,fetch 164
  image,load 160
  image,load asynchronously 162
  indirected string,copy from 167
  messaging 168
  name by ID,fetch 165, 166
  names 155
  non-active,mark 164
  null 164, 180
  on terminate,notify 50
  operations 165
  priorities 154
  priority,get 165
  priority,set 165
  queues 154
  rename 166
  resume 165
  scheduling 79
  shared code segments 158
  state 154
  subsidiary 155
  suspend 104, 160, 165
  suspend until 105
  system 152
  terminate 49, 53, 159
  terminate word 50, 55
  usage count 65
  zero priority 164, 180
processor
  80286 63
  80386 63
  80486 63
  8086 1, 63
  stack 65
program
  environment 2
  small 7
  small model 2, 65
queue
  delta 27, 79, 154, 165
  delta,add entry 27
  delta,remove entry 27
  doubly linked 25
  doubly linked,add entry 26
  doubly linked,remove entry 27
  process 154
  ready 79, 154, 165
  semaphore 79, 154, 165
  time delta 79, 103
quicksort 24
r 157
raise to power 46
RAM
  addressable size in paragraphs 71
  disk,memory used 71
  drive 63
  static 118, 119
  total size in kilobytes 71
random number 44, 46
re-schedule 85
real-time clock 1
record text file access 147
rectangle
  absolute,convert 37
  displace 35
  empty,test 36
  inset 35
  intersection 36
  point inside,test 36
  union 36
rectangle functions 34
reference manuals 8

register based convention 7
registers
segment 3, 63
relative timer 104
REM:: 117
reserved statics 65, 156
reset 152
ROM 2, 63
configuration file 179
DYL 213
masked 118
once programmable 118
system software 1
system tables 14
version 177
ROM:: 117
SYS$CTRY.CFO 179
scheduling
preemptive 151, 155
process 79
scientific functions 44
segments
.$nn 64
.$SC 64
.DYL 64
.LDD 64
.PDD 64
address 64
adjust size 74
available memory 71
close 74
code 2, 158
copy from 73
copy to 73
create 72
data 2
decrease usage count 75
delete 73
device 64
dynamic 64
external memory 3, 71
find by name 74
get size 74
handle 64
increase usage count 75
index table 65
lock 75
memory 64, 71
name 64
open 73
process data 65, 167
registers 3, 63
shared code 158
size 64
unlock 75
usage count 65
semaphores 79, 90
create 84
delete 84
I/O 81, 85
I/O,signal 85
I/O,signal process 85
I/O,wait 86
mutual exclusion 80
primitive functions 84
process scheduling 79
queue 79, 154
serialised access 80
shared resource 80
signal 85
signal,multiple 85
signal,no re-schedule 85
status word 81
wait 85
serial port 9
serialised access 80
server 169
asynchronous message reception 173
functions 172
message reception,cancel 174
message reception,initialise 173
message reception,wait 173
message,free 174
shared code segments 158
shared resource 80
shrinking the heap 66
SIBO 1, 63, 89, 104, 118, 131, 185
architecture 1
sine 44
single-user 151
small model program 2, 65
small program 7
SND: 187
SndFile 188
software interrupts 4, 218
solid state disk 1
sound 187
duration 193
files 188
flags,fetch 188
flags,set 188
loudness 193
piezo-electric 187
play back,asynchronously 193
play back,cancel 193
play back,synchronously 193
record,asynchronously 192
record,cancel 192
record,synchronously 193
Series 3a 188
Series 3a,services 192
sound driver 9
square root 46
SSD 1, 144, 195
drives 118
flash EPROM 118, 119, 145
local,direct read 133
masked ROM 118
once programmable ROM 118
static RAM 118, 119
stack 2, 7, 65
stack based convention 6
status word 81
stray signal 81
stream text file access 146
string 11
arguments,convert 33
capitalise 18
compare 19
compare,case independent 19
concatenate 12
copy 11
copy and fold 17
fold 17
from double,convert 41
functions 11

length of,fetch 11
locate character 20
locate character,case independent 20
locate last match character 20
locate last match character,folded 20
locate sub-string 21
locate sub-string,case independent 21
multiple arguments,convert 31
multiple,concatenate 12
multiple,copy 12
pattern match 22
pattern match,case independent 22
replicate 13
searching 19
to double,convert 42
structures
DbfHeader 198, 199
DbfOpenArgs 200
DbfRecord 196
E_CONFIG 43, 108, 111, 180
E_CPB 162, 163
E_MESSAGE 168
E_PROC 153
E_SUPPLY 182
E_SUPPLY_INFO 183
E_SUPPLY_WARNINGS 184
ImgHeader 158
P_CLASS 211
P_DATE 107, 108, 115
P_DAYSEC 106, 107, 108, 115
P_DELTA 27
P_DINFO 129
P_DTOB 41
P_FPARSE 123
P_INFO 133, 134, 135
P_NINFO 127, 128
P_POINT 35, 100
P_QUE 25, 26, 27
P_RECT 35, 100
SndFile 188
subsidiary process 155
superclass chaining 217
supervisor 168
moving memory 3
suspended process 160, 165
switch off 104, 164, 180
switch on 104, 164, 180
synchronous serial interface 1
SYS$8087.LDD 39
SYS$CTRY.CFO 179
SYS$FSRV 91, 117
SYS$FSRV.$03 152
SYS$MANG 64
SYS$MANG.$02 3, 152
SYS$NULL 64
SYS$NULL.$01 152
SYS$SHLL.$05 152
SYS$WSRV.$04 152
system
addressable RAM in paragraphs 71
asynchronous request 80
auto-switch-off period,set 181
auto-switch-off,allow 181
available segmented memory 71
backlight control,set 187
backlight off 186
backlight on 186
battery type,set 185
country-dependent data,set 180
exit 194
I/O 89
LCD contrast,change 186
memory usage 63, 71
ON key event,disable 182
ON key event,enable 182
panic numbers 50
processes 152
reset 152
ROM 63
services 4
services reference 9
sound flags,set 188
sound,piezo-electric 187
switch off 181
switch-off if mains,disable 181
switch-off if mains,enable 181
tables 14
ticks 151
time 103
time to string,convert 115
time,fetch 103
time,set 103
total RAM in kilobytes 71
system information 177
auto-switch-off period,fetch 181
backlight enablement,fetch 187
battery type,get 185
battery warning 184
country code 178
country-dependent data,fetch 180
display type,fetch 186
keyboard 185
language code 178, 179
last shutdown,cause 177
LCD contrast,get 186
maximum levels 184
o/s data 178
o/s text 179
o/s version 177
power status additional,fetch 183
power status,fetch 182
power supply 182
power supply type 178
ROM version 177
Series 3a,sound 188
sound 187
sound flags,fetch 188
state of keys,fetch 185
switch off 180
switch on 180
switch-off if mains,fetch 181
system wide default path 126
T 157
tangent 45
task 152, 155
terminate
notify 50
notify,all processes 56
notify,cancel 55
notify,request 54, 55
process 49, 53, 159
process,by ID 54
process,this 53
process,unilaterally 53
process,unrecoverable error 53

receive message 54
special notify,cancel 55
word 50, 55
TEXT 5
file,close 148
file,flush 150
file,set end 150
record file access 147
record file,open 148
record file,position 149
record file,read 148
record file,seek 149
record file,write 149
record termination 146
stream file 146
stream file,open 147
ticks 151
TIM: 90
time
class 109
converting representations 106
current to string,convert 115
day to day in week,convert 108
days in month 108
format string 112
P_DATE to P_DAYSEC,convert 108
P_DATE to string,convert 115
P_DAYSEC to P_DATE,convert 107
P_DAYSEC to string,convert 115
P_DAYSEC to system,convert 107
preferences 111
string generation 112
system 103
system to P_DAYSEC,convert 107
system to string,convert 115
system,fetch 103
system,set 103
text form 109
week number 108
time delta queue 79, 103
timer 103
absolute 104
absolute,start 106
asynchronous 105
cancel 106
channel,close 106
channel,open 105
relative 104
relative,start 105
suspend process 104
suspend process until 105
watchdog 3
TopSpeed
C library reference 8
compiler 3
TTY: 90
Turbo C 4
UBYTE 5
UINT 5
ULONG 5
unattended applications 118
unhook notifier interface 58
usage count 65
UWORD 5
variables
environment 63, 75
environment,delete 76, 77
environment,find 77
environment,get value 75, 76
environment,set value 76
magic statics 65
reserved static 65
static,initialised 65
static,uninitialised 65
version 177
volume
label setting 139
w_am 157
w_ws 156
wait handlers 83
activate/deactivate 88
add 87
application 84
attached I/O devices 84
device 84
functions 87
let run 86
polling vs waiting 84
remove 88
watchdog 1, 3
wav2wve.exe 189
wClientData 157
WIMP.DYL 10
window server 168
library 1
panic 53
reference 8
WLIB
library 1, 9
WORD 5
working set 7
wserv_channel 157
zero priority process 164, 180