

I/O DEVICES REFERENCE

Version 2.30

(C) Copyright Psion PLC 1990-97

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, Psion Series 3c, Psion Siena and Psion *Workabout* are trademarks of Psion PLC.

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

Contents

1	Introduction	1-1
	Conventions used in this manual	1-1
2	Console	2-1
	Introduction	2-1
	Automatic opening of a console channel	2-1
	Explicit opening of a console channel	2-3
	P_FSET service call convention	2-3
	Panics	2-3
	Console services	2-3
	Open the console (p_open)	2-3
	Close the console (p_close)	2-4
	Write to the console (p_write)	2-4
	Read a keypress (P_FREAD)	2-5
	Cancel an outstanding read (P_FCANCEL)	2-5
	Test for outstanding keypresses (P_FTEST)	2-5
	Flush keyboard buffer (P_FFLUSH)	2-6
	Edit a string (P_FEDIT)	2-6
	Sense console data (P_FSENSE)	2-6
	Set the console window size (P_FSET, P_SCR_WSET)	2-7
	Scroll the window content (P_SCR_SCROLL)	2-7
	Clear a rectangle (P_FSET, P_SCR_CLR)	2-7
	Position cursor to next line (P_FSET, P_SCR_NEL)	2-8
	Set the cursor position (absolute) (P_FSET, P_SCR_POSA)	2-8
	Set the cursor position (relative) (P_FSET, P_SCR_POSR)	2-8
	Turn the cursor on or off (P_FSET, P_SCR_CURSOR)	2-8
	Set scroll lock (P_FSET, P_SCR_SLOCK)	2-8
	Set auto wrap (P_FSET, P_SCR_WLOCK)	2-8
	Set escape on or off (P_FSET, P_SCR_ESCAPE)	2-9
	Set compatibility on or off (P_FSET, P_SCR_COMPATIBILITY)	2-9
	Set the use of grey on or off (P_FSET, P_SCR_GREY)	2-9
	Additional console services	2-9
	Read an event (P_EVENT_READ)	2-9
	Test for outstanding event (P_EVENT_TEST)	2-10
	Get console data (P_FINQ)	2-10
	Flush window server buffer (P_FWFLUSH)	2-10
	Set the console output rectangle (P_FSET, P_SCR_CSET)	2-10
	Set character attributes (P_FSET, P_SCR_ATTRB)	2-11
	Set the screen font (P_FSET, P_SCR_FONT)	2-11
	Set the last line wrap (P_FSET, P_SCR_LAST_LINE_WRAP)	2-12
	Set window server flushing (P_FSET, P_SCR_FLUSH)	2-12
	Disable reads (P_FSET, P_SCR_DISABLE_READS)	2-12
	Bring to foreground (P_FSET, P_SCR_CLIENT_FOREGROUND)	2-12
	Capture a key (P_FSET, P_SCR_CAPTURE_KEY)	2-13
	Cancel a key capture (P_FSET, P_SCR_CANCEL_CAPTURE_KEY)	2-13
	Example	2-14
3	Parallel Port	3-1
	Introduction	3-1
	Parallel port device names	3-1
	Panics	3-1

Parallel port services	3-1
Open a parallel port (p_open)	3-1
Close a parallel port (p_close)	3-1
Write to a parallel port (P_FWRITE)	3-2
Cancel a write request (P_FCANCEL)	3-2
Sense the input control lines (P_FSENSE)	3-2
Write the output control lines (P_FSET)	3-2
Example	3-3

4 Serial Port	4-1
----------------------	------------

Introduction	4-1
Serial port device names	4-1
Panics	4-1
Serial port parameters	4-1
Baud rate	4-2
Character frame	4-2
Parity	4-3
Handshaking	4-3
Control flags	4-5
Terminator characters	4-5
Serial port errors	4-5
Serial port services	4-6
Open a serial port (p_open)	4-6
Close a serial port (p_close)	4-6
Read from the serial port (P_FREAD)	4-7
Write to the serial port (P_FWRITE)	4-8
Cancel any outstanding requests (P_FCANCEL)	4-8
Sense the serial port characteristics (P_FSENSE)	4-9
Set the serial port characteristics (P_FSET)	4-9
Flush the read buffer (P_FFLUSH)	4-9
Test for received characters (P_FTEST)	4-9
Test and set control lines (P_FCTRL)	4-9
Inquire supported serial characteristics (P_FINQ)	4-10

5 Sound	5-1
----------------	------------

Introduction	5-1
Sound on MC and HC machines	5-1
Sound on Series 3 machines	5-1
Sound on Series 3a machines	5-1
Panics	5-1
Sound services	5-2
Open the sound channel (p_open)	5-2
Close the sound channel (p_close)	5-2
Cancel a write request (P_FCANCEL)	5-2
Sense sound characteristics (P_FSENSE)	5-2
Set sound characteristics (P_FSET)	5-2
Write alarm note sequence (E_FALARM)	5-3
HC, MC and Series 3a additional sound service	5-3
Write to voice n (E_FSSOUNDCHANNELn)	5-3
Series 3 and Series 3a additional sound service	5-4
Write DTMF dial tones (E_FDIAL)	5-4
Example	5-5

6 The Alarm Device Driver	6-1
Introduction	6-1
Panics	6-2
Series 3, Series 3a and MC alarm services	6-2
Open the alarm channel (p_open)	6-2
Close the alarm channel (p_close)	6-2
Cancel an alarm request (P_FCANCEL)	6-2
Queue a timed alarm (A_FTIMED)	6-2
Queue an untimed alarm (A_FUNTIMED)	6-3
Series 3a additional alarm services	6-3
Queue a Series 3a timed alarm (A_FTIMED_X)	6-3
Queue a Series 3a untimed alarm (A_FUNTIMED_X)	6-4
7 The Free-Running Counter	7-1
Introduction	7-1
FRC services	7-1
Open the FRC channel (p_open)	7-1
Close the FRC channel (p_close)	7-2
Cancel the FRC request (P_FCANCEL)	7-2
Start the free-running counter (A_FTIMED)	7-2
Read the elapsed time in E_FRC_COUNTING mode (P_FREAD)	7-2
Read the elapsed time in E_FRC_REPEATING mode (P_FREAD)	7-3
8 The Series 3 World Database	8-1
Introduction	8-1
Mode	8-1
Series 3 and Series 3a World database services	8-2
Open the World channel (p_open)	8-2
Close the World channel (p_close)	8-2
Cancel a World request (P_FCANCEL)	8-2
Find by city (WR_FIND_CITY)	8-2
Find by country (WR_FIND_COUNTRY)	8-2
Find by city and country (WR_FIND_EXACT)	8-3
Find next city (WR_NEXT)	8-3
Find previous city (WR_BACK)	8-3
Find home city (WR_GET_HOME)	8-3
Set home city (WR_SET_HOME)	8-3
Find default country (WR_GET_DEFAULT_COUNTRY)	8-4
Set default country (WR_SET_DEFAULT_COUNTRY)	8-4
Get dial string (WR_GET_DIAL_STRING)	8-4
Open file for additional data (WR_SET_EXTRA)	8-5
Modify additional data (WR_EXTRA)	8-5
Read city data (WR_GET_CITY_DATA)	8-6
Read city data (WR_GET_COUNTRY_DATA)	8-7
Calculate distance, sunrise and sunset (WR_CALC)	8-8
Read next city name (WR_NEXT_LOCK)	8-9
World file types and their locations	8-9
Main World file	8-9
World Extension file	8-9
World File format	8-9
World Extension File format	8-10
File header	8-10
Data block	8-10

9 Xmodem and Ymodem	9-1
Introduction	9-1
Data transfer protocols overview	9-1
One byte checksum	9-1
Two byte Cyclic Redundancy Check (CRC)	9-1
The Xmodem protocol	9-2
Link establishment	9-2
The data transfer phase	9-2
Link termination	9-2
Checksum data flow showing error recovery	9-3
The CRC variant	9-3
CRC data flow showing error recovery	9-3
The 1K variant	9-3
The 1K option data flow	9-4
Abandoning a transfer	9-4
The Ymodem protocol	9-4
Link establishment	9-5
The data transfer phase	9-5
Link termination	9-5
Ymodem file transfer data flow	9-6
The 1K variant	9-6
The G variant	9-6
Ymodem-G file transfer data flow	9-7
Abandoning a transfer	9-7
Protocol problems	9-7
Xmodem/Ymodem services	9-8
Open an Xmodem/Ymodem channel (p_open)	9-8
Close the Xmodem channel (p_close)	9-8
Connect to the remote computer (P_FCONNECT)	9-8
Disconnect from the remote computer (P_FDISCONNECT)	9-10
Read data from the remote computer (P_FREAD)	9-11
Write data to the remote computer (P_FWRITE)	9-12
10 NCP and Link	10-1
Introduction	10-1
Panics	10-1
The Psion logical link layer protocol	10-1
The SYS\$NCP process	10-2
Connection establishment	10-2
Data transfer	10-2
The LINK process	10-3
NCP services	10-3
Open an NCP channel (p_open)	10-3
Close the NCP channel (p_close)	10-4
Connect to a remote process (P_FCONNECT)	10-4
Disconnect from the remote process (P_FDISCONNECT)	10-6
Read data from the remote process (P_FREAD)	10-6
Write data to the remote process (P_FWRITE)	10-7
Cancel any outstanding request (P_FCANCEL)	10-7
Read supervisory information (P_FRSUPER)	10-7
Respond to a supervisory message (P_FINQ)	10-9
Sense the current channel activity (P_FSENSE)	10-10
Request the SYS\$NCP terminate (P_FSTOP)	10-10
Example	10-10

11 Cradle and Docking Station

Introduction	11-1
Cradle/Docking Station services	11-1
Open the device (p_open)	11-1
Close the channel (p_close)	11-2
Read from the device (P_FREAD)	11-2
Cancel a read (P_FCANCEL)	11-2
Set the device type (P_FSET)	11-2
Sense the device type (P_FSENSE)	11-2

12 HC Magnetic Card Reader **12-1**

Introduction	12-1
Open the MCR device (p_open)	12-1
Close the channel (p_close)	12-1
Read from the MCR device (P_FREAD)	12-1
Cancel a read (P_FCANCEL)	12-2
Set the pull-up resistors (P_FSET)	12-2

13 HC Bar Code Reader **13-1**

Hardware Description	13-1
The Bar code reader interface module	13-1
The bar code reader wand.	13-1
Device drivers	13-1
Bar code driver services	13-2
Open the bar code device (p_open)	13-2
Close the channel (p_close)	13-2
Read from the MCR device (P_FREAD)	13-2
Cancel a read (P_FCANCEL)	13-2

14 HC Intelligent Bar Code Reader/RS232 Port **14-1**

RS232/intelligent bar code reader module	14-1
The RS232 serial port interface	14-2
Power consumption	14-2
Powering the HC from an external power source	14-3
DSR auto wakeup switch	14-3
The bar code interface	14-3
Power consumption	14-4
Powering a bar code wand from the HC	14-4
Bar code symbolologies	14-4
Code 128	14-4
Codabar	14-4
Interleaved 2 of 5	14-5
Code 39	14-5
The UPC/EAN bar code formats	14-6
UPC E	14-6
UPC E + 2 digits	14-7
UPC E + 5 digits	14-7
EAN 8 bar code format	14-8
EAN 8 + 2 digits	14-8
EAN 8 + 5 digits	14-8
EAN 13	14-9
EAN 13 + 2 digits	14-9
EAN 13 + 5 digits	14-10
UPC A	14-10
UPC A + 2 digits	14-11
UPC A + 5 digits	14-11

Bar code commands	14-11
Multiple options in a command	14-12
Issuing multiple commands	14-12
Serial intercharacter delay	14-13
Hard reset	14-13
Select bar code symbology	14-13
Check character options	14-14
Decoding options	14-14
Single read mode	14-15
Single read control	14-15
Set Interleaved 2 of 5 length	14-15
Set termination string	14-16
Code ID characters	14-16
Status request	14-17
Scanner enable	14-17
RS232 port/bar code driver services	14-17
Open the device (p_open)	14-17
Close the channel (p_close)	14-18
Sense serial port characteristics (P_FSENSE)	14-18
Set serial port characteristics (P_FSET)	14-18
Read from device (P_FREAD)	14-18
Write to device (P_FWRITE)	14-19
An example program	14-20

15 Introduction to Psion Infrared Communications	15-1
---	-------------

About this chapter	15-1
The IrDA protocol layer model	15-1
Introduction to Psion infrared communications	15-2
The Psion protocol layer model	15-2
The physical layer	15-2
Port geometry	15-2
Data transfer rate	15-3
Data link layer	15-3
Primary and secondary stations	15-4
IrLAP services	15-4
The network layer	15-4
IAS application logging	15-4
Discovery	15-5
Multiplexing	15-6
Link Control	15-6
The Psion IR Communications application	15-6
Third party applications	15-6
System resources	15-6
Keypresses	15-6
The IR printer port device driver	15-7

16 The AccessIr API	16-1
----------------------------	-------------

Using the AccessIr API	16-1
Prerequisites	16-1
Introduction to using the AccessIr API	16-1
Initialising the IR protocol stack	16-2
Opening and closing a channel to the IR device	16-2
Discovery	16-2
Selection	16-2
Connection	16-3
Sending and accepting data	16-3
Disconnection	16-3
Constants	16-3

Opening and closing a channel to the IR device	16-3
Open a channel to the IR device (p_open)	16-3
Close a channel to the IR device (p_close)	16-4
Discovery, selection and connection	16-4
Discover IR enabled machines (P_FIRDISCOVER)	16-4
Select remote machine to connect to (P_FIRSELECT)	16-4
Connect to selected machine (P_FIRMAKECONNECT)	16-5
Wait for remote connect (primary) (P_FIRAWAITCONNECT)	16-5
Send and accept data	16-6
Accept data from selected remote machine (P_FREAD)	16-6
Send data to the selected remote machine (P_FWRITE)	16-6
Disconnection	16-6
Disconnect from remote machine (P_FIRDISCONNECT)	16-6
Example application	16-7
17 The IrMUX API	17-1
Using the IrMUX API	17-1
Prerequisites	17-1
Introduction to using the IrMUX API	17-1
Initialising the IR protocol stack	17-1
Logging on to and logging off from the IrMUX server	17-2
Registering/unregistering applications with the LM-IAS server	17-2
LM-IAS services	17-2
The IAS Get Value By Class message frame	17-3
The IAS Get Value By Class reply frame	17-3
Connectionless calls	17-3
Reading and writing data	17-3
Connection-oriented calls	17-3
Discovery	17-4
Connection - first time	17-4
Disconnecting - first time	17-4
Connection - second time	17-4
Reading and writing data	17-4
Disconnecting - second time	17-4
Using Exclusive mode	17-4
The IrMUX API	17-5
IrMUX message format	17-5
Log on to the IrMUX server (LM_Logon)	17-5
Log off from the IrMUX server (LM_Logoff)	17-5
Register a port number with the LM-IAS server (LM_RegisterPort)	17-5
Free registered port with LM-IAS server (LM_UnRegisterPort)	17-6
Queue a connectionless read request (LM_CLReadRequest)	17-6
Queue a connectionless write request (LM_CLWriteRequest)	17-7
Return info on in-range machines (LM_DiscoverDevicesRequest)	17-7
Attempt to connect to a remote machine (LM_ConnectRequest)	17-7
Wait for remote machine to connect (LM_WaitForConnection)	17-8
Return the status of the link (LM_StatusRequest)	17-9
Queue a read request on a connection (LM_ReadRequest)	17-9
Queue a write request on a connection (LM_WriteRequest)	17-10
Queue an unreliable read request (LM_UReadRequest)	17-10
Queue an unreliable write request (LM_UWriteRequest)	17-11
Obtain/release exclusive access (LM_AccessModeRequest)	17-11
Place the connection into Idle/Active mode (LM_IdleRequest)	17-12
Set retries on each data frame (LM_SetHandshakingLevel)	17-12
Disconnect (LM_DisconnectRequest)	17-13

18 Fast Charger

Introduction	18-1
Docking Station services	18-1
HC/Workabout docking station fast charger services	18-1
Fast charging batteries	18-2
Measuring battery capacity	18-3
Rated charge capacity of standard Psion battery packs	18-3
Example calculations	18-3
Open the fast charger device (p_open)	18-3
Close the channel (p_close)	18-4
Set the battery charge mode (FCHG_SETCHARGEMODE)	18-4
Charging Psion battery packs	18-4
Charging custom battery packs	18-5
Read battery charge mode (FCHG_READCHARGEMODE)	18-6
Read the battery status (FCHG_READSTATUS)	18-6
Read the battery status asynchronously (FCHG_ASYNCREAD)	18-7
Cancel an asynchronous read (FCHG_CANCEL)	18-7
Fast charge the main battery (FCHG_FASTCHARGE1)	18-7
Fast charge the spare battery (FCHG_FASTCHARGE2)	18-7
Discharge the main battery (FCHG_DISCHARGE1)	18-8
Example program	18-8

CHAPTER 1

INTRODUCTION

This manual describes the I/O device drivers that have been written by Psion, except for the files device driver and the asynchronous timer device driver which are described in the *Files* and the *Time, Timers and Dates* chapters of the *PLIB Reference* manual.

Before using any of the devices described in this manual the reader should be familiar with the contents of the *Asynchronous Requests and Semaphores* and the *I/O System* chapters of the *PLIB Reference* manual.

Conventions used in this manual

I/O devices support a number of services, each specified by a function number of the form `P_Fxxxx` (defined in *p_file.h*). For example, to write to a channel opened with a control block at `*pcb`, you may call:

```
p_ioc(pcb,P_FWRITE,additional parameters);
```

The notation `p_ioc(P_FWRITE)` or, more simply, `P_FWRITE` is used to refer to this function call.

All I/O function requests are asynchronous in principle. In practice, however, many I/O requests are implemented synchronously, that is, the I/O operation will complete before the service call returns. For example, the `P_FCLOSE` service is always implemented synchronously but the `P_FREAD` service is commonly implemented asynchronously.

The description of a service that is implemented synchronously gives either the specific synchronous call (such as `p_open` or `p_close`) or the `p_iow` service call prototype. Services implemented synchronously are prototyped using only the `p_iow` I/O primitive, since there is no advantage in calling them asynchronously.

A service that is implemented asynchronously may be called synchronously or asynchronously. The description gives both the synchronous call (as described above) and the asynchronous `p_ioc` service call prototype. Such a service may be called using the `p_iow`, `p_ioc` or `p_ioa` primitives. Note that the use of `p_ioc` is in almost all cases preferable to the use of `p_ioa` (since the former will complete even in the event of an error).

Note that smaller code will be generated by use of the primitives:

```
p_iow2, p_iow3, p_iow4, p_iow5,  
p_ioc3, p_ioc4, p_ioc5,  
p_ioa3, p_ioa4, p_ioa5.
```

Again the use of, for example, `p_ioc3` is in almost all cases preferable to the use of `p_ioa3`. The same statement applies to the other two variants of each function.

CHAPTER 2

CONSOLE

Introduction

The console (CON:) device provides a basic set of screen and keyboard services, suitable for use by relatively simple character-based C application programs. OPL programs automatically open a console channel to provide support for the screen display and keyboard commands.

The console device is implemented differently on different machines in the SIBO range:

- on the HC and Series 3 machines the console device connects directly to the window server.
- on the MC 200 and MC 400 machines the console device is implemented by an intermediate SYS\$CONS display process.

There are also some differences in the services that are available on different machines. These differences are stated in the description of the particular service to which they apply.

Some of the console functions use the P_POINT, P_RECT and P_RECTP structures. They are defined in *p_graf.h* (which is included by *p_cons.h*) as:

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

typedef struct
{
    P_POINT tl; /* top left coordinates */
    P_POINT br; /* bottom right coordinates */
} P_RECT;

typedef struct
{
    P_RECT r; /* rectangle */
    P_POINT p; /* point */
} P_RECTP;
```

Co-ordinates are measured from the top left corner of the window (0, 0), and increment to the right (x) and down (y). Co-ordinates are measured in character units.

When rectangles are specified the top left point is included in the rectangle and the bottom right corner is just outside the rectangle. If a rectangle extends outside the console window then it is clipped to fit inside the window (this feature may be used to save sensing the size of the window when a scroll or clear is required up to the right/bottom edge of the window).

Versions of EPOC prior to version 2.32 do not support the opening of a console device when the application program's total memory usage exceeds 32k.

Automatic opening of a console channel

A channel to the console device is opened automatically during the start-up initialisation of a C program built with the CLIB start-up module. This does not occur in the case of programs built with the PLIB start-up module.

In PLIB programs, however, the console (if not already open) is opened automatically by the first use of one of the PLIB simple console I/O functions (`p_getch`, `p_printf`, etc.) described in the *I/O System* chapter of the *PLIB Reference* manual.

The console channel handle of an automatically opened console device channel is stored in the pre-defined static `winHandle`.

An automatically opened console is set to a size appropriate for the SIBO machine on which the application is running. In a PLIB program you may, however, specify the window size of an automatically opened console by setting the global `P_RECT` structure `_DefScreenRect`, as follows:

```
GLDEF_D _DefScreenRect;

....
_DefScreenRect.tl.x=0;
_DefScreenRect.tl.y=0;
_DefScreenRect.tl.x=40; /* 40 columns */
_DefScreenRect.tl.x=8; /* 8 rows */
...
```

An automatically opened console is also set to the native mode of the machine. On the Series 3a machine the console is opened in non-compatibility, with access to grey. In a PLIB program you may modify the console mode by setting the global variable `_DefScreenMode`. The possible modes that may be set 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 as follows:

```
GLDEF_D INT _DefScreenMode;

....
_DefScreenMode=1;
...
```

Values that are not relevant to a particular type of machine are simply ignored.

If used, the values of `_DefScreenRect` and `_DefScreenMode` must be set before the first usage of any function that causes the console to be opened. Ignore the 'duplication' message generated by the linker when this code is included in your application.

In CLIB programs you can prevent the automatic opening of a console channel by defining the function `p_xwind` in your code, as in the following example:

```
extern void *winHandle;

void p_xwind(void)
{
    winHandle=(void *)1;
}

int main(void)
{
    ...

    return(0);
}
```

You should ignore the warning, given during the linking of your program, that the symbol `_p_xwind` is duplicated.

If you use this technique, your CLIB program should not, of course, make any reference to `stdin`, `stdout` or `stderr`, unless you have redirected them. Setting `winHandle` to 1 (an illegal value for a handle) will guarantee that any such reference will fail with a panic.

Explicit opening of a console channel

In a PLIB program you may, if you wish, explicitly open a console channel by means of `p_open`. The console channel handle must be stored in the pre-defined static `winHandle` that is used for an automatically opened console channel. For example:

```
GLREF_D VOID *winHandle;

p_open(&winHandle, "CON:", -1);
```

This is particularly important if your program uses any of the PLIB simple console I/O functions. Otherwise you may inadvertently attempt to open two console channels from the same process and your process will be panicked.

After explicitly opening the console and before making any other use of its services you must make it visible by setting its size with the `P_FSET` service, using the `P_SCR_WSET` function code.

Compatibility mode and the availability of grey can be set by means of the `P_FSET` service, using the `P_SCR_COMPATIBILITY` and `P_SCR_GREY` function codes.

P_FSET service call convention

The `P_FSET` function provides a number of services, determined by a function code. The function code is assigned to a `UWORD` whose address is passed as the third parameter to the `p_iow` function. The convention:

```
INT p_iow(VOID *pcb, P_FSET, &P_SCR_CLR, P_RECT *prect);
```

is used to denote this. It should be interpreted as:

```
UWORD function;

function = P_SCR_CLR;
p_iow(pcb, P_FSET, &function, prect);
```

or equivalent.

Panics

All services (with the exception of `p_open`) will cause the calling process to be panicked if the passed channel handle is not valid. Other panics are described under the particular service to which they apply.

Console services

p_open(CON:)

Open the console

```
INT p_open(VOID **ppcb, CON:, -1);
```

Open a channel to the console device, writing the channel handle to `*ppcb` if successful. The console window is not made visible until its size has been set by the `P_FSET` service with a function code of `P_SCR_WSET`.

When first opened the console properties are:

- auto wrap on
- auto flushing on
- scroll lock off
- cursor display off
- pause key sequence enabled
- exit key sequence enabled (escape on)
- last line wrap (not available on MC) off

It is recommended that `ppcb` should be the address of the pre-defined static `winHandle`, as in the following example:

```
GLREF_D VOID *winHandle;

LOCAL_D UWORD func;
LOCAL_D INT err;

if ((err=p_open(&winHandle, "CON:", -1))!=0)
{
    fail:
    p_notifyerr(err, "Failed to open console", NULL, NULL, NULL);
    p_exit(0);
}
/* set the screen size */
rect.tl.x = rect.tl.y = 0;
rect.br.x = 25;
rect.br.y = 9;
func = P_SCR_WSET;
if ((err=p_iow4(winHandle, P_FSET, &func, &rect))!=0)
{
    p_close(winHandle);
    goto fail;
}
```

You should not open the console yourself if your program is built with the CLIB start-up module.

The calling process is panicked if it already has an open channel to the console.

Returns zero if successful, or the negative error `E_FILE_ALLOC` if it failed to allocate memory for the channel control block. In versions of EPOC prior to version 2.32, `p_open("CON:")` will fail (and return a negative number) if the total application memory usage exceeds 32k.

p_close Close the console

```
INT p_close(VOID *pcb);
```

Close the channel to the console.

Returns zero.

p_write Write to the console

```
INT p_write(VOID *pcb, UBYTE *buf, UWORD len);
```

Write `len` bytes of data from `buf` to the screen, starting at the current column (x coordinate) in the current line (y coordinate). The value of `len` must be less than or equal to 255.

The request will always return immediately (but the text may not appear immediately if auto flushing is turned off) and there is no advantage in calling this service in any way other than with `p_write`.

Characters are printed directly on the screen unless they are one of:

BELL	sound the bell (buzzer)
TAB	go to the next tab stop (tabs are every 8 characters)
BS	backspace
CR	move the cursor to the beginning of the current line
LF, VT	move the cursor down one line
FF	move the cursor down one screen

If there are more characters in `buf` than will fit on the current line, the writing of the additional characters depends on the auto wrap and scroll states as set by the `P_FSET` service with function codes `P_SCR_WLOCK` and `P_SCR_SLOCK`:

- if auto wrap is on then any additional characters in `buf` are written from character position zero in the following line. If this following line is off the screen then the screen will first be scrolled up by one line, provided scroll lock is off.
- if auto wrap is off then the additional characters in `buf` successively overwrite the last character in the line. On completion of the write request the last character in the row will therefore be the final character in `buf`. This assumes, of course, that there are no cursor movement characters in `buf`.

The request cannot fail and so the function call always returns zero.

P_FREAD

Read a keypress

```
INT p_iow(VOID *pcb, P_FREAD, P_CON_KBREC *kbrec);_VOID p_ioc(VOID *pcb, P_FREAD, WORD
*pstat, P_CON_KBREC *kbrec);
```

RRead a keypress record into the `P_CON_KBREC` struct pointed to by `kbrec`. The `P_CON_KBREC` struct is defined in `p_cons.h` as:

```
typedef struct
{
    UWORD keycode;    /* Code for the key pressed */
    UBYTE modifiers;  /* State shift keys etc */
    UBYTE count;      /* Used to accumulate auto-repeat counts */
} P_CON_KBREC;
```

The content of `modifiers` is a set of bit flags indicating which modifier keys were held down when a key was pressed:

<code>W_SHIFT_MODIFIER</code>	SHIFT key down
<code>W_CTRL_MODIFIER</code>	CONTROL key down
<code>W_PSION_MODIFIER</code>	PSION key down
<code>W_CAPS_MODIFIER</code>	Caps lock on
<code>W_NUM_LOCK_MODIFIER</code>	Num lock on

The content of `keycode` for a standard keypress is simply its ASCII character code. There are also many special values that the `keycode` can provide, corresponding to particular keypresses or keypress combinations. These are fully described in the *Events* chapter of the *Window Server Reference* manual.

A `P_FREAD` request will not complete until either there is an outstanding keypress to read or the request is cancelled. A synchronous request may therefore take an indefinitely long time to return. A prior `p_iow(P_FTEST)` call will determine if there is an outstanding keypress.

The calling process is panicked if `pcb` is not a valid channel handle, or if there is an outstanding `P_FREAD` (or `P_EVENT_READ`) request.

The completion status code is returned by the synchronous `p_iow(P_FREAD)` and written to `*pstat` by asynchronous calls. The completion status code is zero if the `P_FREAD` request completed successfully, or the negative error `E_FILE_CANCEL` if the request was cancelled by a `P_FCANCEL` request.

P_FCANCEL

Cancel an outstanding read

```
INT p_iow(VOID *pcb, P_FCANCEL);
```

Cancel an outstanding read; the request is harmless if there are no reads outstanding.

The request cannot fail and always returns zero.

P_FTEST

Test for outstanding keypresses

```
INT p_iow(VOID *pcb, P_FTEST, UWORD *pflag);
```

Set `*pflag` to `TRUE` if there are any keypresses outstanding, otherwise set it to `FALSE`.

The request cannot fail and always returns zero.

P_FFLUSH**Flush keyboard buffer**

```
INT p_iow(VOID *pcb, P_FFLUSH);
```

Flush the keyboard buffer, all key presses held in the buffer are discarded.

The request cannot fail and always returns zero.

P_FEDIT**Edit a string**

```
INT p_iow(VOID *pcb, P_FEDIT, P_CEDIT *cedit, UWORD *plen);
```

Edit a string. The P_CEDIT struct is defined in *p_screen.h* as:

```
typedef struct
{
    UBYTE cursorpos;    /* not used */
    UBYTE trap;         /* trap input errors or not */
    UBYTE string[P_MAXEDITSTR]; /* string to edit */
} P_CEDIT;
```

Allows the user to edit the text in *string*. The cursor is initially positioned at the end of the string. The user is not allowed to expand the string beyond **plen* characters (*string* must be initially not more than **plen* characters in length). Since **plen* does not include the terminating zero, its value must be *less* than P_MAXEDITSTR (256).

Pressing the left or right arrow keys moves the cursor one character to the left or right in the string. The HOME and END keys move the cursor to the start and end of the string.

Pressing ESC clears the content of the string. The value of *trap* determines what happens if ESC is pressed when the string is already clear. If *trap* is TRUE, then the editing operation will complete with *string* unchanged, if *trap* is FALSE then nothing will happen.

Returns zero, or E_FILE_CANCEL if editing is terminated by pressing ESC.

P_FSENSE**Sense console data**

```
INT p_iow(VOID *pcb, P_FSENSE, P_RECTP *prectp);
```

Write the cursor position to (*prectp->p.x*, *prectp->p.y*) and write the window co-ordinates to *prectp->r*. The window co-ordinates are those specified by the most recent prior call to P_FSET with function code P_SCR_WSET or P_SCR_CSET.

For example:

```
LOCAL_C VOID SenseConsole(VOID)
{
    P_RECTP rectp;
    WORD x_pos, y_pos, width, height;

    p_iow(winHandle, P_FSENSE, &rectp);
    x_pos = rectp.p.x;
    y_pos = rectp.p.y;
    width = rectp.r.br.x - rectp.r.tl.x;
    height = rectp.r.br.y - rectp.r.tl.y;
    ...
    /* do something with the coordinates here */
}
```

The request cannot fail and always returns zero.

P_FSET, P_SCR_WSET**Set the console window size**

```
INT p_iow(VOID *pcb, P_FSET, &P_SCR_WSET, P_RECT *prect);
```

Set the console window size. A `P_RECT` structure is used for historical reasons; in practice `prect->rect.tl` should always be (0,0). For example, to set the console size to 9 lines high by 25 columns wide (the full screen size for HC machines) you should use:

```
P_RECT rect;
UWORD func;
...
rect.tl.x = rect.tl.y = 0;
rect.br.x = 25;
rect.br.y = 9;
func = P_SCR_WSET;
p_iow(winHandle, P_FSET, &func, &rect);
```

Returns zero if successful, otherwise the error `E_GEN_NOMEMORY` if unable to create a window of the new size.

P_FSET, P_SCR_SCROLL**Scroll the window content**

```
INT p_iow(VOID *pcb, P_FSET, &P_SCR_SCROLL, P_RECTP *pscr);
```

Scroll the rectangle `pscr->r` by the vector amount (`pscr->p.x`, `pscr->p.y`). The area left behind the trailing edge(s) is cleared. Scrolling a rectangle is not affected by setting or clearing the scroll lock.

`pscr->r` describes the rectangle to scroll;
`pscr->p.x` gives the horizontal distance to scroll (negative left, positive right);
`pscr->p.y` gives the vertical distance to scroll (negative up, positive down).

For example:

```
LOCAL_C VOID scroll(WORD amnt_v, WORD amnt_h)
/* scroll the whole window vertically amnt_v and horizontally amnt_h */
{
    UWORD func;
    P_RECTP scl;

    p_iow(winHandle, P_FSENSE, &scl);
    scl.r.br.x -= scl.r.tl.x;
    scl.r.br.y -= scl.r.tl.y;
    scl.r.tl.x = 0;
    scl.r.tl.y = 0;
    scl.p.x = amnt_h;
    scl.p.y = amnt_v;
    func = P_SCR_SCROLL;
    p_iow(winHandle, P_FSET, &func, &scl);
}
```

Returns zero.

P_FSET, P_SCR_CLR**Clear a rectangle**

```
INT p_iow(VOID *pcb, P_FSET, &P_SCR_CLR, P_RECT *prect);
```

Clear the rectangle `*prect`. If `*prect` extends outside the console window then it is clipped so that only the part that intersects with the window is cleared.

For example:

```
LOCAL_C VOID clear(P_RECT *prect)
{
    INT func;

    func = P_SCR_CLR;
    p_iow(winHandle, P_FSET, &func, prect);
}
```

Returns zero.

P_FSET, P_SCR_NEL **Position cursor to next line**

```
INT p_iow(VOID *pcb, P_FSET, &P_SCR_NEL);
```

Move the cursor position to the first column on the next line. If the current row is the bottom row then, provided scroll lock is not set, the window image scrolls up.

Returns zero.

P_FSET, P_SCR_POSA **Set the cursor position (absolute)**

```
INT p_iow(VOID *pcb, P_FSET, &P_SCR_POSA, P_POINT *ppoint);
```

Position the cursor to the point *ppoint. If *ppoint is outside the window the cursor moves to the the position in the window closest to *ppoint.

Returns zero.

P_FSET, P_SCR_POSR **Set the cursor position (relative)**

```
INT p_iow(VOID *pcb, P_FSET, &P_SCR_POSR, P_POINT *ppoint);
```

Position the cursor by adding the displacement *ppoint to its current co-ordinates. If *ppoint is outside the window the cursor moves to the the position in the window closest to *ppoint.

Returns zero.

P_FSET, P_SCR_CURSOR **Turn the cursor on or off**

```
INT p_iow(VOID *pcb, P_FSET, &P_SCR_CURSOR, UWORD *pflag);
```

Turn the cursor off if *pflag is FALSE, turn it on if *pflag is TRUE.

Turning the cursor on or off has no affect on the screen driver output other than enabling or disabling the display of the cursor.

Returns zero.

P_FSET, P_SCR_SLOCK **Set scroll lock**

```
INT p_iow(VOID *pcb, P_FSET, &P_SCR_SLOCK, UWORD *pflag);
```

Turn scroll lock off if *pflag is FALSE, turn it on if *pflag is TRUE.

When scroll lock is on scrolling of the console (caused by either P_SCR_NEL or p_write) is disabled. It has no effect on scrolling by means of P_FSET with function code P_SCR_SCROLL.

Returns zero.

P_FSET, P_SCR_WLOCK **Set auto wrap**

```
INT p_iow(VOID *pcb, P_FSET, &P_SCR_WLOCK, UWORD *pflag);
```

Turn auto wrap off if *pflag is FALSE, turn it on if *pflag is TRUE.

If auto wrap is off then character printing by p_write stops at the right margin, with further characters successively overwriting the last character in the line. If auto wrap is on then trying to write a character beyond the end of the line causes the cursor to move to the start of the next line. If the cursor is in the last line of the window (and scroll lock is off) this will cause a scroll.

Returns zero.

P_FSET, P_SCR_ESCAPE**Set escape on or off**

```
INT p_iow(VOID *pcb, P_FSET, &P_SCR_ESCAPE, UWORD *pflag);
```

Turn escape handling off if *pflag is FALSE, turn it on if *pflag is TRUE.

If escape handling is off then the user will not be able to use the exit key sequence to exit the console application. The exit key sequence is PSION-ESC on HC and Series 3 machines, PSION-E on (English-language) MC machines (PSION-? on foreign language MCs, where ? is a character that depends on the language). Additionally, on MC machines, the STOP menu button will be disabled when escape is off.

Escape handling is initially on.

Returns zero.

P_FSET, P_SCR_COMPATIBILITY**Set compatibility on or off**

```
INT p_iow(VOID *pcb, P_FSET, &P_SCR_COMPATIBILITY, UWORD *pflag);
```

This function code is only available on machines that contain version 4, or later, of the window server.

Turn compatibility off if *pflag is FALSE, turn it on if *pflag is TRUE.

If compatibility is turned on, all drawing to the screen of the Series 3a is performed in double pixel mode.

On Series 3a machines, compatibility mode is initially on.

Returns zero.

Note: A TRUE value of *pflag *must* be the value 1. Any other non-zero value may be interpreted differently on future machines.

P_FSET, P_SCR_GREY**Set the use of grey on or off**

```
INT p_iow(VOID *pcb, P_FSET, &P_SCR_GREY, UWORD *pflag);
```

This function code is only available on machines that contain version 4, or later, of the window server.

Turn the use of grey off if *pflag is FALSE, turn it on if *pflag is TRUE.

On Series 3a machines, the use of grey is initially off.

Returns zero.

Additional console services

The following services have been implemented to support OPL/g on HC and Series 3 machines. *They are not available on machines in the MC range.*

If your application is sufficiently complex to require these services it is recommended that you consider writing it as a window server application rather than using the console device.

P_EVENT_READ**Read an event**

```
INT p_iow(VOID *pcb, P_EVENT_READ, P_CON_KBREC *kbrec);
VOID p_ioc(VOID *pcb, P_EVENT_READ, WORD *pstat, P_CON_KBREC *kbrec);
```

RRead an event record (including keypress events) into the P_CON_KBREC struct pointed to by kbrec. The P_CON_KBREC struct is defined in *p_cons.h* as:

```
typedef struct
{
    UWORD keycode;    /* Code for the key pressed */
    UBYTE modifiers; /* State shift keys etc */
    UBYTE count;     /* Used to accumulate auto-repeat counts */
} P_CON_KBREC;
```

If the event was a keypress (a WM_KEY event) the result is exactly as for a P_FREAD request, described earlier.

Other window server events result in one of the following values being written to keycode:

CONS_EVENT_FOREGROUND	received a WM_FOREGROUND event
CONS_EVENT_BACKGROUND	received a WM_BACKGROUND event
CONS_EVENT_ON_OFF	received a WM_ON event
CONS_EVENT_COMMAND	received a WM_COMMAND event

Future versions of the console device may report additional event types. Applications should be written to ignore event types other than those listed above.

A P_EVENT_READ request will not complete until either an event is received or the request is cancelled. A synchronous request may therefore take an indefinitely long time to return. A prior p_iow(P_EVENT_TEST) call will determine if there is an outstanding event to be read.

The calling process is panicked if pcb is not a valid channel handle, or if there is an outstanding P_EVENT_READ (or P_FREAD) request.

The completion status code is returned by the synchronous p_iow(P_EVENT_READ) and written to *pstat by asynchronous calls. The completion status code is zero if the P_EVENT_READ request completed successfully, or the negative error E_FILE_CANCEL if the request was cancelled by a P_FCANCEL request.

P_EVENT_TEST **Test for outstanding event**

```
INT p_iow(VOID *pcb, P_EVENT_TEST, UWORD *pflag);
```

Set *pflag to TRUE if there are any events (including keypress events) outstanding, otherwise set it to FALSE.

The request cannot fail and always returns zero.

P_FINQ **Get console data**

```
INT p_iow(VOID *pcb, P_FINQ, CONSOLE_INFO *pinfo);
```

Return information about the window server resources used by the console. CONSOLE_INFO is defined in *p_cons.h* as:

```
typedef struct
{
    UINT window_handle; /* window server id of the console window */
    UINT font_handle;   /* window server id of the console font */
    UINT line_height;   /* pixel height of a console line - font height + leading */
    UINT char_width;    /* pixel width of a monospaced console character */
} CONSOLE_INFO;
```

Returns zero.

P_FWFLUSH **Flush window server buffer**

```
INT p_iow(VOID *pcb, P_FWFLUSH);
```

Flush the commands buffered to the window server - has no effect when auto flushing is on.

Returns zero.

P_FSET, P_SCR_CSET **Set the console output rectangle**

```
INT p_iow(VOID *pcb, P_FSET, &P_SCR_CSET, P_RECT *prect);
```

Set the console output rectangle. All subsequent output is restricted (and wrapped) to the rectangle described by prect within the console window.

Returns zero if successful, or E_GEN_ARG if any part of prect is outside the console window (i.e. either of prect->tl.x or prect->tl.y is less than zero, or prect->br.x or prect->br.y respectively exceed the width or height of the console window).

P_FSET, P_SCR_ATTRB**Set character attributes**

```
INT p_iow(VOID *pcb, P_FSET, &P_SCR_ATTRB, UWORD *pattrib);
```

Set the character attributes (style) to be used to display future characters written to the console. This service permits the mixing of a number of different styles in the console window.

The value of *pattrib may be

P_SCR_NORMAL No additional style

or any combination of the bit flags:

P_SCR_BOLD	Bold
P_SCR_REVERSE	Reverse video
P_SCR_UNDLIN	Underlined
P_SCR_BLINK	Blinking
P_SCR_ITALIC	Italics

It is not guaranteed that all the above attributes are supported in all versions of the console for the HC and Series 3, but unsupported attributes are harmlessly ignored.

Any attribute which would have the effect of changing the width of a displayed character (such as P_SCR_BOLD in current machines) is ignored. At the time of writing, no machine supports P_SCR_BLINK.

For example:

```
UWORD func;
UWORD attrib;

p_printf("This is normal");
func = P_SCR_ATTRB;
attrib = P_SCR_UNDLIN;
p_iow(winHandle, P_FSET, &func, &attrib);
p_printf("This is underlined");
attrib = P_SCR_NORMAL;
p_iow(winHandle, P_FSET, &func, &attrib);
p_printf("Back to normal text");
```

Returns zero.

P_FSET, P_SCR_FONT**Set the screen font**

```
INT p_iow(VOID *pcb, P_FSET, &P_SCR_FONT, P_SCR_SET_FONT *pfont);
```

Set the console font and style for the entire window to those specified by the content of the P_SCR_SET_FONT struct pointed to by pfont. The struct is defined in *p_cons.h* as:

```
typedef struct
{
    UWORD id;      /* font id */
    UWORD style;   /* font style */
} P_SCR_SET_FONT;
```

The allowed values for the font id and style are those that are appropriate for setting a window server graphics context (see the *Window Server Reference* manual). Note that the font must be monospaced; if you set id to be the font id of a proportional font you should also include G_STY_MONO in style.

For example:

```
UWORD func;
P_SCR_SET_FONT font;

font.id = WS_FONT_BASE;
font.style = G_STY_MONO | G_STY_DOUBLE;
func = P_SCR_FONT;
p_iow(winHandle, P_FSET, &func, &font);
```

Since the size of the console window may be altered by changing the font, a call to P_FSET with function code P_SCR_FONT will automatically clear and resize the console.

It is not possible to mix different fonts and/or styles in the console window by means of this service. Certain aspects of the font style may be changed using the `P_FSET` service with function code `P_SCR_ATTRB`.

Returns zero.

P_FSET, P_SCR_LAST_LINE_WRAP Set the last line wrap

```
INT p_iow(VOID *pcb, P_FSET, &P_LAST_LINE_WRAP, UWORD *pflag);
```

Turn last line wrap on if `*pflag` is `TRUE`, turn it off if `*pflag` is `FALSE`.

If last line wrap is off then the display immediately wraps and scrolls (provided wrapping and scrolling are not disabled) when a character is printed to the bottom right position in the console.

Last line wrap is initially off.

Returns zero.

P_FSET, P_SCR_FLUSH Set window server flushing

```
INT p_iow(VOID *pcb, P_FSET, &P_SCR_FLUSH, UWORD *pflag);
```

Disable auto-flushing if `*pflag` is `FALSE`, enable it if `*pflag` is `TRUE`.

Auto-flushing is initially enabled.

When auto-flushing is enabled each window server function call is flushed immediately.

If auto-flushing is disabled, one or more window server function calls may be buffered. Execution of a window server function may therefore be deferred until after the initiating function call has returned. In such a case a function may return a window server error that was caused by an earlier, but buffered, function call.

Returns zero.

P_FSET, P_SCR_DISABLE_READS Disable reads

```
INT p_iow(VOID *pcb, P_FSET, &P_SCR_DISABLE_READS, UWORD *pflag);
```

Disable reads if `*pflag` is `TRUE`, enable reads if `*pflag` is `FALSE`.

A console application which needs to receive events directly from the window server, by calling `wGetEvent`, should first disable reads. Otherwise all events are preferentially reported to the console.

No `P_FREAD` or `P_EVENT_READ` request should be outstanding when this service is called and no more reads may be performed until reads are re-enabled.

Initially reads are enabled.

Returns zero.

P_FSET, P_SCR_CLIENT_FOREGROUND Bring to foreground

```
INT p_iow(VOID *pcb, P_FSET, &P_SCR_CLIENT_FOREGROUND, UWORD *pid);
```

Make the process with process ID `*pid` the foreground process. Use a value of zero for `*pid` to bring the current process to the foreground.

The service does nothing if the process specified by `*pid` does not exist.

Returns zero.

P_FSET, P_SCR_CAPTURE_KEY **Capture a key**

```
INT p_iow(VOID *pcb, P_FSET, &P_SCR_CAPTURE_KEY, CONSOLE_CAPTURE_KEY *pcapt);
```

Specify a keypress that is to be sent to the calling process, irrespective of its foreground/background status.

The keypress combination to be captured is specified by the content of the CONSOLE_CAPTURE_KEY struct, defined in *p_cons.h* as:

```
typedef struct
{
    UWORD code;
    UBYTE modifier_mask;
    UBYTE dont_care_mask;
} CONSOLE_CAPTURE_KEY;
```

Every time a key is pressed the window server evaluates

```
(key_pressed==code) && ((key_pressed_modifiers&dont_care_mask)==modifier_mask)
```

and if the result is TRUE then the keyboard event is sent to the process that specified the capture.

For example,

```
UWORD func;
CONSOLE_CAPTURE_KEY capture;

capture.code = W_SPECIAL_KEY|'a';
capture.modifier_mask = W_PSION_MODIFIER;
capture.dont_care_mask = W_PSION_MODIFIER|W_SHIFT_MODIFIER;
func=P_SCR_CAPTURE_KEY;
p_iow(winHandle,P_FSET,&func,&capture);
```

captures PSION+A and PSION+CTRL+A.

See also the equivalent window server function *wCaptureKey*, described in the *Window Server Reference* manual. The struct elements *code*, *modifier_mask* and *dont_care_mask* correspond to the *wCaptureKey* parameters *keycode*, *modifiers* and *modifier_mask* respectively.

Returns zero if successful, or *E_FILE_EXIST* if the specified keycode/modifier combination is currently captured by any process.

P_FSET, P_SCR_CANCEL_CAPTURE_KEY **Cancel a key capture**

```
INT p_iow(VOID *pcb, P_FSET, &P_SCR_CANCEL_CAPTURE_KEY, CONSOLE_CAPTURE_KEY *pcapt);
```

Cancel a key capture set up by *P_FSET* with function code *P_SCR_CAPTURE_KEY*. The content of *pcapt* must exactly match that used to initiate the capture.

Returns zero if successful, or *E_FILE_NXIST* if the keycode/modifier combination is not marked as captured.

Example

The following short program exercises many of the services described above. It consists of two parts, the first of which simply receives keys typed by the user and reports them by type on the screen. The second part enables the user to scroll a part of the display and to set the console attributes (scroll lock, line wrap etc) to see how they affect the way typed characters are displayed.

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

GLREF_D VOID *winHandle;

LOCAL_D P_RECTP RectP;

#define CON_HEIGHT 9
#define CON_WIDTH 25

LOCAL_C VOID doSetMode(UWORD mode, VOID *a1)
/*
Shell to P_FSET
*/
{
    INT err;

    if ((err=p_iow4(winHandle,P_FSET,&mode,a1))!=0)
        p_notifyerr(err,"Set service failed",NULL,NULL,NULL);
}

LOCAL_C VOID clear_line(VOID)
/*
position cursor to begining of the line and clear the line
*/
{
    P_RECTP rectp;

    p_iow3(winHandle,P_FSENSE,&rectp);
    rectp.p.x = 0;
    doSetMode(P_SCR_POSA,&rectp.p);
    rectp.r.tl.x = 0;
    rectp.r.br.x = CON_WIDTH;
    rectp.r.tl.y = rectp.p.y;
    rectp.r.br.y = rectp.p.y + 1;
    doSetMode(P_SCR_CLR,&rectp.r);
}

LOCAL_C VOID ReportKey(VOID)
{
    P_CON_KBREC kbrec;

    p_printf("Press Any Key (ESC quits)\r\n");
    FOREVER
    {
        p_iow3(winHandle,P_FREAD,&kbrec);
        clear_line();
        if (kbrec.modifiers&W_PSION_MODIFIER)
            p_printf("Psion key %x",kbrec.keycode);
        else if (p_isprint(kbrec.keycode))
            p_printf("Normal key %c",kbrec.keycode);
        else if (kbrec.keycode==W_KEY_ESCAPE)
            break;
        else
            p_printf("Non printable key %x",kbrec.keycode);
    }
}
```

```

LOCAL_C VOID TryModes(VOID)
{
    INT wrap_lock, scroll_lock, escape;
    P_RECTP rectp;
    P_CON_KBREC kbrec;

    escape = TRUE; /* set toggle flags to default values */
    wrap_lock = TRUE;
    scroll_lock = FALSE;
    rectp.r.tl.x = 5; /* define a rectangle for scrolling */
    rectp.r.tl.y = 1;
    rectp.r.br.x = 20;
    rectp.r.br.y = 6;
    FOREVER
    {
        p_iow3(winHandle, P_FREAD, &kbrec);
        if (!(kbrec.modifiers & W_PSION_MODIFIER))
        {
            if (p_isprint(kbrec.keycode))
                p_putch(kbrec.keycode);
            else if (kbrec.keycode == W_KEY_RETURN)
                doSetMode(P_SCR_NEL, 0);
            else if (kbrec.keycode == '\b')
                /* make backspace destructive */
                p_print("\b \b");
            else if (kbrec.keycode == W_KEY_ESCAPE)
                break;
        }
        else
        {
            kbrec.keycode &= (~W_SPECIAL_KEY);
            switch (kbrec.keycode)
            {
                case W_KEY_LEFT:
                    rectp.p.x = (-1);
                    rectp.p.y = (-1);
                    scroll: doSetMode(P_SCR_SCROLL, &rectp);
                    break;
                case W_KEY_RIGHT:
                    rectp.p.x = (1);
                    rectp.p.y = (1);
                    goto scroll;
                case W_KEY_DOWN:
                    rectp.p.x = 1;
                    rectp.p.y = 0;
                    goto scroll;
                case W_KEY_UP:
                    rectp.p.x = (-1);
                    rectp.p.y = 0;
                    goto scroll;
                case 'w': /* PSION modifier always gives lower case key */
                    wrap_lock = !wrap_lock;
                    doSetMode(P_SCR_WLOCK, &wrap_lock);
                    break;
                case 's':
                    scroll_lock = !scroll_lock;
                    doSetMode(P_SCR_SLOCK, &scroll_lock);
                    break;
                case 'q':
                    escape = !escape;
                    doSetMode(P_SCR_ESCAPE, &escape);
                    break;
                default: /* do nothing */
                    break;
            }
        }
    }
}

```

```
GLDEF_C INT main(VOID)
/*
Allow user to type characters etc
*/
{
    INT err;
    P_RECT rect;

    if ((err=p_open(&winHandle,"CON:",-1))!=0)
    {
        p_notifyerr(err,"No Console device",NULL,NULL,NULL);
        p_exit(0);
    }
    rect.tl.x = rect.tl.y = 0; /* set the screen size */
    rect.br.x = CON_WIDTH;
    rect.br.y = CON_HEIGHT;
    doSetMode(P_SCR_WSET,&rect);

    p_printf("Simple console program");
    ReportKey();

    p_putch(0x0c); /* form feed to clear screen */
    p_printf("Type characters or:");
    p_printf("Psion <- scrolls left/up");
    p_printf("Psion -> scrolls right/dn");
    p_printf("Psion Down scrolls down");
    p_printf("Psion Up scrolls up");
    p_printf("Psion W toggles wrap lock");
    p_printf("Psion S toggles scr1 lock");
    p_printf("Psion Q toggles escape");
    TryModes();

    p_close(winHandle);
    return(0);
}
```

CHAPTER 3

PARALLEL PORT

Introduction

One or more standard centronics parallel ports are available on all SIBO machines, in the form of either a dual serial/parallel expansion module (for example, the HC and MC ranges) or a parallel expansion module (such as the parallel 3-link for the Series 3). The HC cradle provides a third parallel port.

Parallel port device names

Depending on the number and location of ports available, the port device names are "PAR:A", "PAR:B" and "PAR:C". For example, the Series 3 recognises only "PAR:A", but an MC fitted with two serial/parallel expansion modules recognises "PAR:A" (left hand module, looking from the front of the machine) and "PAR:B" (right hand module, looking from the front of the machine).

The parallel port driver is an output-only device driver that does not support a read service.

Panics

All services (with the exception of `p_open`) will cause the calling process to be panicked if the passed channel handle is not valid. Other panics are described under the particular service to which they apply.

Parallel port services

`p_open(PAR:)`

Open a parallel port

```
INT p_open(VOID **pcb, TEXT *pname, -1);
```

Open a channel to the parallel port `*pname`, where `pname` points to the string "PAR:A", "PAR:B" or "PAR:C", writing the address of the channel control block to `*pcb`.

The parallel port lines are powered up and all control lines are cleared low. The port will continue to consume power until the channel is closed.

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

<code>E_FILE_ALLOC</code>	failed to allocate memory for the control block
<code>E_FILE_DEVICE</code>	the port does not exist
<code>E_FILE_LOCKED</code> or <code>E_GEN_INUSE</code>	the port is already open

`p_close`

Close a parallel port

```
INT p_close(VOID *pcb);
```

Power down the parallel port lines and close the port channel corresponding to the specified control block, first cancelling any outstanding `P_FWRITE` request.

Returns zero.

P_FWRITE**Write to a parallel port**

```
INT p_iow(VOID *pcb, P_FWRITE, VOID *buf, UWORD *plen);
VOID p_ioc(VOID *pcb, P_FWRITE, WORD *pstat, VOID *buf, UWORD *plen);
```

Write **plen* bytes of data from buffer *buf* to the parallel port. It is the user's responsibility to preserve the data at **pbuf* and **plen* until the write request completes.

The write request will never complete if the parallel port is not physically connected to a functioning receiver. It is therefore advisable always to write asynchronously to the parallel port, and to use a timer that provides a timeout on each write, as illustrated in the example at the end of this chapter.

Panics if a `P_FWRITE` request is currently outstanding, or if *pcb* is not a valid channel handle.

The completion status code is returned by the synchronous `p_iow(P_FWRITE)` and written to **pstat* by asynchronous calls. The completion status code is zero if the `P_FWRITE` request completed successfully, or one of the following negative error numbers:

<code>E_FILE_WRITE</code>	failed to write
<code>E_FILE_CANCEL</code>	the write was cancelled by a call to the <code>P_FCANCEL</code> service

P_FCANCEL**Cancel a write request**

```
INT p_iow(VOID *pcb, P_FCANCEL);
```

Cancel any outstanding `P_FWRITE` request, causing it to complete with an `E_FILE_CANCEL` completion code.

If a write request is outstanding then an indeterminate amount of data will have been written to the parallel port before the request is cancelled.

Performing a cancel is harmless if no write request is outstanding.

Returns zero.

P_FSENSE**Sense the input control lines**

```
INT p_iow(VOID *pcb, P_FSENSE, UWORD *port);
```

This service is not, at the time of writing, available for any version of either the Series 3 or the Series 3a. It is only available for machines in the HC and MC ranges that contain EPOC with a version number of 2.30 or later.

Write the current values of the centronics port input control lines to **port* (all control lines are cleared low when the port is opened).

The input control line values are defined, in *p_par.h*, according to the following table, where the pin numbers are those appropriate for a 25-way D-type connector.

Symbol	Control line	Pin number
<code>S_BUSY</code>	Busy	11
<code>S_ACK</code>	Acknlg	10
<code>S_ERROR</code>	Error	15
<code>S_PE</code>	Paper error	12

Following the call, the values of the remaining bits at **port* are undefined.

Returns zero.

P_FSET**Write the output control lines**

```
INT p_iow(VOID *pcb, P_FSET, UWORD *type, UWORD *port);
```

This service is not, at the time of writing, available for any version of either the Series 3 or the Series 3a. It is only available for machines in the HC and MC ranges that contain EPOC with a version number of 2.30 or later.

Set or clear the centronics output control lines (all control lines are cleared low when the port is opened).

If *type is 1 then the control lines corresponding to the bits set in *port will be set high.

If *type is 0 then the control lines corresponding to the bits set in *port will be cleared low.

The output control line values are defined, in *p_par.h*, according to the following table, where the pin numbers are those appropriate for a 25-way D-type connector.

Symbol	Control line	Pin number
S_SPARE	---	See below
S_INIT	Init	16
S_AUTOFD	Autofeed	14
S_SELECT	Select	17

Any other bits set in *port are ignored.

The S_SPARE bit is not available on the centronics connector. It may be of use in custom hardware designs as it corresponds to pin 42 of the Psion-specific custom peripheral chip, ASIC5. Changing this bit will have no effect on the standard RS232/parallel expansion module.

Returns zero.

Example

The following example copies one or more files to a parallel port. It uses many of the parallel port services, and illustrates the use of a timer to provide a timeout in conjunction with the P_FWRITE service.

```
#include <p_std.h>
#include <p_sys.h>
#include <p_file.h>
#include <epoc.h>

LOCAL_D VOID *fcb=NULL;
LOCAL_D VOID *pcb=NULL;
LOCAL_D VOID *tcb=NULL;

LOCAL_C VOID Report(INT err)
{
    TEXT buf[E_MAX_ERROR_TEXT_SIZE];

    p_errs(&buf[0],err);
    p_printf("%s",&buf[0]);
}

LOCAL_C INT BufToParallel(VOID)
{
    WORD len;
    WORD pstat,tstat;
    ULONG time;
    TEXT buf[64];

    len=p_read(fcb,&buf[0],64);
    if (len>=0)
    {
        p_ioc5(pcb,P_FWRITE,&pstat,&buf[0],&len);
        time=50; /* 5 second timeout */
        p_ioc4(tcb,P_FRELATIVE,&tstat,&time);
        p_iowait();
        if (pstat==E_FILE_PENDING) /* timer timed out */
        {
            p_iow2(pcb,P_FCANCEL);
            p_waitstat(&pstat);
            return(E_FILE_CANCEL);
        }
        p_iow2(tcb,P_FCANCEL); /* cancel timer */
        p_waitstat(&tstat);
    }
    return(len);
}
```

```
LOCAL_C VOID FileToParallel(VOID)
{
    INT err;

    if ((err=p_open(&pcb,"PAR:A",-1))<0) /* open parallel port */
    {
        Report(err);
        return;
    }
    FOREVER
    {
        if ((err=BufToParallel())<0)
        {
            if (err!=E_FILE_EOF)
                Report(err);
            break;
        }
    }
    p_close(pcb);
    pcb=NULL;
}

GLDEF_C INT main(VOID)
{
    INT err;
    TEXT bb[P_FNAME_SIZE];

    if ((err=p_open(&tcb,"TIM:",-1))<0) /* open timer for timeouts */
    {
        Report(err);
        return(0);
    }
    while (p_getl("Enter file name: ",&bb[0],P_FNAME_SIZE))
    {
        if ((err=p_open(&fcb,&bb[0],P_FOPEN|P_FSTREAM))<0) /* open a file */
        {
            Report(err);
            continue;
        }
        FileToParallel();
        p_close(fcb);
    }
    p_close(tcb);
    return(0);
}
```

Note that this example is based on the assumption that the only two asynchronous events that can occur are the completion of a write to the parallel port and the expiry of the timer. For further discussion of a more general case, see the description of `p_ioc` in the *PLIB Reference* manual.

CHAPTER 4

SERIAL PORT

Introduction

The serial driver in the EPOC operating system supports a fully interrupt driven industry-standard RS-232 serial link.

The driver consists of two cooperating layers. The lower driver is a physical device driver (PDD). It provides a set of services that hide any hardware dependencies from the upper layer.

The upper layer of the driver is a logical device driver (LDD). It provides the logical (hardware independent) services that are described in this chapter. Incoming data is buffered at this level.

Serial port device names

The first serial port on a SIBO machine has the device name "TTY:A". Many machines in the SIBO range have more than one serial port available; a second port has the device name "TTY:B". For example, a Series 3 fitted with a 3 Link (RS232) recognises only "TTY:A", but an MC fitted with two serial/parallel expansion modules recognises "TTY:A" and "TTY:B" (left and right hand modules respectively, looking from the front). HC machines have a third port, on the cradle, with the device name "TTY:C".

HC machines also support serial ports with TTL level signals with device names "TTY:D", "TTY:E", "TTY:F" (direct TTL levels) and "TTY:G", "TTY:H", "TTY:I" (inverted TTL levels). These ports are not described in this chapter.

Panics

All services (with the exception of `p_open`) will cause the calling process to be panicked if the passed channel handle is not valid. Other panics are described under the particular service to which they apply.

Serial port parameters

The serial port parameters are set and sensed with the aid of a `P_SRCHAR` struct, defined in *p_serial.h* as:

```
typedef struct
{
    UBYTE tbaud; /* transmit baud rate */
    UBYTE rbaud; /* receive baud rate */
    UBYTE frame; /* number of data, parity and stop bits */
    UBYTE parity; /* parity type */
    UBYTE hand; /* handshake flags */
    UBYTE xon; /* XON character */
    UBYTE xoff; /* XOFF character */
    UBYTE flags; /* controlling flags */
    ULONG tmask; /* terminator mask */
} P_SRCHAR;
```

The possible settings are described in the following subsections.

One of the major problems with serial communications is ensuring that both ends of a physical link are using the same port parameters. Typically, if one or more of the parameters differ, some data may be transferred successfully whilst other data may not. This can make it difficult to determine which parameter is incorrect.

Baud rate (tbaud, rbaud)

The Baud rate is, strictly speaking, a measure of the electrical signalling rate (the frequency of electrical impulses) of a communications link. In practice, however, it is used to specify the data transfer rate, in bits per second. The Baud rate and the data transfer rate are equal if one bit of data is encoded in each signalling period.

For back-to-back serial communications the Baud rate and the data transfer rate are equivalent. It is only when considering data transfer over a modem that the Baud rate and data transfer rate may differ. Typically, modems that transfer data at a rate higher than 1200 bits per second encode more than 1 bit of data into a single signalling period. For such a modem the Baud rate will differ from the data transfer rate.

The transmit and receive Baud rates (tbaud and rbaud respectively) may be set to one of the following values, which are defined in *p_serial.h*:

```
P_BAUD_50
P_BAUD_75
P_BAUD_110
P_BAUD_134
P_BAUD_150
P_BAUD_300
P_BAUD_600
P_BAUD_1200
P_BAUD_1800
P_BAUD_2000
P_BAUD_2400
P_BAUD_3600
P_BAUD_4800
P_BAUD_7200
P_BAUD_9600
P_BAUD_19200
P_BAUD_38400
P_BAUD_56000
```

The default value for both tbaud and rbaud is P_BAUD_9600. None of the SIBO serial port hardware currently supports split Baud rates (separate transmit and receive Baud rates).

All machines in the SIBO range support Baud of rates of P_BAUD_50 to P_BAUD_9600 inclusive. In addition the MC 200, MC 400 and Series 3a machines support a rate of P_BAUD_19200. The HC and Series 3 machines can also be set to use P_BAUD_19200. However, the clock speed on current machines is slightly too slow to handle this rate of data transfer, and overrun errors occur fairly frequently. Error correcting protocols (eg Link) will run on the HC and Series 3 at 19200 Baud but, because of the high number of retransmissions, they are slower than if they were run at 9600 Baud.

At 19200 Baud an interrupt occurs approximately every 500 microseconds. The following table shows the number of instruction cycles available on a range of machines between two interrupts occurring at this rate.

Machine	Clock speed	Instruction cycles
HC	3.84MHz	480
Series 3	3.84MHz	480
Series 3a	7.68MHz	960
IBM PC/AT	4.77MHz	596
MC 200/400	7.68MHz	960

Character frame (frame)

The basic unit of transmission is the RS-232 asynchronous character frame. This is a sequence of bits which consists of one start bit, between five and eight data bits, an optional parity bit and one or two stop bits.

The start and stop bits are used to synchronise the data transmission.

The number of data bits required largely depends on the nature of the data that is being transferred. For example, eight data bits are needed to transmit arbitrary binary data bytes, but seven bits are sufficient to transmit text, containing only ASCII character codes with values not exceeding 0x7F.

Including a parity bit provides for an elementary degree of error detection in the transmitted data.

The value of `frame` describes the character frame. It may contain one of one of the following values, defined in `p_serial.h`:

<code>P_DATA_5</code>	5 data bits
<code>P_DATA_6</code>	6 data bits
<code>P_DATA_7</code>	7 data bits
<code>P_DATA_8</code>	8 data bits

The value in `frame` may be ORED with any combination of:

<code>P_TWOSTOP</code>	2 stop bits if set, 1 if clear
<code>P_PARITY</code>	parity bit is present if set

All SIBO machines support all of the above settings. The default value of `frame` is

`P_DATA_8`

for eight data bits, one stop bit and no parity.

In the majority of cases the total number of bits needed to transmit a single character is close to ten (for example, one start bit, seven data bits, one parity bit and one stop bit). As a result, a reasonable estimate of the character transfer rate is usually found by dividing the Baud rate by ten.

Parity (`parity`)

If present, the parity bit may be set or cleared to ensure that the combined sum of the set data and parity bits is either odd (odd parity) or even (even parity). Alternatively it may be always set (mark parity) or always clear (space parity). If the `P_PARITY` bit is set in `field`, then `parity` should be set to one of the following values, defined in `p_serial.h`:

<code>P_PAR_EVEN</code>	even parity
<code>P_PAR_ODD</code>	odd parity
<code>P_PAR_MARK</code>	mark parity
<code>P_PAR_SPACE</code>	space parity

Provided that `P_PARITY` is set in `frame`, the parity bit is generated (for transmission) and verified (on receipt) by the port hardware. Reporting of parity errors may be suppressed by setting `P_IGNORE_PARITY` in `flags` (see below).

The default value of `parity` is zero. No machine in the SIBO range supports either mark or space parity.

Handshaking (`xon`, `xoff`, `hand`)

Handshaking (also called *flow control* or *buffer control*) is a mechanism by which either end of a communications link can pause and restart data transmission from the other end. There are two basic forms:

- software handshaking
- hardware handshaking

Handshaking is required to prevent one end from sending data faster than the receiving end can process it. For example, you may be able to send data to a printer at 9600 Baud, but it is unlikely that the printer is capable of printing 960 characters per second. The printer will use handshaking to suspend the transmission of data from time to time (when its internal buffer becomes full) and restart it when it is able to receive more characters (some buffer space becomes available).

Software handshaking involves the transmission of particular characters, one to suspend and one to resume transmission. This is usually called XON/XOFF handshaking, since the characters normally selected to suspend and resume transmission are XOFF (0x13, 0C3) and XON (0x11, 0C1) respectively. Other characters may be used, provided that both ends of the communications link agree.

Software handshaking can sensibly be used only to transfer data that has pure textual content. Arbitrary binary data may include bytes with values equal to one or other of the handshaking control characters. The remote end will not be able to distinguish between such a character being sent as data or as a handshaking control. For example, a WordStar file may contain the binary value 0x13 (XOFF) as an underscore range marker. On receipt of this character the remote end will not transfer any data until it receives an XON and so the transfer process may hang. More importantly, the XOFF character will be removed from incoming data, being interpreted as a control character rather than as data. Thus, in the above example, underscore range markers will be lost during the transfer.

The character codes used to resume and suspend transmission in software handshaking are stored in `xon` and `xoff`. The default values are DC1 (0x11) and DC3 (0x13) respectively.

Hardware handshaking involves the setting of a particular electrical state on specific control lines. It usually involves a pair of lines, the output from one end being the input to the other, and vice versa. A common cause of failure of hardware handshaking is the use of an improperly wired cable. Various control line combinations may be used, as listed below.

DSR/DTR This form of handshaking uses the RS-232 control lines Data Set Ready (DSR) and Data Terminal Ready (DTR). These lines were not originally intended for use as data flow control lines, but were meant to be used to indicate the presence or absence of a remote machine. However, many printers use DSR/DTR handshaking to control the flow of data from a computer.

The receiver's output line (DTR) is held inactive (-ve) to suspend data transfer and set active (+ve) to resume data transfer. This signal is expected to arrive on the remote (sending) machine's DSR input line.

RTS/CTS This form uses the RS-232 control lines Request To Send (RTS) and Clear To Send (CTS). These lines are the ones originally intended for use to control data flow from either end. RTS/CTS handshaking is normally used for hardware handshaking with a modem, and the DSR and DTR lines are used for their originally intended purposes of indicating that a remote machine exists and the port is open.

The output line (RTS) is held inactive (-ve) to suspend data transfer and set active (+ve) to resume data transfer. This signal is received by the remote machine via the CTS input line (ie its now Clear To Send data).

The Data Carrier Detect (DCD) line is not a control of flow handshaking line as described above. It is a condition indicator from a modem to indicate that a connection now exists with a remote modem. The DCD signal is an input only signal and there is no corresponding output line.

The form of handshaking to be used is specified by the value of `hand`. It should contain a combination of the following bits, defined in `p_serial.h`:

P_OBEY_XOFF If set, receipt of the characters specified in `xoff` and `xon` suspend and resume data transmission. If clear, these characters are treated as ordinary data. This controls input XON/XOFF handshaking, which is independent of output XON/XOFF handshaking.

P_SEND_XOFF If set, the characters specified in `xoff` and `xon` are transmitted to the remote device to suspend and resume data transmission from the remote device. This controls output XON/XOFF handshaking, which is independent of input XON/XOFF handshaking.

P_IGN_CTS If clear, the driver will set its RTS line inactive to suspend transmission from the remote device, and transmission will be suspended when the remote device sets its RTS (incoming CTS) line inactive (RTS/CTS handshaking). If set, then RTS is permanently active and the state of the input CTS line is ignored.

P_OBEY_DSR If set, transmission will be suspended when the remote device sets its DTR (incoming DSR) line inactive (DTR/DSR handshaking). If clear, the state of the incoming DSR line is ignored. In all cases the DTR line is held permanently active while the serial port is powered up and open.

P_FAIL_DSR This is ignored unless `P_OBEY_DSR` is set. If `P_FAIL_DSR` is set and the remote device sets its DTR line inactive then any outstanding `P_FWRITE` or `P_FREAD` request is completed with an `E_FILE_LINE` error (rather than simply suspending transmission). The CCITT recommendations state that a serial port should hold its DTR line active while the port is powered up and active. Setting `P_FAIL_DSR` allows application code to detect that a conforming remote device has been removed or the connection broken.

P_OBEY_DCD If set, then transmission is suspended if the remote device sets the incoming DCD line inactive. If clear, the state of the DCD line is ignored.

P_FAIL_DCD This is ignored unless `P_OBEY_DCD` is set. If `P_FAIL_DCD` is set and the remote device sets the incoming DCD line inactive then any outstanding `P_FWRITE` or `P_FREAD` request is completed with an `E_FILE_LINE` error (rather than simply suspending transmission).

The default value of `hand` is zero, setting RTS/CTS handshaking.

In addition to using the control lines for hardware data flow control (which is transparent to the application code), the serial port services allow the explicit testing of the state of the control lines or, for example, to wait for the DCD line to be driven by a modem.

Control flags (`flags`)

The control flags specify how the device driver should handle certain events. At the time of writing there is only one control flag, defined in `p_serial.h`:

<code>P_IGNORE_PARITY</code>	The presence or absence of a parity bit in received data is not under the receiver's control. Setting <code>P_IGNORE_PARITY</code> causes the serial driver to discard any parity errors on received data. A character received with a parity error is placed in the receive buffer and treated as a normal character, even though it may be in error.
------------------------------	--

For future compatibility the remaining bits of `flags` should be zero.

The default value of `flags` is zero.

Terminator characters (`tmask`)

A receiver has no universally reliable way of knowing when the transmitter has no more data to send. Once some data has been received, the receiver may be entitled to assume that a period (say, five seconds) in which no data arrives means that no further data will arrive. Clearly, this is not foolproof.

This problem may be solved provided that the transmitter cooperates with the receiver to the extent of sending mutually agreed terminating characters.

One or more terminating characters may be specified by setting one or more bits of `tmask`. This consists of 32 bit flags, each corresponding (in order, from bit zero to bit 31) to a terminating character code in the range `0x00` to `0x1f` inclusive.

Thus, setting bit zero (`0x00000001`) selects character code `0x00` to be a terminating character and setting bit 26 (`0x04000000`) selects CTRL-Z. An important use is to set bit 13 and/or bit 10 to select CR and/or LF. This allows text to be read a line at a time from a serial port.

The default value of `tmask` is zero, to select no terminating characters.

Serial port errors

Several errors can arise from a serial port that do not occur elsewhere. They include:

<code>E_FILE_PARITY</code>	A parity error means that the received electrical signal corresponding to the parity bit does not match the parity value calculated from the received data bits. This is typically caused by electrical noise and usually indicates that the data has been corrupted in transmission. Parity error detection is not particularly robust, since many errors pass undetected.
<code>E_FILE_FRAME</code>	A framing error means that the received electrical signal does not match the start and stop bits. This most commonly occurs if data is being sent at a Baud rate which differs from that which the receiver is expecting. Alternatively it may mean that the number of stop bits expected by the receiver is greater than the number being sent by the transmitter.
<code>E_FILE_OVERRUN</code>	An overrun error occurs when the received electrical signal does not match the stop bits. Overrun and framing errors are very similar and it is often difficult to differentiate between them. Some devices may not report them as different errors.
<code>E_FILE_LINE</code>	A line error is generated when the hardware detects an inactive signal on an input control line that the receiver requires to be permanently active (see also <i>Handshaking</i> in the <i>Serial port parameters</i> section).

Serial port services

p_open(TTY:)

Open a serial port

```
INT p_open(VOID **ppcb, TEXT *pname, -1);
```

Open a channel to the serial port *pname, where pname points to a serial port device name, as specified earlier. If successful, write the channel to *ppcb and return zero.

For example:

```
if (!p_open(&pSerial, "TTY:A", -1))
{
    ...
    p_close(pSerial);
}
```

If successful the port will be powered up and the DTR line will be driven active to indicate to a connected remote device that the port is open.

When opened a serial port will adopt its default characteristics of

- Baud
- data bits, no parity, 1 stop bit
- RTS/CTS handshaking
- DC1 is the XON, DC3 is the XOFF character
- Parity errors are not ignored
- No terminator characters

These characteristics may be incorrect for the required usage. To ensure that no received data is lost from, say, a buffered modem, the RTS line will not be driven active until either the first P_FREAD request or the setting of some serial characteristics. (This, of course, assumes that the remote device is using RTS/CTS handshaking.)

The call to p_open returns zero if successful, or a negative error. Errors include:

E_GEN_NOMEMORY	there is not enough memory available to open the device
E_GEN_INUSE	the specified port is currently in use
E_FILE_DEVICE	an illegal or non-existent device is specified
E_FILE_LOCKED	attempting to open too many channels to the device driver

p_close

Close a serial port

```
INT p_close(VOID *pcb);
```

Close the specified channel to the serial driver.

The internal receive data buffer is flushed. If a character is currently being transmitted, the P_FCLOSE request waits until its transmission is complete. The transmit data register becomes empty.

Any outstanding serial driver P_FREAD or P_FWRITE requests are cancelled.

The RTS and DTR lines are set inactive to indicate that the serial port is no longer in use and the serial port is powered down.

Returns zero.

P_FREAD**Read from the serial port**

```
INT p_iow(VOID *pcb, P_FREAD, VOID *buf, UWORD *plen);
VOID p_ioc(VOID *pcb, P_FREAD, WORD *pstat, VOID *buf, UWORD *plen);
```

Read up to *plen bytes to *buf from the serial port. The supplied buffer is assumed to be at least *plen bytes long. The only limit to the amount of data that can be received in one call is the size of the data buffer at buf. It is the user's responsibility to preserve the data space pointed to by buf and plen until the P_FREAD request completes.

The P_FREAD request will typically take a significant length of time to complete and should normally be called asynchronously in a quality system.

The request will complete when one of the following occurs:

- *plen bytes have been received and transferred to *buf. The request completes with zero completion status code.
- a receive error is detected. The number of bytes received before the error occurred is written to *plen and these bytes transferred to *buf. The request completes with a negative error completion status code.
- one of the terminating characters specified by the tmask field of the serial characteristics is received. The number of bytes received, including the terminating character, is written to *plen and these bytes transferred to *buf. The request completes with zero completion status code.
- the request is cancelled. The number of bytes so far transferred to *buf is written to *plen. The request completes with the E_FILE_CANCEL completion status code.

The buffering mechanism at the serial driver's logical device driver level means that, following a P_FCANCEL request, more characters may be available that have not yet been transferred to *buf. The user should test for this by calling the P_FTEST service and, if necessary, calling the P_FREAD service synchronously to extract the remaining characters, as in the following example.

```
len=100;
p_ioc(pSerial,P_FREAD,&stat,&buf[0],&len);
....
p_iow(pSerial,P_FCANCEL);
p_waitstat(&stat); /* use up P_FREAD signal */
....
p_printf("Buf[] contains %d bytes",len);
....
p_iow(pSerial,P_FTEST,&len);
if (len)
    p_read(pSerial,&buf[0],len); /* will complete immediately */
....
p_printf("There are %d more bytes available",len);
```

The calling process is panicked if a P_FREAD request is already outstanding or if pcb is not a valid channel handle.

The completion status code is returned by the synchronous p_iow(P_FREAD) and written to *pstat by asynchronous calls. The completion status code is zero if the P_FREAD request completed successfully, otherwise it is a negative error number. Errors include:

E_FILE_PARITY	a parity error occurred while receiving a character. This error will only be reported if the user has not set P_IGNORE_PARITY in the serial characteristics. Otherwise the error is discarded and the character (although in error) will simply be transferred to *buf.
E_FILE_FRAME	a serial framing error has occurred.
E_FILE_OVERRUN	a serial overrun error has occurred.
E_FILE_LINE	a line failure has occurred.
E_GEN_OVER	the internal serial driver buffer has become full and further incoming characters have been discarded. Some form of handshaking is required to slow the transmitter down.

```
INT p_iow(VOID *pcb, P_FWRITE, VOID *buf, UWORD *plen);
VOID p_ioc(VOID *pcb, P_FWRITE, WORD *pstat, VOID *buf, UWORD *plen);
```

```
P_SRCHAR serChar;

p_iow(pSerial,P_FSENSE,&serChar);
serChar.hand|=P_OBEY_DSR;
p_iow(pSerial,P_FSET,&serChar);
len=0;
if (!p_iow(pSerial,P_FWRITE,&buf[0],&len))
    { /* DSR now being driven */
        serChar.hand|=P_FAIL_DSR;
        p_iow(pSerial,P_FSET,&serChar);
        ...
    }
serChar.hand&=~(P_OBEY_DSR|P_FAIL_DSR);
p_iow(pSerial,P_FSET,&serChar);
```

E_FILE_LINE a line failure has occurred.

The `P_FCANCEL` service cannot fail and returns zero.

P_FSENSE **Sense the serial port characteristics**

```
INT p_iow(VOID *pcb, P_FSENSE, P_SRCHAR *pch);
```

Write the current serial characteristics to the P_SRCHAR struct pointed to by pch.

The P_FSENSE service cannot fail and returns zero.

P_FSET **Set the serial port characteristics**

```
INT p_iow(VOID *pcb, P_FSET, P_SRCHAR *pch);
```

Set the serial port characteristics to those passed in the P_SRCHAR struct pointed to by pch.

An application which wishes to change the serial port characteristics will normally first sense the current characteristics, using the P_FSENSE service. It will then make the required change and use the P_FSET service to apply the new characteristics.

The calling process is panicked if there is an outstanding P_FREAD or P_FWRITE request or if pcb is not a valid channel handle.

Returns zero if the P_FSET request completed successfully, otherwise a negative error. Errors include:

E_GEN_ARG	one or more of the values specified in the P_SRCHAR struct is illegal. The current characteristics are not changed.
E_GEN_NSUP	one of the required characteristics is not supported by this particular driver. The current characteristics are not changed.
E_FILE_LINE	either DSR is inactive and P_FAIL_DSR has been specified or DCD is inactive and P_FAIL_DCD has been specified (or both).

P_FFLUSH **Flush the read buffer**

```
INT p_iow(VOID *pcb, P_FFLUSH);
```

Discard the current contents of the internal serial read buffer (at the logical device driver level) and clear any outstanding error status. Any handshaking that has paused transmission from the remote device is cleared so that the remote device is free to resume transmission.

The P_FFLUSH service cannot fail and returns zero.

P_FTEST **Test for received characters**

```
INT p_iow(VOID *pcb, P_FTEST, UWORD *plen);
```

Set *plen to the number of bytes that are currently available in the serial driver's internal buffer.

Following the call it is guaranteed that there are at least *plen bytes available. These bytes may be read synchronously, since a synchronous P_READ request to read *plen bytes will complete immediately.

By the time the P_FTEST request has completed there may be more than *plen bytes available, since data is received under interrupt control.

The calling process is panicked if there is an outstanding P_FREAD request or if pcb is not a valid channel handle.

The P_FTEST service cannot fail and returns zero.

P_FCTRL **Test and set control lines**

```
INT p_iow(VOID *pcb, P_FCTRL, UBYTE *pctrl);
```

Read the current state of the CTS, DSR and DCD input control lines to *pctrl as a bit mask and, optionally, set the DTR output control line.

The following bit flags are defined in *p_serial.h*:

P_SRCTRL_CTS	if set CTS is active, otherwise it is inactive
P_SRCTRL_DSR	if set DSR is active, otherwise it is inactive
P_SRCTRL_DCD	if set DCD is active, otherwise it is inactive

If the byte pointed to by **(pctrl+1)* is non-zero it is used to set the state of the DTR output line. If non-zero it should take one of the following values, defined in *p_serial.h*:

P_SRDTR_ON	to set DTR active
P_SRDTR_OFF	to set DTR inactive

Returns zero if the P_FCTRL request completed successfully, otherwise a negative error number. Errors include:

E_GEN_NSUP	this driver does not support the setting of the DTR line. All current SIBO machines support the setting of the DTR line.
------------	--

P_FINQ **Inquire supported serial characteristics**

```
INT p_iow(VOID *pcb, P_FINQ, UWORD *pmask);
```

Write three words, into **pmask*, **(pmask+1)* and **(pmask+2)*, containing the serial characteristics that the driver supports. Each word consists of a set of bit flags with each bit set indicating a supported characteristic.

The first word, at **pmask*, indicates a combination of the following potentially supported Baud rates, defined in *p_serial.h*:

P_SRINQ_50
P_SRINQ_75
P_SRINQ_110
P_SRINQ_134
P_SRINQ_150
P_SRINQ_300
P_SRINQ_600
P_SRINQ_1200
P_SRINQ_1800
P_SRINQ_2000
P_SRINQ_2400
P_SRINQ_3600
P_SRINQ_4800
P_SRINQ_7200
P_SRINQ_9600
P_SRINQ_19200

The serial driver in all SIBO machines reports that all the above Baud rates are supported, however see *Baud rate* in the earlier *Serial port parameters* section for a discussion of the use of 19200 Baud.

The second word, at **(pmask+1)*, indicates a combination of the following potentially supported Baud rates, defined in *p_serial.h*:

P_SRINQ_38400
P_SRINQ_56000

SIBO machines do not currently support either of these Baud rates.

The third word, at **(pmask+2)*, indicates support for a combination of the following characteristics, defined in *p_serial.h*:

P_SRINQ_DATA5	supports 5 data bits
P_SRINQ_DATA6	supports 6 data bits
P_SRINQ_DATA7	supports 7 data bits
P_SRINQ_DATA8	supports 8 data bits
P_SRINQ_STOP2	supports sending 2 stop bits
P_SRINQ_PAREVEN	supports even parity
P_SRINQ_PARODD	supports odd parity
P_SRINQ_PARMARK	supports mark parity

P_SRINQ_PARSPEC	supports space parity
P_SRINQ_SETDTR	supports the setting of DTR
P_SRINQ_SPLIT	supports the setting of split Baud rates

The SIBO machines support all of the above except P_SRINQ_PARMARK, P_SRINQ_PARSPEC and P_SRINQ_SPLIT.

The P_FINQ service cannot fail and returns zero.

Example

The following example, which assumes that the serial port is connected to a modem, uses many of the serial services described above.

After opening a console it sends a dialling string to the modem and waits for a connection to a remote modem. It then receives text records transmitted by the remote modem and displays them in the console window. This continues until the user exits the program by pressing the ESCAPE key.

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

#define OPEN_TIMER 0
#define OPEN_PORT 1
#define DTR_HAND 2
#define DIALLING 3
#define LINE_FEED 0x0a

LOCAL_D VOID *pTimer=0; /* Timer channel handle */
LOCAL_D WORD timStat; /* Timer completion status */

LOCAL_D VOID *pSerial=0; /* Serial channel handle */
LOCAL_D WORD serWriteStat; /* Serial write completion status */
LOCAL_D WORD serReadStat; /* Serial read completion status */

GLREF_D VOID *winHandle; /* Console channel handle */
LOCAL_D WORD winStat; /* Console completion status */

LOCAL_C VOID reportError(INT func,INT error)
{
    UBYTE bb[100];

    switch (func)
    {
        case OPEN_TIMER:
            p_puts("Failed to open timer channel");
            break;
        case OPEN_PORT:
            p_puts("Failed to open serial port");
            break;
        case DTR_HAND:
            p_puts("Modem not driving DSR");
            break;
        case DIALLING:
            p_puts("Failed to connect to remote Modem");
            break;
        default:
            p_puts("Unknown function error");
    }
    p_errs(&bb[0],error);
    p_puts(&bb[0]);
    p_close(pSerial);
    p_close(pTimer);
    p_exit(1);
}
```

```
LOCAL_C VOID QueueTimer(ULONG timeout)
{
    p_ioc(pTimer,P_FREAD,&timStat,&timeout);
}

LOCAL_C VOID CancelTimer(VOID)
{
    p_iow(pTimer,P_FCANCEL);
    p_waitstat(&timStat);
}

LOCAL_C VOID QueueSerialWrite(UBYTE *buf,UWORD *plen)
{
    p_ioc(pSerial,P_FWRITE,&serWriteStat,buf,plen);
}

LOCAL_C VOID CancelSerialWrite(VOID)
{
    p_iow(pSerial,P_FCANCEL);
    p_waitstat(&serWriteStat);
}

LOCAL_C setHand(INT clrflag,INT setflag)
{
    P_SRCHAR sch;

    p_iow(pSerial,P_FSENSE,&sch);
    sch.hand&=clrflag;
    sch.hand|=setflag;
    return(p_iow(pSerial,P_FSET,&sch));
}

LOCAL_C VOID resetModem(VOID)
/*
Reset a modem by driving DTR low for 2 seconds
*/
{
    UBYTE bb[2];

    bb[1]=P_SRDTR_OFF;
    p_iow(pSerial,P_FCTRL,&bb[0]);
    p_sleep(20L);
    bb[1]=P_SRDTR_ON;
    p_iow(pSerial,P_FCTRL,&bb[0]);
}
```

```
LOCAL_C dialNumber(VOID)
/*
Returns zero if successful, else -ve error number
*/
{
    UWORD len;
    INT ret;

    len=11;
    QueueSerialWrite("ATD9,123456",&len);
    QueueTimer(50L); /* 5 secs to send dial string */
    p_iowait();
    if (serWriteStat!=E_FILE_PENDING)
    { /* sent dial string */
        CancelTimer();
        if (!serWriteStat)
        { /* sent dial string Ok */
            if ((ret=setHand(-1,P_OBEY_DCD))!=0)
                return(ret);
            len=0;
            QueueSerialWrite("",&len); /* Wait for DCD */
            QueueTimer(1200L); /* 2 mins to get through */
            p_iowait();
            if (serWriteStat!=E_FILE_PENDING)
            {
                CancelTimer();
                if (!serWriteStat)
                {
                    if ((ret=setHand(-1,P_FAIL_DCD))!=0)
                        return(ret);
                }
            }
        }
        else
            CancelSerialWrite();
    }
    else /* timeout sending dial string */
        CancelSerialWrite();
    return(serWriteStat);
}
```

```
GLDEF_C main(VOID)
{
    INT ret;
    UWORD len;
    P_SRCHAR sch;
    P_CON_KBREC kbrec;
    UBYTE key[2];
    UBYTE buf[128+2];

    p_puts("Serial Driver Example"); /* opens a console */
    if ((ret=p_open(&pTimer,"TIM:", -1))!=0)
        reportError(OPEN_TIMER,ret);
    if ((ret=p_open(&pSerial,"TTY:A", -1))!=0)
        reportError(OPEN_PORT,ret);
    resetModem();
    if ((ret=setHand(-1,P_OBEY_DSR|P_FAIL_DSR))!=0)
        reportError(DTR_HAND,ret);
    if ((ret=dialNumber())!=0)
        reportError(DIALLING,ret);
    p_iow(pSerial,P_FSENSE,&sch);
    sch.tmask=(1<<LINE_FEED); /* read a record at a time */
    p_iow(pSerial,P_FSET,&sch);
    len=128;
    p_ioc(pSerial,P_FREAD,&serReadStat,&buf[0],&len);
    p_ioc(winHandle,P_FREAD,&winStat,&kbrec);
    FOREVER
    {
        p_iowait();
        if (serReadStat!=E_FILE_PENDING)
        { /* serial read completed */
            if (!serReadStat || serReadStat==E_FILE_RECORD)
            {
                buf[len]=0;
                p_puts(&buf[0]); /* display that record */
                len=128;
                p_ioc(pSerial,P_FREAD,&serReadStat,&buf[0],&len);
                continue;
            }
            p_iow(winHandle,P_FCANCEL);
            p_waitstat(&winStat);
            break; /* Comms failure */
        }
        if (kbrec.keycode==P_KB_ESCAPE)
        { /* Nominated exit key */
            p_iow(pSerial,P_FCANCEL);
            p_waitstat(&serReadStat);
            break;
        }
        key[0]=kbrec.keycode;
        p_write(pSerial,&key[0],1);
        p_ioc(winHandle,P_FREAD,&winStat,&kbrec);
    }
    resetModem();
    p_close(pSerial);
    p_close(pTimer);
    return(0);
}
```

CHAPTER 5

SOUND

Introduction

On all machines sound can be generated from an internal speaker driven by the `SND:` device driver.

In addition all machines, with the exception of the Series 3a, include a built-in piezo-electric buzzer that can be used for emitting warning beeps and other simple sounds. The Series 3a contains software that emulates a buzzer, emitting the sound via the speaker. Using the buzzer has the advantage that (except on the Series 3a) much less power is needed to drive it. In all cases the buzzer (or the buzzer emulator) is controlled by means of the `p_sound` routine.

In all machines output to the sound channel is disabled if either the `E_SOUND_DEVICE` bit of the sound flag is clear or the `E_SOUND_DISABLE` bit of the sound flag is set. These flags are manipulated by the `p_getsnd` and `p_setsnd` functions, described in the *PLIB Reference* manual.

Sound on MC and HC machines

MC and HC machines can emit sound from either a piezo-electric buzzer or an amplifier-driven loudspeaker. In normal circumstances the buzzer output is controlled by the `PLIB p_sound` function and the `SND:` sound device driver controls output to the speaker.

As a power-conserving strategy, all sound output is directed to the buzzer if the main battery voltage is less than the warning voltage level and no mains supply is connected.

The speaker is a two voice output device, that is, it can play two-note chords. This means that DTMF tone dialing is possible.

The sound services provide for the generation of a wide range of notes or note sequences.

Sound on Series 3 machines

The Series 3 can emit sound from either a piezo-electric buzzer or an amplifier-driven loudspeaker. In all circumstances the buzzer output is controlled by the `PLIB p_sound` function and the `SND:` sound device driver controls output to the speaker.

The Series 3 sound device driver services are limited to providing either DTMF dialling tones or simple alarm sounds. The user who wishes to generate more complex sounds must load a suitable device driver - a simple example, *SNDFRC.LDD*, can be found on the SIBOSDK disk included with this manual.

Sound on Series 3a machines

The Series 3a can emit sounds from an amplifier-driven loudspeaker, using either the `SND:` sound device driver or the built-in buzzer emulator. In all circumstances the buzzer emulator output is controlled by the `PLIB p_sound` function. The Series 3a speaker is a two voice output device that can play two distinct notes at the same time. The `SND:` sound device driver provides the same services as on the HC and MC machines, thus allowing for the generation of a wide range of notes or note sequences.

The Series 3a can also record and play `.wve` digital sound files - see the *General System Services* chapter of the *Plib Reference* manual. These are especially useful for adding custom alarm sounds.

Panics

All services (with the exception of `p_open`) will cause the calling process to be panicked if the passed channel handle is not valid. Other panics are described under the particular service to which they apply.

Sound services

The services listed in this section are available on all SIBO machines.

p_open(SND:) Open the sound channel

```
INT p_open(VOID **pcb, "SND:", -1);
```

Open a channel to sound device.

Returns zero if successful, or one of the following errors:

E_FILE_ALLOC	failed to allocate memory for the control block
E_GEN_FAIL	sound device is disabled
E_FILE_LOCKED or E_GEN_INUSE	port is already open or in use

p_close Close the sound channel

```
INT p_close(VOID *pcb);
```

Close the sound channel.

Returns zero.

P_FCANCEL Cancel a write request

```
INT p_iow(VOID *pcb, P_FCANCEL);
```

Cancel any outstanding write request. Performing a cancel is harmless if no write request is outstanding.

Returns zero.

P_FSENSE Sense sound characteristics

```
INT p_iow(VOID *pcb, P_FSENSE, E_SOUND *psound);
```

Sense the sound channel characteristics. The E_SOUND struct is defined in *epoc.h* as:

```
typedef struct
{
    UBYTE beatsPerMinute;
    UBYTE volume;
} E_SOUND;
```

The value of beatsPerMinute has no significance for the Series 3 machine. On HC, MC and Series 3a machines beatsPerMinute may vary from E_SOUND_MIN_BPM (2) to E_SOUND_MAX_BPM (240). The default value is 120.

On HC, MC and Series 3a machines volume may vary from E_SOUND_MAX_VOLUME (0) to E_SOUND_MIN_VOLUME (5). On Series 3 machines the value is restricted to the range 1 to 4 inclusive. On the Series 3a volumes 1 and 2 are identical, as are 4 and 5 - there are thus only four actual volumes. In all cases a smaller value of volume gives a louder sound. The default value is 1.

Returns zero.

P_FSET Set sound characteristics

```
INT p_iow(VOID *pcb, P_FSET, E_SOUND *psound);
```

Set the sound channel characteristics from the data in the E_SOUND struct, as described for the P_FSENSE service. Setting a value for beatsPerMinute has no effect for the Series 3 machine.

To set one sound characteristic without changing the other you should call `P_FSENSE` before `P_FSET`. For example:

```
E_SOUND sound;

p_iow(pcb,P_FSENSE,&sound);
sound.volume = 2;
p_iow(pcb,P_FSET,&sound);
```

will set volume without changing beatsPerMinute.

Returns zero if successful, or `E_GEN_ARG` if either of the characteristics is out of range.

E_FALARM

Write alarm note sequence

```
VOID p_ioc(VOID *pcb, E_FALARM, WORD *pstat, UWORD *palarm);
INT p_iow(VOID *pcb, E_FALARM, UWORD *palarm);
```

Write one of the two standard alarm note sequences.

The alarm note sequence is selected by `*palarm`:

0	'rings' sequence
1	'chimes' sequence

For example,

```
UWORD alarm;

alarm=1;
p_iow(pcb,E_FALARM,&alarm);
```

Panics if an `E_FALARM`, `E_FSSOUNDCHANNELn` or `E_FDIAL` request is currently outstanding, or if `pcb` is not a valid channel handle.

The completion status code is returned by the synchronous `p_iow(E_FALARM)` and written to `*pstat` by asynchronous calls. The completion status code is zero if the `P_FWRITE` request completed successfully, or `E_FILE_CANCEL` if the write was cancelled by a call to the `P_FCANCEL` service.

HC, MC and Series 3a additional sound service

This service is available only on HC, MC and Series 3a machines.

E_FSSOUNDCHANNELn

Write to voice n

```
VOID p_ioc(VOID *pcb, E_FSSOUNDCHANNELn, WORD *pstat, WORD *pnotes, WORD *plen);
```

Write `*plen` notes to voice `n` (`n` may be 1 or 2) of the sound channel. The value of `*plen` must be less than or equal to 16384 (0x4000).

The parameter `pnotes` should point to an array of `*plen` notes. A note consists of two WORDs, the first giving the note frequency in Hz (middle A is 440Hz) and the second gives the note duration in beats. By default there are 120 beats per minute, but this value may be changed using `P_FSET`. A zero frequency corresponds to silence.

To allow the output to both voices to be synchronised, neither output will start until both `E_FSSOUNDCHANNEL1` and `E_FSSOUNDCHANNEL2` services have been called.

Panics if an `E_FSSOUNDCHANNELn` (on the same channel) or an `E_FALARM` request is currently outstanding, or if `pcb` is not a valid channel handle.

The completion status code is written to `*pstat`. It is zero if the request completed successfully, otherwise it is a negative error code.

If you only require to use one of the voices you must still call both services but can, for example, set the value pointed to by `plen` to zero for the `E_FSSOUNDCHANNEL2` service.

Note: the sound and volume should always be sensed, and then set, immediately after opening the sound channel and before playing sound. This is required even if you wish to play sound at the default volume and beats per minute.

The following example will output a rising and falling scale (known as the *ice-cream van* at Psion):

```
#include <plib.h>

LOCAL_C VOID PlaySound(VOID *pcb)
{
    WORD notes1[] = {1048,24,524,12};
    WORD notes2[] = {1048,4,1320,4,1568,4,2092,4,1568,4,1320,4,1048,12};
    WORD sndstat1,sndstat2;
    WORD len1,len2;
    INT i;

    len1 = sizeof(notes1)/4;
    len2 = sizeof(notes2)/4;
    p_ioc5(pcb,E_FSSOUNDCHANNEL1,&sndstat1,&notes1[0],&len1);
    p_ioc5(pcb,E_FSSOUNDCHANNEL2,&sndstat2,&notes2[0],&len2);
    i = -1;
    do
    {
        p_iowait();
        i++;
    } while (sndstat1 == E_FILE_PENDING && sndstat2 == E_FILE_PENDING);
    if (sndstat1 == E_FILE_PENDING)
        p_waitstat(&sndstat1);
    else
        p_waitstat(&sndstat2);
    while (i--)
        p_iosignal();
}

GLDEF_C INT main(VOID)
{
    VOID *pcb;
    E_SOUND sound;

    p_open(&pcb,"SND:",-1);
    p_iow(pcb,P_FSENSE,&sound);
    p_iow(pcb,P_FSET,&sound);
    PlaySound(pcb);
    p_close(pcb);
    return(0);
}
```

Series 3 and Series 3a additional sound service

This service is available only on Series 3 and Series 3a machines.

E_FDIAL Write DTMF dial tones

```
VOID p_ioc(VOID *pcb, E_FDIAL, WORD *pstat, TEXT *pstr, E_DIAL *pdial);
INT p_iow(VOID *pcb, E_FDIAL, TEXT *pstr, E_DIAL *pdial);
```

Write a DTMF tone sequence from the zero terminated string **pstr*, with timing as specified by the content of the *E_DIAL* struct pointed to by *pdial*. This struct is defined in *epoc.h* as:

```
typedef struct
{
    UBYTE toneLengthTicks;    /* tone length in 1/32 sec */
    UBYTE delayLengthTicks;   /* inter-tone delay in 1/32 sec */
    UWORD pauseLengthTicks;   /* pause length in 1/32 sec */
} E_DIAL;
```

DTMF tones are produced for the following valid characters:

- the digits 0 to 9
- upper or lower case alphabetic characters in the range A to F
- the characters # (0x23) and * (0x2A) which are converted to F and E respectively.

Space and comma characters generate a pause of length specified by the `E_DIAL` struct element `pauseLengthTicks`. All other characters are ignored.

The string at `pstr` may be of any length, but the total number of valid tone and pause generating characters in it may not exceed 26.

For example, assuming a dial-out code of 9,

```
E_DIAL dial;
TEXT digits[10];

dial.toneLengthTicks=8;
dial.delayLengthTicks=8;
dial.pauseLengthTicks=48;
p_scpy(&digits[0], "9,0711234");
p_iow(pcb, E_FDIAL, &digits[0], &dial);
```

will emit the DTMF dialling tones to dial the external number 071 1234.

Panics if an `E_FDIAL` or `E_FALARM` request is currently outstanding, or if `pcb` is not a valid channel handle.

The completion status code is returned by the synchronous `p_iow(E_FDIAL)` service and written to `*pstat` by an asynchronous call. The completion status code is zero if the `E_FDIAL` request completed successfully, or one of the following errors:

<code>E_FILE_CANCEL</code>	the write was cancelled by a call to the <code>P_FCANCEL</code> service
<code>E_GEN_ARG</code>	the string contains too many valid characters

Example

The following code implements a simple tone dialing system for HC, MC or Series 3a machines.

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

GLREF_D VOID *winHandle;

LOCAL_D WORD tones1[] =
{941,1,697,1,697,1,697,1,770,1,770,1,770,1,852,1,852,1,852,1,941,1,941,1};

LOCAL_D WORD tones2[] =
{1336,1,1209,1,1336,1,1447,1,1209,1,1336,1,1447,1,1209,1,1336,1,1447,1,1209,1,1477,1};

LOCAL_C VOID dial(VOID *psoundcb, INT tone)
{
    WORD len = 1;
    WORD s1_status;
    WORD *ptone1, *ptone2;

    if(tone >= 0 && tone <= 11)
    {
        ptone1 = &tones1[0] + tone*2;
        ptone2 = &tones2[0] + tone*2;
        p_ioc5(psoundcb, E_FSSOUNDCHANNEL1, &s1_status, ptone1, &len);
        p_iow4(psoundcb, E_FSSOUNDCHANNEL2, ptone2, &len);
        p_waitstat(&s1_status);
    }
}
```

```
GLDEF_C INT main(VOID)
{
    INT c,tone,err;
    VOID *psoundcb;
    UWORD func;
    P_RECT rect;
    E_SOUND sound;

    if ((err=p_open(&winHandle,"CON:", -1))!=0)
    {
        p_notifyerr(err,"No Console device",NULL,NULL,NULL);
        p_exit(1);
    }

    rect.tl.x = rect.tl.y = 0; /* set the screen size */
    rect.br.x = 25;
    rect.br.y = 9;
    func = P_SCR_WSET;
    p_iow4(winHandle,P_FSET,&func,&rect);
    p_printf("Tone dial demo");
    if ((err=p_open(&psoundcb,"SND:", -1))!=0)
    {
        p_notifyerr(err,"Cannot open sound device",NULL,NULL,NULL);
        p_close(winHandle);
        p_exit(1);
    }

    sound.volume = 3;
    sound.beatsPerMinute = 76;
    p_iow3(psoundcb,P_FSET,&sound);
    FOREVER
    {
        c = p_getch();
        if (c==W_KEY_RETURN);
            break;
        p_putch(c);
        if(c=='*')
            tone = 10;
        else if(c=='#')
            tone = 11;
        else
            tone = c - '0';
        dial(psoundcb,tone);
    }
    p_close(psoundcb);
    p_printf("\r\nDemo finished");
    p_printf("Press any key to exit");
    p_getch();
    p_close(winHandle);
    return(0);
}
```

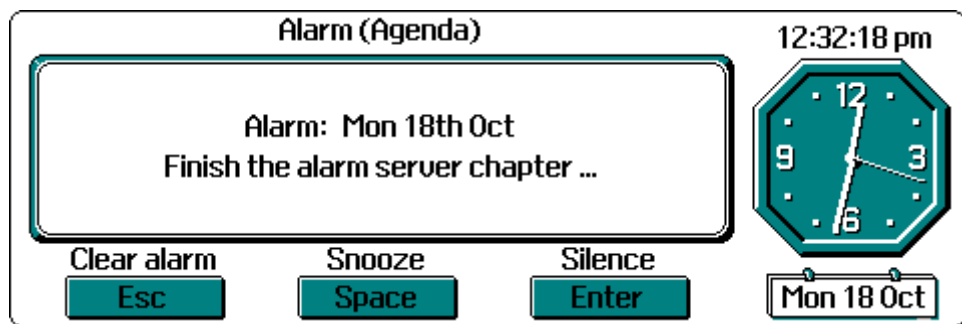
CHAPTER 6

THE ALARM DEVICE DRIVER

Introduction

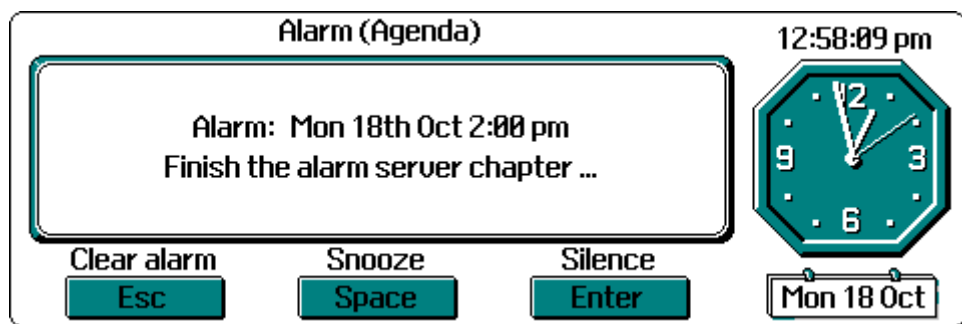
The ALM: device driver supplied on Series 3, Series 3a and MC machines provides support for alarms (as used in the Agenda application). A process may specify an alarm sound (Series 3a only), an alarm message and an appointment time.

The following picture illustrates the screen display for an *untimed* alarm on a Series 3a machine.



Note that the appointment date is displayed but not the appointment time.

The following picture illustrates the screen display for a *timed* alarm on a Series 3a machine.



Both the time and the date of the appointment are displayed.

An alarm is set by sending an alarm request to the alarm device driver (ALM:). This request may be made synchronously, using `p_iow`, or asynchronously, by using `p_ioc`, for example.

Note that the alarm server holds a copy of the data relating to an outstanding alarm request. An outstanding alarm can therefore survive the termination of its parent, that is, the process that made the alarm request. Such an alarm is referred to as an orphaned alarm.

An orphaned alarm is automatically reparented when an identical alarm request is made. This feature is used by, for example, the Agenda application. Thus, even if an Agenda file is repeatedly closed and reopened, an entry with an alarm will not be marked with a large number of identical alarm requests.)

Panics

An application will be panicked if it makes an alarm request before any previous alarm request from that application has completed. All services (with the exception of `p_open`) will cause the calling process to be panicked if the passed channel handle is not valid.

The `#defines` and structs for the alarm server device are in the header file *hasrv.h*.

Series 3, Series 3a and MC alarm services

The Series 3a also has additional services that are described in the following section.

Note that *all* alarm services are provided by the Time application that is built into Series 3 and Series 3a machines, rather than being an integral part of the operating system.

p_open(ALM:) Open the alarm channel

```
INT p_open(VOID **pcb, "ALM:", -1);
```

Open a channel to the alarm device.

Returns zero if the alarm device is opened successfully, otherwise one of the following errors:

```
E_GEN_OPEN
E_GEN_NOMEMORY
E_FILE_NXIST
```

p_close Close the alarm channel

```
INT p_close(VOID *pcb);
```

Close the alarm device channel.

Returns zero.

P_FCANCEL Cancel an alarm request

```
INT p_iow(VOID *pcb, P_FCANCEL);
```

Cancel any outstanding alarm request. Performing a cancel is harmless if no alarm request is outstanding.

Returns zero.

A_FTIMED Queue a timed alarm

```
VOID p_ioc(VOID *pcb, A_FTIMED, WORD *pstat, A_DETAILS *pd, TEXT *pm);
```

Queue a timed alarm specified by a pointer to an `A_DETAILS` struct, and a pointer to a zero terminated text message.

The `A_DETAILS` struct would be defined as follows:

```
typedef struct
{
    ULONG abstime;
    ULONG dueTime;
} A_DETAILS;
```

- `abstime` is the alarm time as a system time.
- `dueTime` is the appointment time as a system time.

The appointment time should not be earlier than the time of the alarm.

The text message pointed to by `pm` should be a zero terminated string of not more than 65 characters including the zero terminator. If no text message is required then `pm` should point to the null string.

The completion status is written to `*pstat`. The status is zero if the service completed successfully otherwise a negative error number.

A_FUNTIMED**Queue an untimed alarm**

```
VOID p_ioc(VOID *pcb, A_FUNTIMED, WORD *pstat, A_DETAILS *pd, TEXT *pm);
```

Queue an untimed alarm specified by a pointer to an A_DETAILS struct, and a pointer to a zero terminated text message.

This service is identical to A_FTIMED (see above) except that only the day and month of the appointment time are displayed, the hours and minutes being omitted.

Series 3a additional alarm services

The services listed in this section are only available on the Series 3a machine.

A_FTIMED_X**Queue a Series 3a timed alarm**

```
VOID p_ioc(VOID *pcb, A_FTIMED_X, WORD *pstat, AXDATA *pa, TEXT *pm);
```

Queue a timed alarm specified by a pointer to an AXDATA struct, and a pointer to a zero terminated text message.

The AXDATA struct is defined in *hasrv.h* as follows:

```
typedef struct
{
    ULONG absTime;
    ULONG dueTime;
    SE_SND sound;
} AXDATA;
```

- absTime is the alarm time as a system time.
- dueTime is the appointment time as a system time.
- sound is the alarm sound (see below).

The SE_SND struct defines the sound to be made and is defined in *hasrv.h* as follows:

```
typedef struct
{
    UBYTE len;
    TEXT name[8];
    UBYTE zero_term;
} SE_SND;
```

- len is the length of the string in the name field.
- name specifies the name of the alarm sound (see below).
- zero_term is the NULL character.

The element name in the SE_SND struct can have one of the following values:

- a single byte containing a value between 1 and 16 inclusive. Currently only 1, 2 and 16 are used, corresponding to the rings, chimes and silent alarms respectively. The contents of following unused bytes are not significant. For example:

```
AXDATA axdata

axdata.sound.len=1;
axdata.sound.name[0]=1;
```

would set the rings sound.

- a non-zero-terminated string specifying one of the .wve digital sound files resident in the ROM. Currently these are: *SY\$AL01* for a 'Fanfare', *SY\$AL02* for 'Soft bells' and *SY\$AL03* for 'Church bells'. For example:

```
AXDATA axdata

axdata.sound.len=8;
p_scopy(&axdata.sound.name[0], "SY$AL03");
```

would set the 'Church bells' sound.

- a non-zero-terminated string specifying a .wve sound file resident in a local \wve\ directory. The contents of following unused bytes are not significant. For example:

```
AXDATA axdata

axdata.sound.len=p_strlen("mysound");
p_scopy(&axdata.sound.name[0], "mysound");
```

would set the sound to be that contained in a \wve\mysound.wve file.

The message pointed to by *pm* should be a zero terminated string of not more than 161 characters, including the zero terminator. If no text message is required, then *pm* should point to a null string.

The completion status is written to **pstat*. The status is zero if the service completed successfully otherwise a negative error number.

A_FUNTIMED_X **Queue a Series 3a untimed alarm**

```
VOID p_ioc(VOID *pcb, A_FUNTIMED_X, WORD *pstat, AXDATA *pa, TEXT *pm);
```

Queue an untimed alarm specified by a pointer to an AXDATA struct, and an associated text message.

This service is identical to the *A_FTIMED_X* service (see above) except that only the day and month of the appointment time are displayed, the hours and minutes being omitted.

CHAPTER 7

THE FREE-RUNNING COUNTER

Introduction

The Series 3a and *Workabout* machines are supplied with a device driver for a built in free-running counter (FRC). This device driver may be accessed by applications and provides for the measurement of elapsed time with an accuracy of +/- 2ppm (corresponding to 2.6 seconds per month) and a resolution of 1/1024 seconds.

The free-running counter device driver may be used in one of three ways:

- to provide a value that increments once every 1/1024 second, effectively a relative clock. An application can read the value at any time, without resetting it.
- to trigger an event after a specified interval, in units of 1/1024 second. In effect this provides the same function as a relative timer, except that the free-running counter works to a higher resolution.
- to provide a value that increments once every specified time interval, in units of 1/1024 second. This allows an application to create a relative clock, with the resolution tailored to the application's requirements. The value is reset to zero each time that it is read.

As the FRC device driver can support only one process at any one time, applications should use it for no longer than is absolutely necessary.

An application that only needs to work with time intervals with a resolution of 1/32 second or greater should not use the FRC, but should use either a relative or an absolute timer: details of these timers can be found in the *Time, Timers and Dates* chapter of the *Plib Reference* manual.

FRC services

The FRC: device driver is specific to the Series 3a and Workabout machines.

All services (with the exception of `p_open`) will cause the calling process to be panicked if the passed channel handle is not valid. Other panics are described under the particular service to which they apply.

p_open(FRC:)

Open the FRC channel

```
INT p_open(VOID **pcb, "FRC:", -1);
```

Open a channel to the FRC device.

Returns zero if the device was opened successfully, otherwise a negative error code. Errors include:

<code>E_FILE_OPEN</code>	The device can not be accessed as it is in use.
<code>E_GEN_NOMEMORY</code>	There is insufficient memory to allow a channel to be opened.

p_close **Close the FRC channel**

```
INT p_close(VOID *pcb);
```

Close the device channel, first cancelling any outstanding read request.

Returns zero.

P_FCANCEL **Cancel the FRC request**

```
INT p_iow(VOID *pcb, P_FCANCEL);
```

Cancel any outstanding request. Performing a cancel is harmless if no request is outstanding.

Returns zero.

P_FSTART **Start the free-running counter**

```
INT p_iow(VOID *pcb, P_FSTART, UWORD *pmode, UWORD *pint);
```

Start the free-running counter.

The UWORD pointed to by *pmode* must specify one of the following *modes*, defined in *epoc.h*:

- | | |
|-----------------|---|
| E_FRC_COUNTING | a counter increments every 1/1024 seconds from an initial value of zero. The value of <i>pint</i> is ignored. |
| E_FRC_REPEATING | a counter increments every * <i>pint</i> multiple of 1/1024 seconds, starting from zero. The value of * <i>pint</i> should be in the range 10 to 65535, representing time intervals approximately in the range of 0.01 to 64 seconds. |

Any outstanding P_FREAD request will be cancelled and the counter reset to zero.

Returns zero if the request completed successfully, otherwise returns a negative error number. Errors include:

- | | |
|-------------|---|
| E_GEN_NSUP | an invalid mode was specified. |
| E_GEN_RANGE | the specified repeat interval is not valid, i.e. it is in the range 0 to 9 inclusive. |

P_FREAD **Read the elapsed time in E_FRC_COUNTING mode**

```
VOID p_ioc(VOID *pcb, P_FREAD, WORD *pstat, ULONG *arg1);  
INT p_iow(VOID *pcb, P_FREAD, ULONG *arg1);
```

Sense the current counter value, following a previous P_FSTART request that set E_FRC_COUNTING mode.

The value written to **arg1* is the time, in units of 1/1024 seconds, since the last P_FSTART request. The value is not reset by the P_FREAD request.

The application will be panicked if an earlier P_FREAD request is outstanding.

The completion status is written to **pstat* for *p_ioc* and returned by *p_iow*. It is zero if the request completed successfully. Otherwise it is a negative error number. Errors include:

- | | |
|---------------|--|
| E_GEN_OVER | the elapsed number of repeat intervals is too large to be written to * <i>arg1</i> . |
| E_FILE_CANCEL | the P_FREAD request was cancelled before it completed. |
| E_FILE_READ | either the counter was not running, or the machine was switched off while the counter was running. |

P_FREAD Read the elapsed time in E_FRC_REPEATING mode

```
VOID p_ioc(VOID *pcb, P_FREAD, WORD *pstat);  
INT p_iow(VOID *pcb, P_FREAD);
```

Sense the current counter value, following a previous P_FSTART request that set E_FRC_REPEATING mode.

The completion status value that results from a successful call to this service is the number of complete time intervals (as specified by the earlier P_FSTART request) that have elapsed since the last request for either a P_FSTART or a P_FREAD service. The service will not complete until one time interval has elapsed, otherwise completion is immediate. Time is not lost between successive P_FREAD requests, since the FRC hardware is not reset.

The application will be panicked if an earlier P_FREAD request is outstanding.

The completion status is written to *pstat for p_ioc and returned by p_iow. It is a positive value if the request completed successfully. Otherwise it is a negative error number. Errors include:

E_GEN_OVER	the elapsed number of repeat intervals exceeds 32767.
E_FILE_CANCEL	the P_FREAD request was cancelled before it completed.
E_FILE_READ	either the counter was not running, or the machine was switched off while the counter was running.

CHAPTER 8

THE SERIES 3 WORLD DATABASE

Introduction

The World database is supported by all of the Series 3 range of machines. It stores a range of data for a large number of countries - in practice only small or very recently created countries are not present. Further data is stored for the larger cities in each country.

The WLD: device driver allows an application to navigate the database by means of a number of search services. A successful search sets the current city and/or country for other services that can access or update the constituent data according to specific requirements. It supports multiple channels.

The services of this device driver are provided by the World application that is built into the ROM of all machines in the Series 3 range.

The data stored in the database includes various codes required for dialling as follows:

- the national code for each country.
- the international prefix code for each country.
- the national prefix for long distance calls.

The database also includes the following data for each city:

- the deviation of the local time from Greenwich Mean Time.
- the times of sunset and sunrise expressed as a local time (these are actually calculated).
- the latitude and the longitude of the city.
- the coordinates of the city on the World map.
- the zone for daylight saving time (DST).

An application may also:

- set its home city.
- set its default country.

The database supports the creation of a file containing extra items which may be either new cities, or replacement data for existing cities: only 32 extra items may be included in any one file. (A greater number of extra items would significantly degrade the performance of the database search operations.)

Mode

A minority of services change the mode of the the database which can effect the operation of subsequent services. Thus the WR_NEXT service moves to the next city if the database is in city mode. Otherwise the WR_NEXT service moves to the next country.

A service is assumed not to change the mode unless it is explicitly stated to the contrary.

Series 3 and Series 3a World database services

Note that these services are provided by the World application that is built into Series 3 and Series 3a machines, rather than being an integral part of the operating system.

p_open(WLD:) **Open the World channel**

```
INT p_open(VOID **ppcb, "WLD:", 0xffff);
```

Open a channel to the World database and write the handle of the channel to *ppcb.

Returns zero on success, otherwise a negative error code including the following:

`E_FILE_ALLOC` failed to allocate memory for the control block.

p_close **Close the World channel**

```
INT p_close(VOID *pcb);
```

Close the channel to the World database specified by pcb.

Returns zero on success.

P_FCANCEL **Cancel a World request**

```
INT p_iow(VOID *pcb, P_FCANCEL);
```

Cancel any outstanding World request on channel pcb and return zero - performing a cancel is harmless if there is no outstanding request.

The `P_FCANCEL` request cannot fail.

WR_FIND_CITY **Find by city**

```
INT p_iow(VOID *pcb, WR_FIND_CITY, TEXT *match, WR_FIND_RES *result);
```

Find the first city that matches the string at address match writing the name of the matching city and the associated country to the `WR_FIND_RES` struct pointed to by result - the matching is not case sensitive.

The `WR_FIND_RES` struct is defined as follows:

```
typedef struct
{
    TEXT city[WR_MAX_NAME+1];
    TEXT country[WR_MAX_NAME+1];
} WR_FIND_RES;
```

The significance of the members of the `WR_FIND_RES` struct is as follows:

city the name of the matching city expressed as a zero terminated string.

country the name of the associated country expressed as a zero terminated string.

Returns `WR_FOUND` on success, or `WR_NOT_FOUND` otherwise.

WR_FIND_COUNTRY **Find by country**

```
INT p_iow(VOID *pcb, WR_FIND_COUNTRY, TEXT *match, WR_FIND_RES *result);
```

Find the first country that matches the string at address match writing the name of the matching country and its capital city to the `WR_FIND_RES` struct pointed to by result - the matching is not case sensitive.

The `WR_FIND_RES` struct is defined as follows:

```
typedef struct
{
    TEXT city[WR_MAX_NAME+1];
    TEXT country[WR_MAX_NAME+1];
} WR_FIND_RES;
```

The significance of the members of the `WR_FIND_RES` struct is as follows:

`city` the name of the capital city of the matching country expressed as a zero terminated string.

`country` the name of the matching country expressed as a zero terminated string.

Returns `WR_FOUND` on success, or `WR_NOT_FOUND` otherwise.

WR_FIND_EXACT

Find by city and country

```
INT p_iow(VOID *pcb, WR_FIND_EXACT, WR_FIND_RES *match);
```

Find the country that exactly matches the city specified in `match->city`, or if this is `NULL`, the country specified in `match->country` - the matching is not case sensitive.

For a description of the `WR_FIND_RES` struct, see the above descriptions of the `WR_FIND_CITY` and `WR_FIND_COUNTRY` services.

Returns `WR_FOUND` on success, or `WR_NOT_FOUND` otherwise.

WR_NEXT

Find next city

```
VOID p_iow(VOID *pcb, WR_NEXT, WR_FIND_RES *result);
```

Write the names of the next city and the associated country, or the next country and its capital city, depending on the mode, to the `WR_FIND_RES` struct pointed to by `result`.

The service is always successful.

WR_BACK

Find previous city

```
VOID p_iow(VOID *pcb, WR_BACK, WR_FIND_RES *result);
```

Write the name of the previous city and the associated country, or the previous country and its capital city, depending on the mode, to the `WR_FIND_RES` struct pointed to by `result`.

The service is always successful.

WR_GET_HOME

Find home city

```
VOID p_iow(VOID *pcb, WR_GET_HOME, WR_FIND_RES *result);
```

Write the names of the home city and the associated country to the `WR_FIND_RES` struct pointed to by `result`.

The service is always successful.

WR_SET_HOME

Set home city

```
INT p_iow(VOID *pcb, WR_SET_HOME);
```

Set the home city to the current city.

This is a system wide setting.

Returns zero on success, or `WR_NOTVALID_ERR` if the current city is no longer valid i.e. the current city has been deleted.

WR_GET_DEFAULT_COUNTRY Find default country

```
VOID p_iow(VOID *pcb, WR_GET_DEFAULT_COUNTRY, WR_FIND_RES *result);
```

Write the name of the default country and its capital city to the WR_FIND_RES struct pointed to by result.

Note that the service sets the database to city mode.

The service is always successful.

WR_SET_DEFAULT_COUNTRY Set default country

```
INT p_iow(VOID *pcb, WR_SET_DEFAULT_COUNTRY);
```

Set the default country current country to the current country.

Note that the service sets the database to country mode.

Returns zero on success, or E_GEN_UNDER if the current city is no longer valid i.e. the current country has been deleted.

WR_GET_DIAL_STRING Get dial string

```
INT p_iow(VOID *pcb, WR_GET_DIAL_STRING, TEXT *inString, TEXT *outString);
```

Convert the string pointed to by inString into a dial string, written to the buffer pointed to by outString.

The output string may be played using the sound device: for further details of tone dialling see the *Sound* chapter of the *I/O Devices Reference* manual.

During the conversion, both spaces and hyphens are removed, but commas are left unchanged. A full stop is interpreted as marking the end of the input string. For the significance of commas see the *Sound* chapter of the *I/O Devices Reference* manual. As an example, 0-171,123 4567.89 would be converted to 0171,1234567.

Returns zero on success, otherwise one of the following negative error codes:

E_GEN_ARG	either the input string includes only one square bracket, or the country specified in square brackets does not match a country in the database.
E_GEN_OVER	at some stage in the conversion, the output string exceeded the maximum allowed length of WR_MAX_DIAL_STRING characters.

The conversion is illustrated by the following examples:

009 44 171 234 5678

009 is an *international access code* which indicates that an international call is being made. The example international access code is that of Denmark. In the United Kingdom the international access code is 001.

44 is a *country code* which indicates the destination country. The example country code is that of the United Kingdom.

171 is an *area code* - in this case it is the Inner London code.

234 5678 is the local telephone number.

For this example the service simply outputs 009441712345678.

009 44 (0) 171 234 5678

(0) is the *national prefix* and is enclosed in brackets to indicate that it is not required for international calls.

the other components are as described above.

the output string is 009441712345678. In this case the number corresponds to calling London from Denmark.

+44 171 234 5678

+ is an abbreviation for the international access code.

44 is the country code for the United Kingdom.

171 is the area code.

234 5678 is the local number.

the interpretation of the string depends on the home country. When this is Denmark, the service replaces the plus character with the international access code of Denmark giving 009441712345678. On the other hand, when the home country is the United Kingdom, the plus character and the country code are replaced with the national prefix and the output string is 01712345678.

171 234 5678 [United Kingdom]

171 is the area code.

234 5678 is the local number.

[United Kingdom] is a convenient means of specifying the destination country: note the space between the last digit and the first bracket and that the country is specified in full.

When the home country is Denmark, the output string is 009441712345678.

When the home country is the United Kingdom, the output string is 01712345678.

0 171 234 5678

0 is the *national prefix* - sometimes referred to as the *STD prefix* - and is omitted on international calls.

the output string is 0171 234 5678.

WR_SET_EXTRA

Open file for additional data

```
INT p_iow(VOID *pcb, WR_SET_EXTRA, WORD *flag, TEXT *name);
```

Open a file to store extra cities where name should point to the full file specification stored as a zero terminated string and flag should always be set to P_FOPEN.

The filename is parsed with an extension of .WLD.

The service will fail if the file does not exist.

Returns zero on success, otherwise E_GEN_IMAGE if the file is either badly formatted, or has a different database version number, or a system error code.

WR_EXTRA

Modify additional data

```
INT p_iow(VOID *pcb, WR_EXTRA, WORD *func, WR_EXTRA_DATA *data);
```

Add, update or delete database entries in the file opened by the earlier WR_SET_EXTRA call.

The service assumes that an extra World database file has been opened using the WR_SET_EXTRA service.

The func argument may take one of the following values:

WR_EXTRA_ADD_CITY	specifies that a city should be added to the World database - the data is read from the WR_EXTRA_DATA union pointed to by data. The city must be in an existing country.
-------------------	--

The WR_EXTRA_DATA union is defined as follows:

```
typedef union
{
    WR_CITY_DATA ci;
    WR_COUNTRY_DATA co;
} WR_EXTRA_DATA;
```

For further details of the WR_CITY_DATA struct, see the description of the WR_GET_CITY_DATA service.

WR_EXTRA_UPDATE_CITY	specifies that a city in the World database should be updated - the new data is read from the WR_EXTRA_DATA union pointed to by data. The city must be in an existing country.
----------------------	--

For details of the WR_EXTRA_DATA union see above. For details of the WR_COUNTRY_DATA struct see the description of the WR_GET_COUNTRY_DATA service.

WR_EXTRA_DELETE_CITY	delete a city in the extra database file - the name of the city and the name of the country are read from the WR_EXTRA_DATA union pointed to by data.
WR_EXTRA_UPDATE_COUNTRY	update a country in the World database - the data associated with the country is read from the WR_EXTRA_DATA union pointed to by data. Renaming a country does not delete or otherwise modify the member countries

On success, the service returns either zero or one of the following values:

WR_REVERTED	the city has reverted to its original built-in data.
WR_DELETED	the city has been deleted.

Otherwise the service returns a negative error code which includes the following:

E_GEN_ARG	tried to update a non-existent country, or specified a non-existent city as the capital of a country.
WR_TOO_MANY_ERR	attempted to exceed the maximum allowed number of items in the extra World database file. The maximum number of allowed items is 32.
WR_DUPLICATE_ERR	the new name for a city/country matches an already existing name.
WR_NOT_VALID	attempted to delete a built-in city.
WR_DELHOME_ERR	attempted to delete the home city.
WR_DEL_CAPITAL_ERR	attempted to delete a capital city.

WR_GET_CITY_DATA

Read city data

```
INT p_iow(VOID *pcb, WR_GET_CITY_DATA, WR_CITY_DATA *result);
```

Write data associated with the current city to the WR_CITY_DATA struct pointed to by result.

The WR_CITY_DATA struct is defined as follows:

```
typedef struct
{
    WR_FIND_RES f;
    UBYTE units;
    UBYTE DST;
    WORD GMT;
    LATL latl;
    TEXT dial[WR_MAX_DIAL+1];
    TEXT STD[WR_MAX_CODE+1];
    P_POINT pos;
} WR_CITY_DATA;
```

The members of the WR_CITY_DATA struct have the following significance:

f	the names of the current city and the associated country: for further details see the description of the WR_FIND_CITY service.
units	may be one of the following values: WR_UNITS_MILES the units of distance are to be miles. WR_UNITS_KILOMETERS the units of distance are to be kilometers. WR_UNITS_NAUTICAL the units of distance are to be nautical miles.
DST	may be <i>one</i> of the following values: 0x00 the daylight saving time is to be constant. 0x02 the daylight saving time is to be European. 0x04 the daylight saving time is to be American. 0x08 the daylight saving time is to be Southern.

GMT	the local time - this is expressed as a deviation from Greenwich Mean Time in units of minutes.
latl	the LATL struct is defined as follows: <pre>typedef struct { WORD iLat; WORD iLong; } LATL;</pre> the <code>iLat</code> member gives the latitude of the city in units of minutes of arc. Positive values correspond to northern latitudes. the <code>iLong</code> member gives the longitude of the city in units of minutes of arc. Positive values correspond to western longitudes.
dial	the city dialling code - stored as a zero terminated string.
STD	the city area code - stored as a zero terminated string.
pos	the coordinates of the city on the World map: used, for example, by the World application.

Returns zero on success, otherwise `WR_NOTVALID_ERR` if the item is no longer valid, for example, if the current city has been deleted or renamed.

WR_GET_COUNTRY_DATA

Read city data

```
INT p_iow(VOID *pcb, WR_GET_COUNTRY_DATA, WR_CITY_DATA *result);
```

Write the data for the current country to the `WR_COUNTRY_DATA` struct pointed to by `result`.

The `WR_COUNTRY_DATA` struct is defined as follows:

```
typedef struct
{
    WR_FIND_RES f;
    UBYTE baseGMT;
    UBYTE DST;
    WORD GMT;
    CO_DIAL dial;
} WR_COUNTRY_DATA;
```

The members of the `WR_COUNTRY_DATA` struct have the following significance:

f	contains the names of the current country and the associated capital city: for further details see the description of the <code>WR_FIND_COUNTRY</code> service.
baseGMT	reserved for internal use.
DST	may be <i>one</i> of the following values: <pre>0x00 the daylight saving time is to be constant. 0x02 the daylight saving time is to be European. 0x04 the daylight saving time is to be American. 0x08 the daylight saving time is to be Southern.</pre>
GMT	the local time - this is expressed as the deviation from Greenwich Mean Time in units of minutes.
dial	the country dialling code information (see below).

The `CO_DIAL` struct is defined as follows:

```
typedef struct
{
    TEXT dialIntra[WR_MAX_INTRA+1];
    TEXT dialInter[WR_MAX_INTER+1];
    TEXT dial[WR_MAX_CODE+1];
    UBYTE dummy;
} CO_DIAL;
```

The significant members of the CO_DIAL struct have the following meaning:

dialIntra	a zero terminated string of up to four digits, containing the national dialling prefix.
dialInter	a zero terminated string of up to four digits, containing the international dialling prefix.
dial	a zero terminated string of up to eight digits, containing the national code.

Returns zero on success, otherwise WR_NOTVALID_ERR indicating that the current country is no longer valid, for example, if the current country has been renamed.

WR_CALC Calculate distance, sunrise and sunset

```
VOID p_ioc(VOID *pcb, WR_CALC, WORD *pstat, WORD *state, WR_CALC_DATA *calc);
```

```
INT p_iow(VOID *pcb, WR_CALC, WORD *state, WR_CALC_DATA *calc);
```

Obtain the distance of the target city from the home city and the local sunrise and sunset times.

The calculation proceeds in stages in order that it may be discontinued as and when required.

The first time the WR_CALC service is called, *state must be set to WR_START_STATE. The service should then be repeatedly called until *state is equal to WR_END_STATE indicating that the calculation is complete.

The WR_CALC_DATA struct is defined as follows:

```
typedef struct
{
    WR_CITY_DATA in;
    WR_CALC_OUT out;
} WR_CALC_DATA;
```

The members of the WR_CALC_DATA struct have the following significance:

in specifies the data for the target city: for further details of the WR_CITY_DATA struct see the description of the WR_GET_CITY_DATA service.

out the result of the calculation: the WR_CALC_OUT struct is defined as follows:

```
typedef struct
{
    WORD distance;
    WORD sunRise;
    WORD sunSet;
    WORD always;
} WR_CALC_OUT
```

the distance member specifies the distance from the home city in the specified units.

the sunRise member specifies the time of sunrise - this is a local time in units of minutes.

the sunSet member specifies the time of sunset - this is a local time in units of minutes.

the always member specifies whether the city is always dark (-1), always light (1) or neither (0).

The completion status code is returned by the synchronous p_iow and written to *pstat by the asynchronous p_ioc.

The completion status code is zero if the request completed successfully, otherwise a negative error code.

WR_NEXT_LOCK**Read next city name**

```
INT p_iow(VOID *pcb, WR_NEXT_LOCK, WR_FIND_RES *result);
```

Writes the name of the next city as a zero terminated string to the `city` member of the `WR_FIND_RES` struct pointed to by `result`.

The name of the associated country is also written to the `country` member of the `WR_FIND_RES` struct pointed to by `result`.

Returns zero on success, otherwise a negative error.

World file types and their locations

This section gives a summary of the file structures used by the World Server. The main bulk of the data is stored using a complex compression algorithm. This cannot be duplicated realistically by a third party developer, and the functionality is available through the World Server device driver, so it is not given in detail.

Main World file

The main World database file is stored in the ROM. For a single-language ROM it is:

```
ROM::WORLD.DAT
```

For a multi-lingual ROM is:

```
ROM::WORLD._<lang>
```

where *lang* is the language number selected as returned by `p_get language`, formatted as a 2-digit number, left-filled with 0 if the language number is <10.

For English the filename of the World file in a multi-lingual ROM is `ROM::WORLD.DAT`. In a multi-lingual ROM you are obviously going to have a number of such World files.

World Extension file

Users by default get an extension file *World.dat* created, but they can make their own world extension files with different names. These files contain changes to the main data (city additions and changes, and country changes), which the World Server reads to override the main world database file.

World File format

First 2 bytes	The World file variant signature. This is a scrambled combination of the file version and language. (Each language can have up to 15 versions; these correspond to new releases.)
Next 30 bytes	15 two-byte table pointers. These point to tables, or key positions within tables
Remaining data	Various tables, including a decode table, because all the data following is encoded.

World Extension File format

File header

A World Extension File has a header of 32 bytes:

First 16 bytes	World Extension File signature <code>worldFileType**<0x0></code>
17-18th bytes	World file variant signature. On creation is copied from the main database file. Files which are opened which do not match the main database file are rejected.
19-20th bytes	Unsigned WORD containing size of extra data block
21st-32nd bytes	Not used, filled with <code>0x0</code> bytes

Data block

The extra data block follows, which is a string of leading byte count data blocks. Each contains either city or country alteration information. With the Series 3a and its newer brothers, there is a limit on the number of blocks to 32 (excluding the final, empty, field). This data block is at least one byte long (`0x0`, i.e. a leading byte count list terminator).

CHAPTER 9

XMODEM AND YMODEM

Introduction

All Psion SIBO machines include an Xmodem/Ymodem device driver which implements the industry standard Xmodem and Ymodem data transfer protocols. Only the data transfer protocols are implemented. The applications perform the required file I/O.

The Psion SIBO Xmodem/Ymodem device driver is attached to a lower level driver which must support the services provided by the Psion SIBO serial device driver. As it is an attached driver it can run over any serial channel without modification.

There are numerous variations of both the Xmodem and the Ymodem protocols. The following are supported:

- Xmodem Checksum
- Xmodem CRC
- Xmodem CRC (1K)
- Ymodem
- Ymodem (1K)
- Ymodem-G
- Ymodem-G (1K)

Data transfer protocols overview

The Xmodem and Ymodem protocols were established to allow two way error correcting data transfers between remote computers. The protocols define the data as a series of data blocks each of which is transmitted with a check for data corruption (a few other bytes are also added - see later sections). The Xmodem/Ymodem device driver supports either a one byte checksum or a two byte CRC data integrity check.

One byte checksum

The one byte checksum is the sum of the bytes in the data block with the carries discarded. A one byte checksum is not as reliable an integrity check as a two byte CRC.

Two byte Cyclic Redundancy Check (CRC)

The two byte CRC is the remainder after the datablock (treated as a large binary number) is divided by a sixteen bit number. The supplied device driver uses the CRC sixteen bit divisor recommended by the CCITT (in polynomial form this is $x^{16} + x^{12} + x^5 + 1$). The reader should be aware that other CRCs are also in widespread use (the CRC-16 polynomial for example).

The Xmodem protocol

The Xmodem data transfer protocol was developed in the late seventies as a 'quick hack' for transferring data between dissimilar machines. It was written on and for machines that had eight bit UARTS (Universal Asynchronous Receivers and Transmitters) and is thus an eight bit data transfer protocol.

The protocol is very simple and thus easy to port to other machines. This and the fact that the original implementation was placed in the public domain very early in its life has meant that the Xmodem protocol has become a defacto industry standard data transfer protocol.

The protocol lacks many features considered mandatory in modern data transfer protocols including full duplex transfer and windowing. Worse still it is not robust: corruption of acknowledgement characters or the data frame header characters can fool the protocol into adopting the wrong state.

The file transfer facilities provided by applications that use the protocol simply send the body of the file as data in data frames. Since the protocol was originally developed on CP/M machines, text files should consist of lines of printable ASCII terminated by the CRLF (0x0d 0x0a) character sequence, with the file terminated with the SUB (0x1a) character. It is the responsibility of the application to perform the required file translation. The SIBO filing system allows text files to be opened in 'stream' mode (i.e. as a constant data stream with each text record delimited by CRLF characters), see the P_FSTREAM_TEXT section within the *Files* chapter of the *PLIB Reference Manual*. The local filing system of machines in the SIBO range store data in the same format as MSDOS. Thus a text file can be opened using the P_FSTREAM mode and still present CRLF delimited data. However a remote filing system, that of the Macintosh for example, would not necessarily store data in the same format.

Binary files can be transferred with the Xmodem protocol. In this case no format is implied. All the data should be written to file as it is presented in the Xmodem data frames.

Link establishment

The Xmodem protocol does not define a distinct link establishment phase. The sender of the data waits for a link establishment character (NAK) and then sends the first data frame. When the first data frame is received the link is established and both sides are in the data transfer phase.

If the sender (receiver) does not receive the expected NAK character (data frame) within a reasonable time (two minutes for example) it should quit. The receiver will continue sending NAK characters at regular intervals until either the first data frame is received or the allowed time period has elapsed.

The data transfer phase

The sender transmits a series of Xmodem Checksum data frames. These consist of:

SOH	the start of header character (0x01).
Block number	a one byte binary number labelling each data block. The number wraps to zero at 0xff.
0xff - block number	the one's complement of the block number. This validates the block number.
Data block	a block of 128 data bytes.
One byte checksum	the sum of the data bytes with the carries discarded.

Each frame *must* contain exactly 132 bytes. Thus when there is insufficient data to fill the data block padding must be added. Typically the padding consists of end of file (SUB) characters, although NULL characters would probably be better when sending binary files.

If the data and the block number have not been corrupted the receiver will send a positive acknowledgement character (ACK) requesting the sender to transmit the next data frame. Otherwise the receiver will transmit a negative acknowledgement character (NAK) requesting the sender to retransmit the previous (corrupted) data frame.

Link termination

An Xmodem link termination phase does not exist separately to the data transfer phase. When the last data frame has been transmitted the sender transmits a one byte EOT character that informs the receiver that there is no more data. The receiver should acknowledge this in the same way that a normal data frame is acknowledged, that is with either ACK or NAK characters. Once successfully acknowledged the link has terminated.

Checksum data flow showing error recovery

Sender	Receiver
	NAK
	<timeout>
	NAK
SOH,0x01,0xFE,<128bytes>,CHK	
	NAK
SOH,0x01,0xFE,<128bytes>,CHK	
	ACK
SOH,0x02,0xFD,<128bytes>,CHK	
	ACK
SOH,0x03,0xFC,<128bytes>,CHK	
<timeout>	
SOH,0x03,0xFC,<128bytes>,CHK	
	ACK
EOT	
	NAK
EOT	
	ACK

The CRC variant

Xmodem CRC differs from the Xmodem Checksum protocol only in the data integrity check and the link establishment character.

The Xmodem CRC protocol data frames include a two byte CRC data integrity check (with the high byte being sent before the low byte) in place of the one byte checksum. This gives an improved data integrity check at the expense of an extra byte of non-data - data frames are thus 133 bytes long. The PLIB function `p_crc` will generate the required CRC.

The receiver establishes an Xmodem CRC link by sending a C (0x43) link establishment character. The link establishment will fail if the sender does not support the CRC mode.

CRC data flow showing error recovery

Sender	Receiver
	C
	<timeout>
	C
SOH,0x01,0xFE,<128 bytes>,CRCHI,CRCLO	
	ACK
SOH,0x02,0xFD,<128 bytes>,CRCHI,CRCLO	
	NAK
SOH,0x02,0xFD,<128 bytes>,CRCHI,CRCLO	
<timeout>	
SOH,0x02,0xFD,<128 bytes>,CRCHI,CRCLO	
	ACK
EOT	
	ACK

The 1K variant

The 1K variant allows the sender to transmit 1024 bytes of data per data frame. Each 1K data frame starts with an STX character (*not* an SOH character) and is otherwise identical to an Xmodem CRC data frame. The link establishment, data transmission and link termination phases are the same as for the Xmodem CRC protocol.

The 1K option allows 1024 byte data frames to be intermixed with 128 byte data frames, with the restriction that a retransmitted frame must be the same size as the originally transmitted frame.

1K data frames are more sensitive to corruption (remember that the whole data block is destroyed by the corruption of only one data byte). Thus 1K protocols should not be used when the transmission line is noisy.

The 1K Xmodem variant supports only a two byte CRC data integrity check. It is inadvisable to use a one byte checksum with 1K data frames and this option is not supported by the supplied driver.

The 1K option data flow

Sender	Receiver
STX,0x01,0xFE,<1024 bytes>,CRCHI,CRCLO	C
STX,0x02,0xFD,<1024 bytes>,CRCHI,CRCLO	ACK
STX,0x02,0xFD,<1024 bytes>,CRCHI,CRCLO <timeout>	NAK
STX,0x02,0xFD,<1024 bytes>,CRCHI,CRCLO	ACK
SOH,0x03,0xFC,<128 bytes>,CRCHI,CRCLO	ACK
STX,0x04,0xFB,<1024 bytes>,CRCHI,CRCLO	ACK
EOT	ACK

Abandoning a transfer

The Xmodem specification does not explicitly provide for abandoning a data transfer session. In practice various implementations will request link abandonment by sending a series of CAN (0x18) characters. This method does not work well when the sender is currently transmitting a data frame. The safest option is to continue sending the data frame and afterwards transmit the CAN characters. For a high quality interface this alternative implies an unacceptable wait (for the 1K protocol with a 1200 baud transmission rate the wait would be about eight seconds). The second alternative is to quit data frame transmission immediately and send a series of CAN characters (which could be misinterpreted as belonging to the abandoned data frame). Even if the receiver misinterprets some CAN characters the session will eventually be abandoned.

When sending data the supplied driver will simply stop transmitting the current data frame and send a CAN character immediately.

When receiving data the supplied driver will send a CAN character.

If a CAN character is received by the driver (except as part of a data frame) it will fail any outstanding request with the E_FILE_CANCEL completion status.

The Ymodem protocol

The Ymodem protocol was developed from the realisation that the Xmodem protocol has several weaknesses:

- Only one file can be transferred per command. The file name must be entered at both ends of the link implying that the user has direct or indirect access to both computers (in addition to the Xmodem link).
- The transferred files can contain up to 127 (or even 1023) useless bytes. These are added to pad out the last data block.
- The time and date that the file was last modified are lost.

Some of the other weaknesses of the original specification had already been resolved by developing variants of the Xmodem protocol. These variants (discussed in the previous section) allow for a more sophisticated data integrity check (the CRC variant) and larger data frames (the 1K variant).

The three weaknesses in the above list are all concerned with file transfer rather than data transfer. Unlike Xmodem, the Ymodem protocol is specifically designed for file transfer.

The Ymodem protocol does not attempt to overcome problems with or enhance the actual data flow protocol. Thus Ymodem is subject to the same data corruption problems as Xmodem.

All Ymodem variants use CRCs.

Link establishment

The Link establishment phase is identical to that for Xmodem CRC except that the first data frame transmitted has a block number of zero rather than one. This block contains file information as follows:

- A zero terminated name. This is either the fully specified path name for the file to be transferred or (more commonly) the file name with no path specified. In either case the name must be acceptable to both the sender's and receiver's filing systems (in general machines that have different filing systems will accept only the file name). This field is mandatory.
- The file length. This is stored as a sequence of decimal digits immediately following the zero terminated name. The receiver may use the file length to set the end of file position at the end of the transfer. If further fields are present the file length *must* be present. If there are no further fields the file length is optional.
- The file modification date. This is stored as a sequence of octal digits specifying the time at which the file was last modified in seconds from 00:00:00, January 1, 1970 (Unix time). A single space character separates the file modification date from the file length. If further fields are present the file modification date *must* be present. If there are no further fields the file modification date is optional.
- The mode. This is stored as a sequence of octal digits and specifies the file mode. Unless the file was sent from a Unix machine the mode is set to zero. Files sent from Unix machines that have the mode set to 0x8000 are assumed to be a Unix type regular file. The mode field is separated from the file modification date by a single space character. If further fields are present the mode *must* be present. If there are no further fields the mode is optional.
- Serial number. This is stored as a sequence of octal digits and specifies the serial number of the sender's software. The receiver's use of this number is optional. The serial number field is separated from the mode field by a single space character. This field is optional.

All undefined fields, and the remaining bytes of the data frame, should be set to zero to allow for future compatibility.

The following code fragment generates the first data frame containing the required Ymodem information for filename, length and modification date.

```
p_bfil(&buf[0],128,0);
p_finfo(pFilename,&info);
p=p_scpy(&buf[0],pFilename)+1;
p+=p_gltob(p,info.size,10);
*p++=' ';
p+=p_gltob(p,info.modst,8);
```

When the first frame arrives the receiver should attempt to open the specified file. If the open request is successful the receiver should send back an ACK to continue with the transfer, otherwise a CAN to cancel the transfer.

When there are no more files to transfer the link connection should be re-established with a zero length file name, after which the batch file transfer ends.

The data transfer phase

The data transfer phase is identical to the Xmodem CRC data transfer phase.

Link termination

The Ymodem file transfer termination phase does not exist separately from the data transfer phase. When the file has been transmitted a one byte EOT character is sent to the receiver. The receiver should acknowledge this in the same way that a normal data frame is acknowledged using either ACK or NAK characters. The transfer of the file is now complete. When one or more additional files are to be sent the sender must reestablish a link with the receiver and send the next file in exactly the same manner. When all files have been sent the sender must send an initial (block number zero) data frame containing a zero length file name.

Ymodem file transfer data flow

Sender	Receiver
	C
SOH,0x00,0xFF,<filename,len...>,CRCHI,CRCLO	
	ACK
SOH,0x01,0xFE,<128bytes>,CRCHI,CRCLO	
	ACK
SOH,0x02,0xFD,<128bytes>,CRCHI,CRCLO	
	ACK
EOT	
	ACK
	C
SOH,0x00,0xFF,<file name,len...>,CRCHI,CRCLO	
	ACK
SOH,0x01,0xFE,<128bytes>,CRCHI,CRCLO	
	ACK
SOH,0x02,0xFD,<128bytes>,CRCHI,CRCLO	
	ACK
EOT	
	ACK
	C
SOH,0x00,0xFF,<null file name>,CRCHI,CRCLO	
	ACK

The 1K variant

The 1K variant allows the sender to transmit 1024 bytes of data per frame. The 1K variant data frames are identical to Xmodem 1K data frames (with the exception of the first frame as discussed earlier).

The G variant

The Ymodem-G protocol does not have any specified way of reporting an error to the sender, or more specifically of requesting a re-transmission of a broken data frame. The only advantages over straight ASCII transfer is that the receiver can detect errors (although nothing can be done about them) and multiple files can be transferred. The design philosophy seems to have been that error correcting modems guarantee that the files are not corrupted during transfer and that the inclusion of software error corection severely reduces the modem's throughput. However error correcting modems only ensure that the data is not corrupted during the transfer from one modem to another. They do not check for corruption of the data while it is being sent from the computer to the modem, and vice versa. A typical problem can occur with PCs connected to networks where some network software insists on disabling interrupts for extended periods of time, certainly long enough to get serial overrun errors. The Ymodem-G protocol is highly susceptible to such errors.

A Ymodem-G link is established by the receiver sending a G (0x47) character.

The Ymodem-G protocol only has acknowledgement characters after the first data frame (indicating that the file was successfully opened) and after the EOT character (indicating that the file was successfully closed).

The supplied device driver will send a CAN character if an error was detected while in Ymodem-G mode. This will typically abort the entire transfer.

Ymodem-G file transfer data flow

Sender	Receiver
SOH,0x00,0xFF,<filename,len...>,CRCHI,CRCLO	G
SOH,0x01,0xFE,<128bytes>,CRCHI,CRCLO	G
SOH,0x02,0xFD,<128bytes>,CRCHI,CRCLO	
EOT	ACK
	G
SOH,0x00,0xFF,<file name,len...>,CRCHI,CRCLO	G
SOH,0x01,0xFE,<128bytes>,CRCHI,CRCLO	
SOH,0x02,0xFD,<128bytes>,CRCHI,CRCLO	
EOT	ACK
	G
SOH,0x00,0xFF,<null file name>,CRCHI,CRCLO	

Abandoning a transfer

The Ymodem specification does not explicitly provide for abandoning a data transfer session. Various implementations will send CAN (0x18) characters in an attempt to inform the remote computer that the data transfer session should be abandoned.

When sending data the supplied driver will simply stop transmitting the current data frame and send a CAN character immediately.

When receiving data the supplied driver will send a CAN character.

If a CAN character is received by the driver (except as part of a data frame) it will fail any outstanding request with the `E_FILE_CANCEL` completion status.

Protocol problems

The primary task of any data transfer protocol is to ensure that the data sent is the same as the data received. If the physical media being used could not corrupt the data then ASCII file transfer would be ideal since no overheads are required in validating the data.

Although the Xmodem and Ymodem protocols address the majority of problems concerned various holes in the error recovery have been pointed out. For example:

- Synchronisation will be lost if the NAK character sent by the receiver is corrupted to an ACK character. The transfer will thus fail.
- The use of different link establishment and acknowledgement characters for Xmodem, Ymodem and Ymodem-G protocols can lead to confusion if the character is corrupted. For example, the Ymodem protocol attempts to establish the link by sending a C character. Corruption of the C to a G would fool the receiver into connecting in Ymodem-G mode.
- The use of anonymous acknowledgement characters for requesting data frame (re)transmission can cause timing problems. For example, consider a receiver that sends an ACK or NAK character requesting transmission of a data frame and finds that the sender is preoccupied. The protocol allows for the receiver to wait for a specified time interval and then send another ACK or NAK character. This will work fine unless the sender replies after transmission of the second ACK or NAK character, and before its receipt. In this case the receiver will assume that the transmitted data frame corresponds to the second ACK or NAK character and thus synchronisation will be lost.

In actual field usage the Xmodem and Ymodem protocols perform more than adequately.

Xmodem/Ymodem services

p_open(XMD:) Open an Xmodem/Ymodem channel

```
INT p_open(VOID **ppXmodem, "XMD:", -1);
```

Attach the Xmodem/Ymodem driver to the open channel specified by ppXmodem. All I/O requests on that channel will now be routed to the Xmodem/Ymodem device driver.

The passed channel is assumed to support the set of services provided by the serial driver. Currently only the TTY: driver supports the serial services.

For example:

```
VOID *pcb;

if (!p_open(&pcb, "TTY:A", -1))
{
    if (!p_open(&pcb, "XMD:", -1))
    {
        ...
        p_close(pcb);
    }
    p_close(pcb)
}
```

Note that two calls to p_close are required, one to close the channel to the Xmodem/Ymodem driver and the second to close the channel to the serial driver.

The Xmodem/Ymodem driver senses the current serial driver's characteristics, removes any XON/XOFF handshaking, sets the terminator mask to zero and sets the framing to eight bits, no parity and one stop bit. The original serial characteristics are restored when the driver is closed.

Once a channel to the Xmodem/Ymodem driver has been opened, the application must connect to the computer at the remote end. This done with the P_FCONNECT service.

The calling process will be panicked if the serial driver to which the Xmodem/Ymodem driver is attached has any outstanding requests on it.

Returns zero if the request completed successfully otherwise a negative error number.

p_close Close the Xmodem channel

```
INT p_close(VOID *pXmodem);
```

Close the Xmodem channel specified by pXmodem. The device driver should be closed when the file transfer has completed. All I/O requests on the driver channel will then be routed to the underlying serial driver.

Every outstanding request on the Xmodem/Ymodem driver will be completed, its completion status word will be set to E_FILE_CANCEL and a signal will be generated.

The characteristics of the serial driver are restored to the values held before the Xmodem/Ymodem driver was opened.

The P_FCLOSE request cannot fail and returns zero.

P_FCONNECT Connect to the remote computer

```
VOID p_ioc(VOID *pXmodem, P_FCONNECT, WORD *pstat, UWORD &type, UWORD &mode);
INT p_iow(VOID *pXmodem, P_FCONNECT, UWORD &type, UWORD &mode);
```

Obtain a connection to a computer assumed to be running some Xmodem/Ymodem software.

The connection type can be established as one of:

- P_XMDM_ACP
- P_XMDM_INIT

where `P_XMDM_ACCP` would be used to accept a connection (the application wishes to transmit a file or files) and `P_XMDM_INIT` would be used to initiate the connection (the application wishes to receive a file or files).

The connection mode determines the file transfer mode and can be one of:

<code>P_XMDM_CRCORCHECKSUM</code>	the connection should be established in either Xmodem checksum or Xmodem CRC mode depending on which mode is supported by the remote computer. The driver has a bias towards CRC mode. When transmitting connection request characters it sends two CRC connection request characters (C,0x43) to every checksum connection request character (NAK,0x15). When accepting a connection it will throw away the first NAK character it receives while waiting for a potential C character.
<code>P_XMDM_CHECKSUMMODE</code>	the connection should be established in Xmodem checksum mode only. This will connect to a remote computer that is using checksum mode faster than the <code>P_XMDM_CRCORCHECKSUM</code> option. (If the remote computer is trying to establish a connection in CRC mode a connection will still be established. This fact will be reported to the caller.)
<code>P_XMDM_CRCMODE</code>	the connection should be established in Xmodem CRC mode only. If the remote computer does not support CRC mode no connection will be established.
<code>P_YMODEM_MODE</code>	the connection should be established using the Ymodem batch protocol.
<code>P_YMODEM_G_MODE</code>	the connection should be established using the Ymodem-G batch protocol.

As an option on any of the protocols that support CRC error checking (Xmodem CRC and all the Ymodem variants) the `P_XMDM_ONE_K` flag can be ORed into `mode` to indicate that data frames of size 1K can be transmitted and received.

For example:

```
mode=P_XMDM_CRCMODE|P_XMDM_ONE_K;
```

When this flag is not set the Xmodem/Ymodem driver will refuse to accept 1K data frames and will instead reply with NAK characters. This will eventually cause the data transfer to fail.

When a connection cannot be established, the reason for the failure will be written back to the completion status word.

If the connection is successfully established the completion status word will contain zero and the actual connection mode will be written back to `mode`.

For example:

```
type=P_XMDM_INIT;
mode=P_XMDM_CRCORCHECKSUM;
if (!p_iow(pXmodem,P_FCONNECT,&type,&mode))
{
    if (mode==P_XMDM_CRCMODE)
        p_puts("CRC mode")
    else if (mode==P_XMDM_CHECKSUMMODE)
        p_puts("Checksum mode")
    ...
}
```

Note that the data space for `mode` must be preserved until the connection request completes.

In general the connection request will take an extended time. Thus the request should be made asynchronously. The connection request can be cancelled with the `P_FDISCONNECT` service.

If the `P_FCONNECT` request was started successfully the I/O request returns either zero or a negative error number.

The completion status word is returned by the synchronous `p_iow` request and written to `*pstat` by the asynchronous `p_ioc` request. It is set to `E_FILE_PENDING` while the request is outstanding and zero on successful completion. If the request fails to complete it is set to a negative error number.

The calling process will be panicked if there is an outstanding `P_FCONNECT` request.

Examples

To receive files using the Xmodem-CRC 1K protocol:

```
type=P_XMDM_INIT;
mode=P_XMDM_CRCMODE|P_XMDM_ONE_K;
if (!p_iow(pXmodem,P_FCONNECT,&type,&mode))
{
    ...
}
```

To send files using the Ymodem batch protocol:

```
type=P_XMDM_ACCP;
mode=P_YMODEM_MODE;
if (!p_iow(pXmodem,P_FCONNECT,&type,&mode))
{
    ...
}
```

To send files using the Ymodem-G 1K batch protocol:

```
type=P_XMDM_ACCP;
mode=P_YMODEM_G_MODE|P_XMYM_ONE_K;
if (!p_iow(pXmodem,P_FCONNECT,&type,&mode))
{
    ...
}
```

P_FDISCONNECT **Disconnect from the remote**

```
INT p_iow(VOID *pXmodem, P_FDISCONNECT);
```

Disconnect from the remote computer.

The P_FDISCONNECT request can also be used to abandon the Xmodem/Ymodem session during the connection establishment or data transfer phases.

If a connection has already been established then the P_FDISCONNECT request will cause a CAN character to be transmitted to the remote computer. This may or may not be picked up by the remote Xmodem/Ymodem implementation to indicate that the data transfer phase is being abandoned. The Xmodem/Ymodem protocols do not have any standard mechanism for abandoning data transfer.

If no connection has been established yet, or if the transfer is now complete, no characters will be transmitted to the remote computer.

Any outstanding asynchronous requests will be cancelled by the P_FDISCONNECT request, and their completion status words set to E_FILE_CANCEL. For each outstanding request a signal will be generated and must be used up by the application.

Typically the P_FDISCONNECT request would be called synchronously.

The P_FDISCONNECT request can not fail and returns zero.

Example

```
type=P_XMDM_INIT;
mode=P_XMDM_CRCORCHECKSUM;
p_ioc(pXmodem,P_FCONNECT,&xStat,&type,&mode);
p_ioc(pConsole,P_FREAD,&kStat,&kbr);
p_iowait();
if (xStat!=E_FILE_PENDING)
{
    /* Xmodem connect completed */
    ...
}
else
{
    /* key press occurred - cancel P_FCONNECT */
    p_iow(pXmodem,P_FDISCONNECT);
    p_waitstat(&xStat);
}
```

P_FREAD **Read data from the remote computer**

```
VOID p_ioc(VOID *pXmodem, P_FREAD, WORD *pstat, UBYTE *buffer, UWORD *plen);
INT p_iow(VOID *pXmodem, P_FREAD, UBYTE *buffer, UWORD *plen);
```

Read data from the remote computer. If the connection was not established with a type of P_XMDM_INIT the request will fail with the negative error E_FILE_DISC.

When data is available it will be written to the buffer pointed to by `buffer` and the length of the data will be written to `*plen`.

When the end of data indicator (EOT) from the remote computer is received the P_FREAD request will be completed with the (negative) E_FILE_EOF error number.

The application is responsible for supplying a buffer large enough to hold the largest data frame that can arrive. If the P_XMDM_ONE_K mode flag was set in the P_FCONNECT request this is 1024 bytes otherwise it is 128 bytes.

If the Ymodem protocols are being used the first data frame read will contain the Ymodem file information.

If the P_FREAD request was started successfully the synchronous I/O request returns zero otherwise it returns a negative error number.

Whilst an asynchronous P_FREAD request is outstanding the completion status word is set to E_FILE_PENDING. On completion the status word is set to zero if the read request completed successfully otherwise a negative error number. In particular the completion code is set to E_FILE_EOF if there is no more data to read.

The calling process will be panicked if there is an outstanding P_FREAD request.

Examples

Receiving multiple files using the Ymodem protocol:

```
type=P_XMDM_INIT;
mode=P_YMODEM_G_MODE;
FOREVER
{
    if (ret=p_iow(pXmodem,P_FCONNECT,&type,&mode))
        break;
    if (ret=p_iow(pXmodem,P_FREAD,&fileinfo[0],&len))
        break;
    if (!p_slen(&fileinfo[0]))
    { /* finished multi-file receive */
        ret=0;
        break;
    }
    p_open(&pFile,&fileinfo[0],P_FUPDATE|P_FSTREAM|P_FREPLACE);
    while (!(ret=p_iow(pXmodem,P_FREAD,&buf[0],&len)))
    {
        if (ret=p_write(pFile,&buf[0],len))
            break;
    }
    if (ret==E_FILE_EOF)
    {
        /* Truncate the file as specified in fileinfo[] */
    }
    p_close(pFile);
    /* set the file modification date */
    if (ret!=E_FILE_EOF)
        break;
}
p_iow(pXmodem,P_FDISCONNECT);
if (ret)
    p_puts("Error receiving files");
```

Receiving a single file using the Xmodem protocol:

```
type=P_XMDM_INIT;
mode=P_XMDM_CRCORCHECKSUM;
if (!p_iow(pXmodem, P_FCONNECT, &type, &mode))
{
    /* Xmodem connect completed */
    p_open(&pFile, "temp.tmp", P_FUPDATE|P_FSTREAM|P_FREPLACE);
    while (!(ret=p_iow(pXmodem, P_FREAD, &buf[0], &len)))
    {
        if (ret=p_write(pFile, &buf[0], len))
            break;
    }
    p_close(pFile);
    p_iow(pXmodem, P_FDISCONNECT);
    if (ret!=E_FILE_EOF)
        p_puts("Error occurred");
}
```

P_FWRITE Write data to the remote computer

```
VOID p_ioc(VOID *pXmodem, P_FWRITE, WORD *pstat, VOID *buffer, UWORD *plen);
INT p_iow(VOID *pXmodem, P_FWRITE, VOID *buffer, UWORD *plen);
```

Write data to the remote computer. If the connection was not established with a type of P_XMDM_ACCP the request will fail with the negative error E_FILE_DISC.

Data can be written in buffers of 128 or 1024 bytes, although the device driver does not check whether or not the connect request specified the P_XMDM_ONE_K flag. If the receiver has not been set up to receive 1024 byte frames (the 1k option) it will in general fail to accept the data.

The 1K option specification allows the user to intermix the transmission of 128 and 1024 byte frames, this driver conforms to that specification.

If the write request is for fewer than 128 bytes the internal transmit buffer is padded to 128 bytes with SUB (0x1a) characters. If the request is for more than 128 but fewer than 1024 characters, the internal buffer will similarly be padded with SUB characters.

As an exception to the above, if a write request of zero length is received this is taken to indicate that the end of text marker should be transmitted to the remote computer, thus completing that data transfer.

Unlike for many device drivers the application does not have to preserve the data buffer or the length word data spaces until data transmission has completed.

If the Ymodem protocols are being used the first data frame to be written is assumed to contain the Ymodem file information.

If the P_FWRITE request was started successfully the I/O request returns zero otherwise it returns a negative error number.

Whilst an asynchronous P_FWRITE request is outstanding the completion status word is set to E_FILE_PENDING. On completion the status word is set to zero if the write request completed successfully otherwise, it is set to a negative error number.

The calling process will be panicked if there is an outstanding P_FWRITE request.

Examples

Sending a single file using the Xmodem protocol:

```
type=P_XMDM_ACCP;
mode=P_XMDM_CRCMODE;
if (!p_iow(pXmodem,P_FCONNECT,&type,&mode))
{
    /* Xmodem connect completed */
    p_open(&pFile,"temp.tmp",P_FOPEN|P_FSTREAM|P_FSHARE);
    while ((len=p_read(pFile,&buf[0],128))>0)
    {
        if (ret=p_iow(pXmodem,P_FWRITE,&buf[0],&len))
            break;
    }
    if (!ret && (!len || len==E_FILE_EOF))
    {
        len=0;    /* Force EOT to be sent */
        p_iow(pXmodem,P_FWRITE,&buf[0],&len);
    }
    p_close(pFile);
    p_iow(pXmodem,P_FDISCONNECT);
    if (ret)
        p_puts("Error occurred");
}
```

Sending multiple files using the Ymodem protocol:

```
type=P_XMDM_INIT;
mode=P_YMODEM_MODE|P_XMDM_ONE_K;
FOREVER
{
    if (ret=p_iow(pXmodem,P_FCONNECT,&type,&mode))
        break;
    if (!(ret=getNextFileName(&name[0])))
        break; /* gets Ymodem 1st frame info */
    len=128;
    if (ret=p_iow(pXmodem,P_FWRITE,&name[0],&len))
        break;
    p_open(&pFile,&name[0],P_FOPEN|P_FSTREAM|P_FSHARE);
    while ((len=p_read(pFile,&buf[0],1024))>0)
    {
        if (ret=p_iow(pXmodem,P_FWRITE,&buf[0],&len))
            break;
    }
    if (!ret && (!len || len==E_FILE_EOF))
    {
        len=0;    /* Force EOT to be sent */
        p_iow(pXmodem,P_FWRITE,&buf[0],&len);
    }
    p_close(pFile);
    if (ret)
        break;
}
p_iow(pXmodem,P_FDISCONNECT);
if (ret)
    p_puts("Error sending files");
```


CHAPTER 10

NCP AND LINK

Introduction

This chapter describes the Psion Link application and the peer to peer data transfer services that are provided to enable client server type applications to communicate with each other when the client and server processes exist on different machines.

The basic Link application consists of two processes: LINK and SYS\$NCP. The LINK process varies from machine to machine and provides only the user interface and one basic service. The SYS\$NCP process is identical on all machines and handles the physical connection and data transfer services.

The Link application is organised in four layers, each of which is responsible for a particular job:

- Physical Layer: for example the Serial Driver. This layer provides a set of services that hide any hardware dependencies from the Logical Layer.
- Logical Link Layer. This layer provides the logical (hardware independent) services required by the third layer.
- Data Transport and Multiplexing Layer
- Application Layer

The first three layers are contained within the SYS\$NCP process, whose services are accessed via the NCP: device driver. The SYS\$NCP process provides the data transport mechanism that allows client server processes on different machines to communicate with each other.

There are many possible application layers, one of which is the LINK process. Client and server processes are also examples of application layer processes.

Panics

All services (with the exception of p_open) will cause the calling process to be panicked if the passed channel handle is not valid. Other panics are described under the particular service to which they apply.

The Psion logical link layer protocol

The Psion Link protocol provides the logical link layer services required by the SYS\$NCP process. The services provided are fairly primitive. The SYS\$NCP process adds value to these services. A brief summary of the protocol is presented here to aid in the explanation of the overall system.

The Psion Link protocol is a proprietary protocol based on the MNP protocol. It provides a symmetrical 'error-free' link between computers including:

- Full duplex data transmission.
- 16 bit CRC error detection.
- Multiple retransmissions.
- Dataframe sequencing.
- Variable dataframe size up to a maximum of 300 bytes per dataframe.
- A window size of one.

One of the major design criteria of the protocol was that it could be implemented in a very small amount of code and required a small working set. Thus features such as large data frames (eg 1K) and windowing were rejected.

The protocol is a data transfer protocol, not a file transfer protocol. The file transfer or more accurately file access and management services are provided by application layer processes.

The SYS\$NCP process

The two primary functions of the SYS\$NCP process are connection establishment and data transfer.

Connection establishment

A connection is established when the following criteria have been met:

- A physical link has been established.
- A logical link has been established.
- The SYS\$NCP process has successfully exchanged logon messages with its remote counterpart.

If any of the above fail or the remote counterpart is not compatible a connection will not be established.

A physical link is deemed to have been established when the local machine detects that the remote machine is driving the DTR (local machines DSR) hardware handshaking line (for further details see the *Serial Port* chapter).

When establishing a modem link, a physical link is deemed to have been established when a modem driver reports that an incoming call has arrived or that the dial request has been successful.

A logical link is deemed to have been established when the P_FCONNECT request on the link layer protocol driver completes successfully.

The SYS\$NCP processes exchange logon messages that contain a version number and the time at which the process was started.

The version numbers are checked to ensure compatibility (for possible future expansion).

The communicating SYS\$NCP processes exchange the time at which each was started so as to allow connection re-establishment in the event of an error at either the logical or physical layer. The re-establishment of the application layer connections (if present) will be transparent to the applications.

Data transfer

The SYS\$NCP process creates eight separate logical (software implemented) channels through which applications can communicate. The first channel, channel zero, is used by the SYS\$NCP process, leaving seven channels free for client server applications. When sending data the SYS\$NCP process multiplexes the eight logical channels into one physical channel. Conversely, when receiving data the process demultiplexes the data from one physical channel to eight logical channels.

The transferred data can be divided into control data and application data.

Control data is the data exchanged between communicating SYS\$NCP processes that concerns application connection, disconnection and data flow control.

Application data is the data exchanged between connected applications. It is transparent to the SYS\$NCP process (which places no significance on any of the transferred bytes). Full 8 bit data transfer is available. The SYS\$NCP process simply ensures that the data sent on a particular channel is routed to the correct destination process.

The SYS\$NCP process removes a limitation of the logical link layer, namely the data frame size. The SYS\$NCP process will 'segment' application data such that the only frame size limitation is that of the particular application.

The LINK process

The three primary tasks of the LINK process are:

- to provide a user interface for the Link application
- to respond to state changes in the SYS\$NCP process
- to set up the remote filing system, after successful connection establishment.

On both the Series 3 and the HC, the Link application has no visible user interface: the application is started and stopped by the system process. For example, on the Series 3 the user turns the Link application on or off and sets the baud rate by selecting the Remote link option on the Special menu. The system process launches or terminates the application accordingly.

Since there is no user interface on the Series 3 and the HC, all state changes are handled without prompting the user for a response. See the P_FRSUPER and P_FINQ services described below.

When the SYS\$NCP informs the LINK process that a connection has been established the LINK process at each end runs a process called SYS\$RFSV (the remote file server process). The SYS\$RFSV process provides remote file access and management services. The LINK process then adds the REM:: filing system PDD (Physical Device Driver) to the file server, thus creating a client server pair.

The LINK process also sets itself up as an IPC (InterProcess Communication) message receiver. An application may send the following messages to the LINK process:

LNKMSG_TERMINATE	request that the Link application terminate. This involves removing the remote file system client server pair, and informing the SYS\$NCP process that it should terminate. Instead of sending a message directly an application can use the p_pterminate system service to send this message indirectly.
LNKMSG_LOADREMOTE	request that a message to load and run a specified process be sent to the remote LINK process. The message is assumed to contain up to E_MAX_NAME + 2 bytes. The message consists of the name of the remote process that is to be loaded followed by an optional leading byte count command line. If a command line is required then the process name must be padded out to E_MAX_NAME + 2 bytes (with NULLS) for compatibility purposes. The name is parsed with .IMG and the resultant file name (no device or path) taken as the match pattern for a search that will firstly look in the default directory of the remote LINK process (typically LOC::M:\) then in the root directory of any devices that exist on the remote machine and finally in the ROM. If a process is successfully loaded it will be resumed immediately, the remote LINK process will force a context switch to allow the newly loaded process to run, typically for long enough to open a channel on the NCP: device. The resultant process id and full process name are sent from the remote LINK to the local LINK process. The local LINK process will write back the full process name into the buffer passed and complete the IPC message send with the positive remote process id or negative error number, very similar to the return values for p_execc.
LNKMSG_CONFIG	request that the LINK process inform the SYS\$NCP process to change its driver configuration. The message parameter is the full DVRS struct to be passed to the SYS\$NCP process.

The LNKMSG_... message numbers can be found in *link_def.h* and the DVRS struct definition is in *sys\$ncp.h*.

NCP services

p_open(NCP:)

Open an NCP channel

```
INT p_open(VOID **ppNcp, "NCP:", -1);
```

Requests that a channel to the SYS\$NCP process be opened and a channel allocated to the calling process in preparation for connection to, and communication with, a remote process.

The NCP: device driver is a root device driver, it does *not* require any opened channels to be passed to the `p_open` request. All I/O requests on the allocated channel will be routed to the `SYS$NCP` process by this device driver automatically.

Opening a channel does not cause any data to be transferred to the remote machine.

A single process may open as many channels on the NCP: device as it likes.

If a process terminates for any reason without closing the channel to the NCP: device the `SYS$NCP` process will tidy up, reporting to any remotely connected channel that this channel has now been closed.

Returns zero if the open request completed successfully otherwise a negative error number. A typical error is that the `SYS$NCP` process is not currently running in which case the `E_FILE_NXIST` error value will be returned.

Example

```
if (!p_open(&pNcp, "NCP:", -1))
{
    p_close(pNcp);
}
```

p_close()

Close the NCP channel

```
INT p_close(VOID *pNcp);
```

Requests that the currently opened channel to the `SYS$NCP` process be closed.

Any outstanding I/O requests will be completed with the `E_FILE_CANCEL` completion status and a signal will be generated.

If the process is still connected to a remote process the remote process will be informed that the connection has been closed, any outstanding requests the remote process has will be completed with the `E_FILE_DISC` completion status.

The close request cannot fail and always returns zero.

P_FCONNECT

Connect to a remote process

```
VOID p_ioc(VOID *pNcp, P_FCONNECT, WORD *pstat, UBYTE *pname, UWORD *plen);
INT p_iow(VOID *pNcp, P_FCONNECT, UBYTE *pname, UWORD *plen);
```

Requests that the channel be connected to a remote channel that has been opened by the named process.

In a typical client server application only the client process would make a `P_FCONNECT` request. The server process would typically open an NCP: channel and queue a `P_FREAD` request awaiting its first command from a client.

If both processes of an application attempt to obtain a connection they should ensure that their counterpart is running and has an NCP: channel open before making the `P_FCONNECT` request otherwise the request will fail with the `E_FILE_NXIST` completion status. Obviously if the connection request fails with the `E_FILE_NXIST` completion status the request can be retried. The number of retries should, however, be limited.

If an open channel is already connected to a remote channel the request will complete successfully.

In the Link application both of the `LINK` processes attempt to obtain a connection. In this case both of these processes will be running and have an NCP: channel open, since they launch the `SYS$NCP` process in the first place.

The `pname` parameter is a pointer to a buffer containing the name of the remote process to connect with. The name is used as the match parameter to the `p_pidfind` service on the remote machine. If that process does not exist or does not have an opened `SYS$NCP` channel the request will fail with the `E_FILE_NXIST` completion status. The data space pointed at is assumed to remain valid until the completion of the `P_FCONNECT` request.

The `p_len` parameter points to a word containing the length of the buffer at `pname` including the zero terminator. The data space pointed at is not required to remain valid until the completion of the `P_FCONNECT` request.

The `P_FCONNECT` request will typically take a significant length of time and as such should be called asynchronously in a quality system.

The `P_FCONNECT` request may be cancelled by using the `P_FCANCEL` service, the original request will be completed with `E_FILE_CANCEL` completion status and a signal generated.

How the remote process came to be running in the first place is of no concern to the `SYS$NCP` process, it is however of great concern to an application writer.

Three primary methods are available to get the remote process running:

- In turnkey systems the remote process may be automatically loaded by the system initialisation.
- The user may be prompted to run the remote process from the command shell or system applications.
- An application can request that the `LINK` process run the remote process on its behalf.

The latter of these three methods is the most general and is best explained by an example as given below.

The completion status is written to `*pstat` by an asynchronous request and returned by a synchronous request. The status is zero if the service completed successfully, otherwise it is a negative error number.

Errors include:

<code>E_FILE_NXIST</code>	the named process does not have a channel open on the remote <code>SYS\$NCP</code> .
<code>E_FILE_DISC</code>	the link is disconnected, there is no data path available to talk to the remote <code>SYS\$NCP</code> .
<code>E_FILE_LINE</code>	the link was disconnected whilst attempting to send data to the remote <code>SYS\$NCP</code> .
<code>E_FILE_RETRAN</code>	the retransmission threshold was reached because the link has been disconnected.

Example

Simple connection to a currently running remote process:

```
if (!p_open(&pNcp, "NCP:", -1))
{
    len=8; /* incl zero terminator */
    if (!p_iow(pNcp, P_FCONNECT, "RPROC.*", &len))
    {

        p_iow(pNcp, P_FDISCONNECT);
    }
    p_close(pNcp);
}
```

Connection to a remote process that is not currently running:

```
if ((linkPid=p_pidfind("LINK.*"))<0)
    p_exit(1);
p_scpy(&bb[0], "RPROC.IMG"); /* name only, no paths */
p=(&bb[0]);
if (p_msndreceivew(linkPid, LNKMSG_LOADREMOTE, &p)<0)
    p_exit(1);
len=p_slens(&bb[0])+1; /* full process name here now */
if (!p_iow(pNcp, P_FCONNECT, &bb[0], &len))
{

    p_iow(pNcp, P_FDISCONNECT);
}
```

P_FDISCONNECT **Disconnect from the remote process**

```
INT p_iow(VOID *pNcp, P_FDISCONNECT);
```

Requests that the current connection to a remote process be broken. An application may use the channel to connect to the same remote process or a different remote process if required.

Any outstanding P_FREAD or P_FWRITE requests will be completed with the E_FILE_CANCEL completion status and a signal generated.

A P_FDISCONNECT request is harmless if no connection has been established or the current connection is temporarily disconnected.

The P_FDISCONNECT request cannot fail and returns zero.

P_FREAD **Read data from the remote process**

```
VOID p_ioc(VOID *pNcp, P_FREAD, WORD *pstat, UBYTE *buf, UWORD *plen);  
INT p_iow(VOID *pNcp, P_FREAD, UBYTE *buf, UWORD *plen);
```

Requests that the next 'message' sent by the remote process be placed in the buffer provided, the length of which be written to *plen.

A 'message' is the data sent by the remote application process in a P_FWRITE request. The contents of the message and its format are entirely determined by the application using the NCP: channel.

The buffer provided must be large enough to hold the largest message that can be sent by the remote application at this point in time (it does not necessarily follow that this is the largest possible message that can be sent). The SYS\$NCP process does not check that the buffer provided is large enough, it simply writes the message into the buffer. If the buffer is not large enough then other data will invariably become corrupted.

The data space pointed at by both the buf and plen parameters must be preserved until the P_FREAD request completes.

The P_FREAD request will typically take a significant length of time and as such should be called asynchronously in a quality system.

The P_FREAD request may be cancelled by using the P_FCANCEL service, the original request will be completed with E_FILE_CANCEL completion status and a signal generated.

A channel connection does not have to exist for a process to queue a P_FREAD request on the channel. In a typical client-server application the client would run the remote server process which would open an NCP: channel and queue a P_FREAD request. The client would queue a P_FCONNECT request and when complete send the server any messages as required using the P_FWRITE service.

The calling process will be panicked if a P_FREAD request is currently outstanding or if pNcp is not a valid channel handle.

The completion status is written to *pstat by an asynchronous request and returned by a synchronous request. The status is zero if the service completed successfully, otherwise it is a negative error number.

Errors include:

E_FILE_LINE or E_FILE_RETRAN	the physical link has failed. This is typically caused by the remote machine switching off or the pack doors being opened. The application process if a server can re-queue a P_FREAD request awaiting the next message. Typically the server should preserve the current state awaiting the next request. A client would typically report the error and await user input before retrying the operation.
E_FILE_DISC	the channel has become disconnected because the remote application process has terminated (either normally or abnormally). The application should tidy up any resources and terminate.
E_GEN_RECEIVER	the remote SYS\$NCP process is terminating or a new one has attempted to connect to the local SYS\$NCP process. The application should tidy up any resources and terminate.

Example

See the client-server example at the end of this chapter.

P_FWRITE**Write data to the remote process**

```
VOID p_ioc(VOID *pNcp, P_FWRITE, WORD *pstat, UBYTE *buf, UWORD *plen);
INT p_iow(VOID *pNcp, P_FWRITE, UBYTE *buf, UWORD *plen);
```

Requests that the 'message' contained in the buffer of length *plen be sent to the connected process.

The data space pointed to by buf must be preserved until the P_FWRITE request completes.

The SYS\$NCP process contains a flow control mechanism such that if a process sends data to a connected process faster than it can handle it, backwards pressure is applied to the sending process by the local SYS\$NCP process. This mechanism is transparent to both the sender and receiver of the message. This flow control is such that a sending process may send data as fast as it likes without the receiver ever being swamped.

The amount of data that can be sent in one message is only restricted to the size of the sending processes data space (64K less the stack, static variables and other allocated cells) however, the receiving process must have a buffer as large as the largest message that can be sent.

A channel must be connected to a remote channel before a P_FWRITE request is made, if not the request will complete with E_FILE_DISC completion status.

When the P_FWRITE request completes successfully this indicates that the remote process to which the message has been sent has received that message. It does not indicate that the remote process has successfully processed that message.

The P_FWRITE request will typically take a significant length of time and as such should be called asynchronously in a quality system.

The P_FWRITE request may be cancelled by using the P_FCANCEL service, the original request will be completed with E_FILE_CANCEL completion status and a signal generated.

The calling process will be panicked if a P_FWRITE request is currently outstanding or if pncp is not a valid channel handle.

The completion status is written to *pstat by an asynchronous request and returned by a synchronous request. The status is zero if the service completed successfully, otherwise it is a negative error number.

Errors include E_FILE_LINE, E_FILE_RETRAN, E_FILE_DISC and E_GEN_RECEIVER, all of which have the same meaning as for the P_FREAD service.

Example

See the client-server example at the end of this chapter.

P_FCANCEL**Cancel any outstanding request**

```
INT p_iow(VOID *pNcp, P_FCANCEL)
```

Requests that any outstanding requests be cancelled, the outstanding requests will complete with the E_FILE_CANCEL completion status and a signal be generated. It is indeterminate as to how much of the message being sent to the remote process (using P_FWRITE) has actually been sent.

The P_FCANCEL request should typically only be used to cancel requests immediately before the application terminates.

The P_FCANCEL request cannot fail and returns zero.

P_FRSUPER**Read supervisory information**

```
INT p_iow(VOID *pNcp, P_FRSUPER, NCLINK_INFO *pinfo, UWORD *plen);
```

This function should only be called by a process that replaces the LINK process.

Requests that the next state change be written back to the supplied buffer.

One process in the system is responsible for receiving the state change messages and responding to them. In the supplied system this is the LINK process. In order for the process receiving the state change messages to ensure that it does not miss any of the messages (and cause potential system deadlock) it should have a process priority higher than that of the SYS\$NCP process. A priority of 0xB0 is adequate.

A connection does not have to exist (and infact should not exist before the first request is made) for this request to be queued sucessfully.

Some state changes are purely informational and can be used as such as required, others require some action before the SYS\$NCP process can continue. The action may be hard coded into the LINK process (typically the non user interface versions) or prompt the user for a solution to the new state.

The following informational state changes are reported:

- `PHYS_PHYS_LINK_ESTABLISHED`, a physical link has been sucessfully established. The `LINK` process on the Series 3 and HC ignores this message.
- `PHYS_WAITING_FOR_CALL`, the physical layer is waiting for an incoming call from the modem driver in order to obtain a physical connection. The `LINK` process on the Series 3 cannot receive this status message as there is no modem driver available. On an HC the message is ignored.
- `PHYS_DIALLING_NUMBER`, the physical layer is currently dialing a phone number in order to obtain a physical connection. The phone number that is being dialled is in the 'phoneno' field of the `DVRS` struct. The `LINK` process on the Series 3 cannot receive this status message as there is no modem driver available. On an HC the message is ignored.
- `PHYS_CONFIGURING_MODEM`, the physical layer is currently waiting for the modem driver to finish sending modem configuration commands. The `LINK` process on the Series 3 cannot receive this status message as there is no modem driver available. On an HC the message is ignored.
- `PHYS_NCP_LINK_ESTAB_OK`, a connection to a remote SYS\$NCP process has been established or re established if the physical or logical connections have previously failed. The `LINK` process on the Series 3 and HC ignores this message. The connection details are contained in the `DVRS` struct.
- `PHYS_NCP_LINK_ESTAB_NEW_NCP`, a connection to a remote SYS\$NCP process has been established, however the remote SYS\$NCP is different to the one we were connected to earlier. The connection details, eg port and baud rate are contained in the `DVRS` struct. The `LINK` process on the Series 3 and HC should respond by sending a `NCLINK_CTRL_NEW_NCP_OK` response.

The following error state changes are reported, they all require some action, the response is sent back to the SYS\$NCP process via the `P_FINQ` service.

- `PHYS_NCP_LINK_ESTAB_INVALID_VER`, the remote SYS\$NCP is version 1.0. We cannot continue with the session since the operation of the two SYS\$NCP's is significantly different. There are very few version 1.0 SYS\$NCP's. The local end should terminate by responding with the appropriate `P_FINQ` message.
- `PHYS_NCP_LINK_END`, the remote SYS\$NCP is terminating, the local end should either prompt for continuation or termination and respond to the SYS\$NCP with the appropriate `P_FINQ` message.
- `PHYS_SERCONFIG_FAILED`, the physical layer has reported that the serial port the user has specified does not exist. The local end should report the error to the user and typically terminate by sending the appropriate `P_FINQ` message.
- `PHYS_CHARS_FAILED`, the physical layer has reported that an attempt to set the serial characteristics failed, presmably because the serial driver does not support the specified configuration. The local end should report the error to the user and typically terminate by sending the appropriate `P_FINQ` message.
- `PHYS_LINK_FAILED`, the logical link layer driver has reported that the link has failed. If the physical connection was over a modem link or a previous logical link connection has been made the local end should ask the user for confirmation to re-try for a physical connection. If no link has ever been established (no `PHYS_NCP_LINK_ESTAB_OK` status message been received) the local end should simply ask the SYS\$NCP process to retry for a link.
- `PHYS_CONNECT_FAILED`, the waiting for a physical connection has failed, typically this is caused by a modem being removed from the serial port whilst waiting for an incoming call to arrive. The local end should either exit or request the physical action be re-tried by sending the appropriate `P_FINQ` response.

- `PHYS_INIT_FAILED`, the physical layer has reported that the initialisation of the modem failed, presumably because of an invalid modem configuration string. The local end should either exit or request the physical action be re-tried by sending the appropriate `P_FINQ` response.
- `PHYS_DIAL_FAILED`, the physical layer has reported that dialling the phone number failed. The local end should either exit or request the physical action be re-tried by sending the appropriate `P_FINQ` response.
- `PHYS_MDMCONFIG_FAILED`, the physical layer has reported that sending additional modem configuration strings has failed. The local end should either exit or request the physical action be re-tried by sending the appropriate `P_FINQ` response.

The `P_FRSUPER` request will typically take a significant length of time. In a high quality system it should be called asynchronously.

The `P_FRSUPER` request may be cancelled by using the `P_FCANCEL` service. The original request will be completed with `E_FILE_CANCEL` completion status and a signal generated.

All `SY$NCP` defines and structure definitions can be found in `sys$ncp.h`.

The calling process will be panicked if a `P_FRSUPER` request is currently outstanding or if `pNcp` is not a valid channel handle.

The `P_FRSUPER` request cannot fail and returns zero.

P_FINQ **Respond to a supervisory message**

```
INT p_iow(VOID *pNcp, P_FINQ, INT response, DVRS *pdvr);
```

This function should only be called by a process that replaces the `LINK` process.

In response to a `P_FRSUPER` event the state change process handler (typically `LINK`) must send back a response.

A response message may be one of the following:

- `NCLINK_CTRL_RETRY`, to retry the action that failed. This would typically retry for a link connection or a physical connection depending on which event occurred earlier. The `pdvr` parameter is irrelevant for this message type.
- `NCLINK_CTRL_EXIT`, to request that all channels be closed, the local `SY$NCP` process inform the remote `SY$NCP` process that it is about to terminate and actually break the physical and logical connections. The `pdvr` parameter is irrelevant for this message type.
- `NCLINK_CTRL_NEW_NCP_OK`, to inform the `SY$NCP` process that the different remote `SY$NCP` process is acceptable and that the controlling process (`LINK`) has sorted out any channels that it is responsible for (the remote filing system channels). The `pdvr` parameter is irrelevant for this message type.

At any point in time a process may inform the `SY$NCP` process that it should restart with new parameters by sending it the following message

- `NCLINK_CTRL_NEW_CONFIG`, restart the `SY$NCP` process with the new configuration as specified in `pdvr`. If any of the parameters are illegal or the request fails then a subsequent `P_FRSUPER` request will receive the failure state.

The `P_FINQ` request cannot fail and returns zero.

All `SY$NCP` defines and structure definitions can be found in `sys$ncp.h`.

P_FSENSE **Sense the current channel activity**

```
INT p_iow(VOID *pNcp, P_FSENSE, NCLINK_SREC *prec);
```

If an application wishes to determine the channel activity occurring within the SYS\$NCP process the statistical information contained in an NCLINK_SREC structure can be obtained.

The prec parameter is assumed to point to an array of 8 NCLINK_SREC structures, one for each of the possible channels that the SYS\$NCP process can handle. Channel 0 is used by the SYS\$NCP process and the information in that array entry should be discarded.

All SYS\$NCP defines and structure definitions can be found in *sys\$ncp.h*.

The P_FSENSE request cannot fail and returns zero.

P_FSTOP **Request the SYS\$NCP terminate**

```
INT p_iow(VOID *pNcp, P_FSTOP);
```

If an application process wishes to terminate the SYS\$NCP process and all connected client server applications it should use the P_FSTOP service.

When the SYS\$NCP process receives this request it will fail all outstanding requests on all open channels immediately with a E_FILE_DISC completion status then tell the remote SYS\$NCP process that it is about to terminate. Any further requests made by any application process on any channel will fail with the E_GEN_NOPROC completion status.

The SYS\$NCP process will not terminate until all channels have been closed by all the application processes.

Server processes will typically always have an outstanding P_FREAD request or be processing a request the result of which is required to be sent back to the client. In both cases it will make a request on the NCP: channel that will fail. If the error handling is as suggested in the P_FREAD and P_FWRITE error section then the server will terminate gracefully.

Client processes on the other hand tend not to make any requests until user input requires them to, hence typically do not have any outstanding or make any requests on the NCP: channel.

In a high quality system the client should close its NCP: channel as soon as it is informed that the SYS\$NCP process is about to terminate.

If the client process is the one that makes the P_FSTOP request is not too much of a problem for it to know to close its channel.

If the client does not make the request it will not know to close its channel. A solution to this is for a client process to always have a P_FREAD request outstanding even though it may never read any data from the remote process. If it does read data from the remote process (presumably after writing a request to the remote process to make the data available) then it can still use the outstanding P_FREAD to read this data.

Example

This example is for a client server application which allows a process to be run on a remote machine. The full path name is specified. The connection establishment and data transfer phases are illustrated in as simple a manner as is possible. For clarity the code does not implement the full error handling required in a working application.

The client side

```

GLDEF_C VOID main(VOID)
{
    WORD len,nlen,pid;
    VOID *pNcp;
    TEXT *p;
    TEXT srvName[20];
    TEXT cmd[80]

    if (pid=p_pidfind("LINK.*"))<0)
        p_exit(1);
    if (p_open(&pNcp,"NCP:", -1))
        p_exit(1);
    p_scpy(&srvName[0], "EXECSRV");
    p=(&srvName[0]);
    if (p_msndreceivew(pid, LNKMSG_LOADREMOTE, &p)<0)
        p_exit(1);
    len=p_getl("Name and command line",&cmd[0],80);
    if (len)
    {
        nlen=p_slen(&srvName[0])+1;
        if (!p_iow(pNcp,P_FCONNECT,&srvName[0],&nlen))
        {
            if (!p_iow(pNcp,P_FWRITE,&cmd[0],&len))
            {
                if (!p_iow(pNcp,P_FREAD,&cmd[0],&len))
                {
                    pid=cmd[0]+(cmd[1]<<8);
                    if (pid>0)
                        p_printf("Remote pid %x",pid);
                    else
                        p_puts("No such process");
                }
            }
            p_iow(pNcp,P_FDISCONNECT);
        }
    }
    p_close(pNcp);
    p_exit(0);
}

```

The server side

```

GLDEF_C VOID main(VOID)
{
    WORD len,pid;
    VOID *pNcp;
    TEXT *pname,*pcmd;
    TEXT cmd[80]

    if (p_open(&pNcp,"NCP:", -1))
        p_exit(1);
    if (!p_iow(pNcp,P_FREAD,&cmd[0],&len))
    {
        pname=p_skipwh(&cmd[0]);
        pcmd=p_skipch(pname);
        if (*pcmd)
        {
            *pcmd++=0; /* zero terminate proc name */
            pcmd=p_skipch(pcmd);
        }
        pid=p_exec(pname,pcmd,p_slen(pcmd));
        cmd[0]=pid;
        cmd[1]=pid>>8; /* return pid or error */
        len=2;
        p_iow(pNcp,P_FWRITE,&cmd[0],&len);
    }
    p_close(pNcp);
    p_exit(0);
}

```


CHAPTER 11

CRADLE AND DOCKING STATION

Introduction

This chapter refers to two units: the HC cradle and the docking station for HC and *Workabout* computers. In this document the term 'Cradle' refers to the HC cradle, (which connects to the side pins of older HC models). The term 'Docking Station' refers both to the HC docking station, (which connects to an HC LIF interface fitted to the bottom of the HC), and to the *Workabout* docking station. The term 'computer' refers to an HC or *Workabout*.

This chapter describes the services which are supported by the Cradle/Docking Station device driver (CRD:). The Fast Charger services provided by the Docking Station are described in the *Fast Charger* chapter.

The Cradle/Docking Station device driver (CRD:) reports changes of state when a computer is inserted or removed from the Cradle or Docking Station. It is supplied to allow a program to perform specific operations automatically when the computer is inserted into the Cradle or Docking Station and to 'tidy up' when the computer is removed.

Notes:

1. the Cradle/Docking Station device does not have to be open to use the Cradle/Docking Station expansion port. The operating system will automatically stop and start active devices in the cradle.
2. the "Cradle/Docking Station in" signal is generated when the computer first touches the connector, but software running on the computer must wait until the connection is fully home before attempting to access any device in the Cradle/Docking Station.

If a user inserts the computer into the cradle slowly, EPOC may not recognise that the Cradle/Docking Station expansion port is present. In such a case it may be prudent to open and close an expansion device using the Cradle/Docking Station device as an indicator of the connection state. If an attempt to open a device when the computer is inserted into the Cradle/Docking Station fails, a retry can be attempted after a delay of, say, two seconds.

Cradle/Docking Station services

p_open(CRD:)

Open the device

```
INT p_open(VOID **ppcb, "CRD:", -1);
```

Open a channel to the current Cradle or Docking Station device, as set by any previous call to the P_FSET service. If there has been no previous call to this service, or if the machine has just been reset, it will open a channel to the Cradle/Docking Station.

Returns zero if the device is opened successfully, otherwise a negative error. Errors include:

E_GEN_NOMEMORY	failed to allocate memory for control block
E_FILE_LOCKED or	port is already open or
E_GEN_INUSE	in use

p_close**Close the channel**

```
INT p_close(VOID *pcb)
```

Close the channel. Returns zero.

P_FREAD**Read from the device**

```
VOID p_ioc(VOID *pcb, P_FREAD, WORD *pstat, UWORD *pcstate);  
INT p_iow(VOID *pcb, P_FREAD, UWORD *pcstate);
```

Read a change of state from the Cradle/Docking Station, writing the new state to *pcstate.

When called for the first time after opening the device it will complete immediately, reporting the current state. Thereafter it will complete whenever the computer is removed from or inserted into the Cradle/Docking Station.

On completion *pcstate is TRUE if the computer is currently in the Cradle/Docking Station and FALSE if the computer is out of the Cradle/Docking Station.

Panics if a P_FREAD request is currently outstanding, or if pcb is not a valid channel handle.

The completion status code is returned by the synchronous p_iow(P_FREAD) and written to *pstat by asynchronous calls. The completion status code is zero if the P_FREAD request completed successfully, or E_FILE_CANCEL if the read was cancelled.

P_FCANCEL**Cancel a read**

```
INT p_iow(VOID *pcb, P_FCANCEL);
```

Cancel any outstanding P_FREAD request. Performing a cancel is harmless if no read request is outstanding.

Returns zero.

P_FSET**Set the device type**

```
INT p_iow(VOID *pcb, P_FSET, WORD *pctype);
```

If *pctype is TRUE, sets all future CRD: operations to apply to the Docking Station and all accesses to port C and related ports to apply to any expansion fitted into the Docking Station. It will also check that a valid Docking Station expansion is fitted to the bottom of the HC. Since this call requires use of the high speed serial channel on the bottom of the HC and affects all port C devices, it will fail with E_GEN_INUSE if anything is open on port B or port C.

If *pctype is FALSE, sets all future CRD: operations to apply to the Cradle and all accesses to port C and related ports to apply to any expansion fitted into the Cradle. Since this call affects all port C devices, it will fail with E_GEN_INUSE if anything is open on port C.

Note: This setting will remain even after the CRD: device is closed. It will remain in force until it is changed by another call to P_FSET, or until the machine is reset. It is envisaged that one call will be made to this function when an application first runs, and that this will set the device type for the life of the application.

E_GEN_NOMEMORY	failed to allocate memory for the control block
E_GEN_INUSE	the port is in use (e.g. TTY:B is open when trying to set to the Docking Station, or TTY:C in the Docking Station is open when trying to set to the Cradle)

P_FSENSE**Sense the device type**

```
INT p_iow(VOID *pcb, P_FSENSE, WORD *pctype);
```

The current device type is returned in *pctype. If set to TRUE then the Docking Station is the current type, if set to FALSE then the Cradle is the current type.

Note: This service does not check if the hardware is actually present (e.g. if an HC LIF interface is fitted) and therefore may be called at any time. To check if the hardware is present, use the P_FSET service.

Returns zero.

CHAPTER 12

HC MAGNETIC CARD READER

Introduction

The HC Magnetic Card Reader (MCR:) device driver is built into the HC's operating system.

The MCR interface may be fitted to the top (MCR:A) or bottom (MCR:B) of the HC, or in the cradle (MCR:C).

MCR services

p_open(MCR:)

Open the MCR device

```
INT p_open(VOID **ppcb, "MCR:A", -1);
```

Open a channel to a Magnetic Card Reader device.

Returns zero if the device was opened successfully, otherwise a negative error. Errors include:

E_GEN_NOMEMORY	failed to allocate memory for control block
E_FILE_DEVICE	no interface found in specified slot
E_FILE_NAME	invalid device name
E_FILE_LOCKED or	port is already open or
E_GEN_INUSE	in use

p_close

Close the channel

```
INT p_close(VOID *pcb)
```

Close the Magnetic Card Reader device channel.

Returns zero.

P_FREAD

Read from the MCR device

```
VOID p_ioc(VOID *pcb, P_FREAD, WORD *pstat, UBYTE *buf1, UBYTE *buf2);  
INT p_iow(VOID *pcb, P_FREAD, UBYTE *buf1, UBYTE *buf2);
```

Read either or both tracks of a card. Track 1 is read into *buf1 and track 2 into *buf2, where each buffer must be at least 256 bytes long. The data in each buffer is written as leading byte count ASCII text. The leading byte count is zero if the read was not successful.

Either buf1 or buf2 may be passed as NULL to disable reading of the relevant track. For example, to read only track 2:

```
UWORD stat;  
UBYTE buf[256];  
  
p_ioc(pcb, P_FREAD, &stat, NULL, &buf[0]);  
p_waitstat(&stat);
```

Track 1 usually contains alphanumeric data (card holder's name and account number) while track 2 contains numeric data only. The most common MCRs read only track 2.

Panics if a P_FREAD request is currently outstanding, or if pcb is not a valid channel handle.

The completion status code is returned by the synchronous p_iow(P_FREAD) and written to *pstat by asynchronous calls. The completion status code is zero if the P_FREAD request completed successfully, or one of the following negative error numbers:

E_FILE_READ	an error was detected decoding the data
E_GEN_OVER	device driver buffer overflow
E_FILE_CANCEL	the read was cancelled

P_FCANCEL **Cancel a read**

```
INT p_iow(VOID *pcb, P_FCANCEL);
```

Cancel any outstanding P_FREAD request. Performing a cancel is harmless if no read request is outstanding.

Returns zero.

P_FSET **Set the pull-up resistors**

```
VOID p_iow(VOID *pcb, P_FSET, UWORD *mask);
```

This service is only available in EPOC versions 2.32 or later.

Set the programmable pull-up/pull-down resistors on the five MCR reader lines.

The MCR device contains programmable 100k ohm pull-up/pull-down resistors on each of the reader lines DATA1, CLK1, DATA2, CLK2 and CLS. By default when the MCR device is opened these are set to pull down.

The resistor on each line is controlled by a bit in *mask. If the bit is set to 1 the resistor is programmed to pull-up, if the bit is cleared to 0 the resistor is programmed to pull-down.

The bits which control each line should be defined in the application source file, according to the following table:

Symbol	Value (binary)	Line
M_DATA1PU	00000001	Track 1 data
M_CLK1PU	00000010	Track 1 clock
M_DATA2PU	00000100	Track 2 data
M_CLK2PU	00001000	Track 2 clock
M_CLSPU	00010000	Card present signal

All other bits are ignored.

CHAPTER 13

HC BAR CODE READER

Hardware Description

The Bar code reader interface module

The bar code interface module is a grey plastic moulding which is designed to fit into one of the slots at either end of the HC. To insert the module you will need to open the rear door of the HC, and unlock one of the interface slots to remove the existing module or blank. Place the bar code reader module into the slot and gently ease it home making sure it fits snugly. Once this is done you will need to lock the module in place by moving the black switch into the locked position. After this close the door again otherwise the HC will refuse to switch on.

The interface may be fitted to the top (BAR:A) or bottom (BAR:B) of the HC.

The bar code reader wand.

The bar code wand is supplied with a locking mini-din plug which fits into the mini-din socket in the top of the bar code module. Simply lining up the grooves and pushing the plug home will lock the plug into place. The locking plug type is used so that if the HC is accidentally suspended by the bar code reader cable, the machine will not come off the connector. To remove the wand from the module grip the plastic moulding which surrounds the plug and pull gently. The plastic cover will slide back slightly releasing its grip on the module. From there the plug should easily slide out.

Device drivers

At the time of writing, there is no bar code device driver or decoding software built into the HC's operating system. Bar code readers are supported by separate device driver files which must be loaded by the application code, by means of either the PLIB `p_loadldd` function or the OPL `DevLoadLDD` call.

There are currently five combined decoder/device drivers:

BAREAN.LDD	supports EAN8, EAN13, UPC and UPCE decoding
BARC39.LDD	supports CODE 39 decoding
BARITF.LDD	supports Interleaved 2 of 5 (ITF) decoding
BAR128.LDD	supports CODE 128 decoding
BARMPLS.LDD	supports Modified Plessey decoding
BARRAW.LDD	supports raw decoding

None of the currently available drivers supports auto discrimination of bar codes.

Installing the LDD optional component of the SDK copies these files into the `\sibosdk\lib` directory. The required LDD must be copied into the appropriate directory on the HC using `MCLINK`.

A device driver may be loaded, for example, by:

```
p_loadldd("BARC39.LDD");
```

To remove the device (and free the memory it uses) you should call

```
p_devdel("BAR");
```

or the equivalent OPL `DevDelete` call.

The bar code device driver software is updated from time to time; update information is available from Psion Support.

Bar code driver services

p_open(BAR:) **Open the bar code device**

```
INT p_open(VOID **ppcb, "BAR:", -1);
```

Open a channel to a bar code device by means of a previously loaded bar code device driver.

Returns zero if the device was opened successfully, otherwise a negative error. Errors include:

E_GEN_NOMEMORY	failed to allocate memory for control block
E_FILE_DEVICE	no interface found in specified slot
E_FILE_NAME	invalid device name
E_FILE_LOCKED or E_GEN_INUSE	port is already open or in use

p_close **Close the channel**

```
INT p_close(VOID *pcb)
```

Close the bar code device channel.

Returns zero.

P_FREAD **Read from the MCR device**

```
VOID p_ioc(VOID *pcb, P_FREAD, WORD *pstat, UBYTE *buf);  
INT p_iow(VOID *pcb, P_FREAD, UBYTE *buf);
```

Read a bar code into *buf, which must be at least 256 bytes long. The data is written as leading byte count ASCII text. The leading byte count is zero if the read was not successful. The first character of the text indicates the type of the bar code:

A	EAN8 or EAN13
B	UPC
C	Code 39
D	ITF
I	Code128
M	Modified Plessey
U	UPCE

The remainder of the text is the decoded bar code data.

Panics if a P_FREAD request is currently outstanding, or if pcb is not a valid channel handle.

The completion status code is returned by the synchronous p_iow(P_FREAD) and written to *pstat by asynchronous calls. It is zero if the P_FREAD request completed successfully, or one of the following negative error numbers:

E_GEN_OVER	device driver buffer overflow
E_FILE_CANCEL	the read was cancelled

P_FCANCEL **Cancel a read**

```
INT p_iow(VOID *pcb, P_FCANCEL);
```

Cancel any outstanding P_FREAD request. Performing a cancel is harmless if no read request is outstanding.

Returns zero.

CHAPTER 14

HC INTELLIGENT BAR CODE READER/RS232 PORT

See the *HC Bar Code Reader* chapter, for the description of the device driver for an alternative HC bar code reader expansion module.

RS232/intelligent bar code reader module

The RS232/intelligent bar code reader is an expansion module which contains an intelligent bar decoding micro controller. The module is fully compatible with the HC, HC-DOS and RWAN series of computers.

The expansion module provides one serial port connection to the HC computer, and this port is shared between the two interfaces. The serial port may be configured by software to open a channel *either* to the PC-AT style RS232 interface, *or* to the intelligent bar code scanner interface; both interfaces cannot be used simultaneously. This module has been designed to allow an HC computer to read bar code labels, using a wand attached to the bar code reader interface. Data collected from bar code scans may then be transferred to an external computer (such as an IBM PC-AT compatible machine) from the HC computer, via the RS232 serial port interface.

The bar code reader interface contains an intelligent micro controller that will automatically read and decode data from a bar code wand (or a wand emulator) connected to the expansion module. The unit can read and automatically discriminate between the following bar code formats:

- EAN/JAN 8
- EAN/JAN 13
- UPC A
- UPC E
- Codabar
- Code 128
- Interleaved 2 of 5
- Code 39 (standard or extended)

The interface may be programmed to verify scanned data against check digits/characters on bar code labels and to read, or to ignore any supplement digits in a bar code, as required.

The micro controller will transmit bar code data to the HC computer as an ASCII string of characters: i.e. each digit or character read from a bar code label will be transmitted to the HC computer as one ASCII-coded character. The order of the bar code data will always be transmitted to the HC computer in the correct order (i.e. reading from left to right across the bar code label), irrespective of the direction that the bar code label was actually scanned. By default, the end of each complete bar code scan will be marked by a single ASCII carriage return character.

The micro controller inside the expansion module is programmable. It may be instructed to decode only a subset of the bar code symbologies that it recognises, to transmit check digits and check characters with each bar code scan, read or ignore supplement digits, as well as many other bar code symbology-specific options. The micro controller can also be programmed to mark the end of each scan with a customised ASCII terminating string of up to four characters, in place of the default carriage return termination character. This document contains detailed information on how to program the micro controller from C, and from OPL programs.

You may connect this expansion module either to the top slot, or to the bottom slot of an HC computer. When the expansion module has been plugged into the top slot on the computer, the RS232 port can be accessed by opening TTY:A, and the bar code interface may be accessed by opening TTY:D; if the module has been connected to the bottom slot on the computer, then the RS232 port can be accessed by opening TTY:B and the bar code interface may be accessed by opening TTY:E.

The bar code interface communicates to the host computer at the following (standard HC Comms) settings:

- Baud
- data bits
- stop bit
- Xon/Xoff handshaking (No hardware handshaking)

The expansion module possesses two, male 9-pin D-type connectors: the RS232 connector is a standard PC-AT type connector¹; the bar code interface connector is a standard click-lock D-type connector for a bar code wand.

To reduce power consumption, the bar code interface and the RS232 port interface are only powered up when the channel to the serial port is open.

The RS232 serial port interface

The RS232 interface provides standard RS232 level signals to a 9-pin D-type male connector. The connector is PC-AT compatible, although pin 9 (normally the RI pin) is not driven by the interface. When the interface unit has been connected to the top slot of the HC, then the RS232 port may be accessed by opening the TTY:A device. If the unit has been connected to the bottom slot of the HC, then the RS232 port may be accessed by opening TTY:B.

The table below displays the pinout for the PC-AT type RS232 serial port.

1	DCD input
2	RX input
3	TX output
4	DTR output
5	Ground (0v)
6	DSR input
7	RTS output
8	CTS input
9	Optional Vsup connection

Power consumption

The RS232 port is only powered up when the appropriate channel is open. The interface will draw approximately 10mA, plus the current drawn by the device connected to the other end of the RS232 cable. If the remote device is a PC-AT type computer, then the total current drawn by the interface will typically be 20mA, although this figure may vary from one PC to another.

When the HC computer is also powering an external device through the RS232 port, then that external device should not draw more than:

- 250mA from the HC if **no** other expansion modules are attached to the HC, or
- 200mA from the HC if another expansion module **is** attached to the HC and is powered up.

Please note also that external device will draw current from the HC computer even when the HC computer has been switched off. Consequently, the external device ought to have its own on/off switch, and to avoid excessive battery drain, the external device should be switched off when it is not in use.

¹The ringing indicator (RI) pin has not been implemented on the RS232 port. It may, however, be connected permanently to the HC computer Vsup power supply rail, if required. This action will allow an external unit such as an infrared laser scanner to have power supplied to it through the RI connection in the serial link cable, from the HC computer.

Powering the HC from an external power source

The HC computer may be powered by an external source through pin 9 of the RS232 interface. To do this, a 2-pin header on the RS232/bar code interface expansion module PCB should be shorted together with a jumper. (The 2-pin header is located at the extreme bottom left-hand side of the PCB, if the expansion module is oriented so that the electrical components are uppermost and the two D-type connectors face upwards.)

This alteration will route the *v_{sup}* main power supply rail from pin 9 of the serial port to the HC computer. The external *v_{sup}* power supply should be in the range of 7-10v. The external power supply **should be diode-isolated** from the HC to prevent any power drain from the HC computer.

Please note that the main battery inside the HC computer will not be charged if power is supplied to the HC from the RS232 port.

DSR auto wakeup switch

If required, the interface and HC computer may be woken up by an external device whenever the DSR pin of the RS232 link is asserted. To enable this facility, move the PCB switch (located at the bottom left-hand corner of the circuit board) to the left-hand position. (If you hold the module with the component side of the PCB facing you and the D-type connectors facing upwards, the switch is located to the bottom left hand corner of the PCB.)

The bar code interface

The bar code reader interface contains a Hewlett-Packard HBCR-1610 series bar decoding micro controller which is capable of reading and discriminating between the following bar code formats: EAN/JAN 8, EAN/JAN 13, UPC A, UPC E, Codabar, Code 128, Interleaved 2 of 5 and standard or extended Code 39 bar labels.

The interface possesses a 9-pin D-type click-lock male connector which may be attached to many standard digital wands and wand emulating scanner units. However undecoded laser scanners (HHLC) are **not** supported by this interface.

When the interface unit has been connected to the top slot of the HC, then the bar code port may be accessed by opening the `TTY:D` device. If the unit has been connected to the bottom slot of the HC, then the bar code port may be accessed by opening `TTY:E`. To conserve power, the bar code interface is only powered up while the serial port is open.

The bar code interface communicates to the host computer with the following settings:

- Baud
- data bits, no parity and 1 stop bit
- Parity errors are not ignored

These are the default serial port settings for the HC computer. However, in addition, the bar code port must also be set up so that:

- the `tmask` parameter of the serial port characteristics identifies the last character in the terminating string for each incoming message, from the bar code interface. The default terminating string is a single carriage return character: i.e. `tmask=2000`.
- the CTS/RTS handshaking protocol is disabled, by setting the `IGN_CTS` flag.

Full details are given later in this section, which describe how `tmask` and `IGN_CTS` may be set from within a C or an OPL program. Please note that these two settings are only required to configure the bar code ports `TTY:D` and `TTY:E` for the bar code interface; they do not apply to the RS232 ports `TTY:A` and `TTY:B`.

The table below displays the pinout for the two expansion module ports.

1	DCD input
2	Bar data input
3	Not connected
4	Switched <i>v_{sup}</i> output
5	DSR input
6	DTR output
7	Ground (0v)
8	Ground (0v)
9	Switched 5v regulated output

Power consumption

External units should not draw more than:

- 250mA from the HC if **no** other expansion modules are attached to the HC, or
- 200mA from the HC if another expansion module **is** attached to the HC, and is powered up.

The bar code port draws an idle current of 10mA, and typically 24mA when a scan is in progress. This figure does **not** include the additional current drawn by the wand, or scanner, attached to the interface. To conserve power, the RS232 interface is only powered up while the serial port is open.

Powering a bar code wand from the HC

The HC computer can supply a 5v ($\pm 5\%$) regulated supply and a v_{sup} (6-10v) supply to an external bar code wand, or a wand emulating device (such as an RS232 laser scanner). These two power supplies are available on pin 9 and pin 4, respectively, on the bar code interface connector.

To allow the HC to supply power an external unit a simple adjustment must be made to the expansion module: a 2-pin header on the PCB of the RS232/bar code interface expansion module must be shorted together with a jumper. (The 2-pin header is located at the extreme bottom left-hand side of the PCB, if the expansion module is oriented so that the electrical components are uppermost and the two D-type connectors are facing upwards.)

The two power supply outputs on the bar code interface (and the decoding IC) are switched on only when the port is open. As a result, **the power will be switched off** when the HC computer powers down - if it is left unused for longer than the timeout period. Please note that all of the custom settings programmed into the bar code interface with escape sequences will be lost when the port is closed, or if the HC auto-powers down.

Please note that the 2-pin header should **only** be taken if the external unit must be powered by the HC, or if the HC must be powered by the external device.

Bar code symbologies

The bar code reader interface will transmit data from each bar code scan as a stream of ASCII text data. Each data stream will be terminated by the default termination string (a carriage return character, $0x0d$). Alternatively, the interface may be programmed to terminate each stream of scan data with a custom termination string of up to four characters.

Bar code data will always be transmitted to the host computer in the correct order (i.e. from left to right), irrespective of the direction in which the bar label was originally scanned. Bar code labels containing supplement digits can only be scanned in the forwards direction (i.e. from left to right); all other bar code labels may be scanned either forwards or backwards. The interface may be programmed to include, or strip out, check characters and ID characters in the transmitted data, as desired. The maximum scanning rate for any bar code format is 30 ips

The remainder of this section will now describe each bar code format that may be scanned by the interface. In each case, the programmable options that are associated with each format will also be described.

Code 128

Code 128 labels contain a variable number of digits, and one check character. There are three types of Code 128 bar code labels that may be decoded by the bar code interface: code A, code B and code C. Both Code A and Code B labels may contain a maximum of 31 characters; Code C labels may contain a maximum of 62 characters.

No user-definable options are available to alter Code 128 bar code data. The bar code reader interface will always check the label data against the check character on the bar code label, but the check character will never be transmitted as part of the scanned data.

Codabar

Codabar labels contain one start character, a number of digits and one stop character. Only one user-definable option exists with Codabar format data: the bar code reader interface may be programmed to transmit, or to ignore any start and stop characters in the label data.

The start and stop characters may be any of the following four upper case characters: 'A', 'B', 'C', or 'D'. A start character does not have to be the same as the stop character.

The table below illustrates the effect that each Codabar option will have on the output data from the bar code interface.

Start/stop chars Input label data Output data

Transmit	A123456B	123456
Ignore	A123456B	A123456B

Interleaved 2 of 5

An Interleaved 2 of 5 bar code will always contain an even number of digits (including the check digit) and no characters. It may also contain a number of additional check sum digits also. The maximum number of digits that may be contained within a label is 32; the minimum number is 2. The interface may be programmed to read either:

- any Interleaved 2 of 5 labels containing any even number of digits within the range of 2 and 32 digits,
- only Interleaved 2 of 5 labels containing 6 or 14 digits, or
- only Interleaved 2 of 5 labels containing a preset, even number of digits.

There will always be one check digit at the end of the bar code label. The bar code scanner interface may be programmed to verify the scanned data against this check digit, and it may also be programmed either to transmit the check digit as part of the output data string, or to omit the check digit from the output data.

In the table below, 123456 represents a bar code scan containing the valid check digit 6 and 123457 represents a scan that contains the invalid check digit 7. The output generated by the bar code interface is displayed in the right-hand column.

Input data	Verify the check digit?	Transmit the check digit ?	Output data
123456	No	Yes or No	123456
123457	No	Yes or No	123457
123456	Yes	No	12345
123457	Yes	No	No output
123456	Yes	Yes	123456
123457	Yes	Yes	No output

Code 39

A Code 39 bar code label may contain a minimum of one character and a maximum of 32 characters. It will contain no digits. The bar code reader interface may be programmed to verify the label check character, and to transmit the check character in the output data message, or strip the check character from the output data.

In the following table, ABCX represents a Code 39 label containing the valid check character x, and the string ABCD represents a Code 39 label containing an invalid check character d.

Input label data	Verify the check character?	Transmit the check character	Output data
ABCD	No	Yes or No	ABCD
ABCX	No	Yes or No	ABCX
ABCD	Yes	No	No output
ABCX	Yes	No	ABC
ABCD	Yes	Yes	No output
ABCX	Yes	Yes	ABCX

The user interface may be programmed in Code 39 scans may be converted as character pairs (as defined by the Code 39 symbology) or alternatively, each character in the scan may be decoded individually.

The UPC/EAN bar code formats

The UPC/EAN bar code family all contain a fixed number of digits, and all contain one check digit. The bar code reader interface may be programmed to verify this check digit against the rest of the scanned data. The interface may also be programmed to transmit the check digit as part of the bar code, or to omit the check digit from the output data.

There twelve variants in the UPC/EAN family of bar codes. Note that JAN 8 labels are equivalent to EAN 8 format labels and JAN 13 labels are equivalent to EAN 13 format labels.

UPC E

The UPC E bar code label format contains a leading ID character, one number system digit, six data digits and one check digit. The leading ID character must be an ASCII 'E'. The check digit will be followed by 2 supplement digits in UPC E + 2 digit labels, and with 5 supplement digits in UPC E + 5 digit labels.

The bar code interface may be programmed to remove the ID character, or the check digit from the data transmitted to the host computer. UPC E bar code label data may also be expanded into UPC A bar code format automatically by the interface unit - in which case the six existing data digits will be expanded to ten digits, and the leading ID character will become an ASCII 'A'. Refer to the UPC A section in this chapter for more information about this label format.

The following table displays the four different user-definable options available for modifying the output data from UPC E scans.

Transmit ID char	Transmit check digit	Standard output (UPC E)	Expanded output (UPC A)
No	No	dddddd	nddddddddd
Yes	No	Enddddd	Andddddddd
No	Yes	dddddd	ndddddddddcc
Yes	Yes	Enddddd	Anddddddddcc

where:

E = an ID character 'E' (0x45)

A = an ID character 'A' (0x41)

d = one data digit

n = one number system digit

c = one check digit

s = one supplement digit

UPC E + 2 digits

As its name suggests, the UPC E + 2 bar code label format contains a leading ID character, one number system digit, six data digits, one check digit and two supplement digits. The leading ID character must be an ASCII 'E'. UPC E bar code label data may also be expanded into UPC A bar code format automatically by the interface unit - in which case the six existing data digits will be expanded to ten digits, and the leading ID character will become an ASCII 'A'.

The bar code interface may be programmed to remove the ID character, or the check digit from the output data transmitted to the host computer. The following table displays the four different options available:

Transmit ID char	Transmit check digit	Standard output (UPC E)	Expanded output (UPC A)
No	No	ddddddss	ndddddddddss
Yes	No	Enddddddss	Anddddddddss
No	Yes	ddddddss	ndddddddddcss
Yes	Yes	Enddddddss	Anddddddddcss

where:

E = an ID character 'E' (0x45)

A = an ID character 'A' (0x41)

d = one data digit

n = one number system digit

c = one check digit

s = one supplement digit

UPC E + 5 digits

As its name suggests, the UPC E + 5 bar code label format contains a leading ID character, one number system digit, six data digits, one check digit and five supplement digits. The leading ID character must be an ASCII 'E'. UPC E bar code label data may also be expanded into UPC A bar code format automatically by the interface unit - in which case the six existing data digits will be expanded to ten digits, and the leading ID character will become an ASCII 'A'.

The bar code interface may be programmed to remove the ID character, or the check digit from the output data transmitted to the host computer. The following table displays the four different options available:

Transmit ID char	Transmit check digit	Standard output (UPC E)	Expanded output (UPC A)
No	No	ddddddsssss	ndddddddddsssss
Yes	No	Enddddddsssss	Anddddddddsssss
No	Yes	ddddddsssss	ndddddddddcsssss
Yes	Yes	Enddddddsssss	Anddddddddcsssss

where:

E = an ID character 'E' (0x45)

A = an ID character 'A' (0x41)

d = one data digit

n = one number system digit

c = one check digit

s = one supplement digit

EAN 8 bar code format

The EAN 8 bar code label format contains the following components: two leading ID characters, two flag digits, five data digits and one check digit. The two leading ID characters will be the upper case string "FF".

The bar code interface may be programmed to remove the two ID characters, or the check digit from the output data transmitted to the host computer. The following table displays the four different options available:

Transmit ID chars	Transmit check digit	Output
No	No	ffddddd
Yes	No	FFffddddd
No	Yes	ffdddddcc
Yes	Yes	FFffdddddcc

where:

F = an ID character 'F' (0x46)

f = one flag digit

d = one data digit

c = one check digit

EAN 8 + 2 digits

The EAN 8 + 2 bar code label format contains the following components: two leading ID characters, two flag digits, five data digits, one check digit and two supplement digits. The two leading ID characters will be the upper case string "FF".

The bar code interface may be programmed to remove the two ID characters, or the check digit from the output data transmitted to the host computer. The following table displays the four different options available:

Transmit ID chars	Transmit check digit	Output
No	No	ffdddddss
Yes	No	FFffdddddss
No	Yes	ffdddddcss
Yes	Yes	FFffdddddcss

where:

F = an ID character 'F' (0x46)

f = one flag digit

d = one data digit

c = one check digit

s = one supplement digit

EAN 8 + 5 digits

The EAN 8 + 5 bar code label format contains the following components: two leading ID characters, two flag digits, five data digits, one check digit and five supplement digits. The two leading ID characters will be the upper case string "FF".

The bar code interface may be programmed to remove the two ID characters, or the check digit from the output data transmitted to the host computer. The following table displays the four different options available:

Transmit ID chars	Transmit check digit	Output
No	No	ffddddsssss
Yes	No	Fffddddsssss
No	Yes	ffddddcssss
Yes	Yes	Fffddddcssss

where:

F = an ID character 'F' (0x46)

f = one flag digit

d = one data digit

c = one check digit

s = one supplement digit

EAN 13

The EAN 13 bar code label format contains the following components: a leading ID character, two flag digits, ten data digits and one check digit. The leading ID character will be an ASCII upper case 'F'.

The bar code interface may be programmed to remove the ID character, or the check digit from the output data transmitted to the host computer. The following table displays the four different options available:

Transmit ID char	Transmit check digit	Output
No	No	ffddddddddd
Yes	No	Fffddddddddd
No	Yes	ffdddddddddcc
Yes	Yes	Fffdddddddddcc

where:

F = an ID character 'F' (0x46)

f = one flag digit

d = one data digit

c = one check digit

EAN 13 + 2 digits

The EAN 13 + 2 bar code label format contains the following components: a leading ID character, two flag digits, ten data digits, one check digit and two supplement digits. The leading ID character will be an ASCII upper case 'F'.

The bar code interface may be programmed to remove the ID character, or the check digit from the output data transmitted to the host computer. The following table displays the four different options available:

Transmit ID char	Transmit check digit	Output
No	No	ffdddddddddss
Yes	No	Fffdddddddddss
No	Yes	ffdddddddddcss
Yes	Yes	Fffdddddddddcss

where:

F = an ID character 'F' (0x46)

f = one flag digit

d = one data digit

c = one check digit

s = one supplement digit

EAN 13 + 5 digits

The EAN 13 + 5 bar code label format contains the following components: a leading ID character, two flag digits, ten data digits, one check digit and five supplement digits. The leading ID character will be an ASCII upper case 'F'.

The bar code interface may be programmed to remove the ID character, or the check digit from the output data transmitted to the host computer. The following table displays the four different options available:

Transmit ID chars	Transmit check digit	Output
No	No	ffddddddddsssss
Yes	No	Fffddddddddsssss
No	Yes	ffddddddddcsssss
Yes	Yes	Fffddddddddcsssss

where:

F = an ID character 'F' (0x46)

f = one flag digit

d = one data digit

c = one check digit

s = one supplement digit

UPC A

The UPC A bar code label format contains a leading ID character, one number system digit, ten data digits and one check digit. The leading ID character must be an ASCII 'A'.

The bar code interface may be programmed to remove the ID character, or the check digit from the output data transmitted to the host computer. The following table displays the four different options available:

Transmit ID chars	Transmit check digit	Output
No	No	nddddddddd
Yes	No	Anddddddddd
No	Yes	nddddddddcdc
Yes	Yes	Anddddddddcdc

where:

A = an ID character 'A' (0x41)

n = one number system digit

d = one data digit

c = one check digit

UPC A + 2 digits

The UPC A + 2 bar code label format contains a leading ID character, one number system digit, ten data digits, one check digit and two supplement digits. The leading ID character must be an ASCII 'A'.

The bar code interface may be programmed to remove the ID character, or the check digit from the output data transmitted to the host computer. The following table displays the four different options available:

Transmit ID chars	Transmit check digit	Output
No	No	ndddddddddss
Yes	No	Andddddddddss
No	Yes	nddddddddcss
Yes	Yes	Anddddddddcss

where:

A = an ID character 'A' (0x41)

n = one number system digit

d = one data digit

c = one check digit

s = one supplement digit

UPC A + 5 digits

The UPC A + 5 bar code label format contains a leading ID character, one number system digit, ten data digits, one check digit and five supplement digits. The leading ID character must be an ASCII 'A'.

The bar code interface may be programmed to remove the ID character, or the check digit from the output data transmitted to the host computer. The following table displays the four different options available:

Transmit ID chars	Transmit check digit	Output
No	No	nddddddddsssss
Yes	No	Anddddddddsssss
No	Yes	nddddddddcssss
Yes	Yes	Anddddddddcssss

where:

A = an ID character 'A' (0x41)

n = one number system digit

d = one data digit

c = one check digit

s = one supplement digit

Bar code commands

This section describes the commands that may be used to program the HC bar code reader interface.

The bar code interface is programmed by writing an escape sequence to the interface unit via a serial port. With two exceptions, each escape sequence consists of a short text string in the following format:

```
<Esc>-y<code><command>
```

where:

1. <Esc> is the escape character (0x1b)
2. - is a '-' character (0x2d)
3. y is a 'y' character (0x79) or a 'Y' character (0x59)
4. <code> is a parameter to the command, consisting of sequence of one to three numeric digit, representing a decimal number between zero and 255.
5. <command> is an upper case alphabetic character identifying the command to be executed.

The two exceptions are the hard reset command:

<Esc>E

and the command to set the data termination string, which includes additional text following the letter that identifies the command:

<Esc>-y<code><command><text>

In all cases the escape sequence must not contain any embedded spaces.

Commands to the bar code reader interface do not have to be issued individually. The *Issuing multiple commands* section in this chapter describes how several commands may be issued as a multiple escape sequence, within one text string.

Multiple options in a command

Many of the bar code interface commands offer more than one option, selected by the <code> value. To select multiple options from a single command, simply sum all the required individual <code> values together, and then pass this summed value as the <code> value within the escape sequence.

For example, the *Select bar code symbology* command <Esc>-y<code>F (described later in this chapter) contains options to select five different bar code formats. For example, a <code> value of 1 selects Code 39, a value of 4 selects Codabar and a value of 8 selects UPC/EAN. You can select all three formats in a single command by setting the <code> value to $1+4+8 = 13$, as follows:

<Esc>-y13F

Issuing multiple commands

Several individual escape sequence commands may be concatenated together and be transmitted to the bar code interface as a single escape sequence. To do this, append one or more additional <code><command> character sequences to the end of a standard, single, escape sequence.

When issuing multiple commands, all of the intermediate <command> ASCII characters must be lower case letters, and the terminating <command> character must be an upper case ASCII character.

As with a single command, the escape sequence must not contain any embedded spaces.

For example, following three commands:

<Esc>-y13F Select Codabar, UPC/EAN and Code 39 symbologies

<Esc>-y2H Do not transmit Codabar start and stop characters

<Esc>-y1D Insert a 10 ms delay between each data character

may be concatenated into a single escape sequence:

<Esc>-y13f2h1D

The order of issuing commands is usually not important, so that the following three escape sequences are all functionally identical:

<Esc>-y13f2h1D

<Esc>-y2h1d13F

<Esc>-y1d13f2H

Exceptions to this rule are mentioned explicitly in the following descriptions of the individual commands.

Serial intercharacter delay**<Esc>-y<code>D**

Enable or disable a ten millisecond delay between the transmission of each data character in the bar code data string.

<code>	Option
0	No delay
1	Turn 10 millisecond delay on

The default value is equivalent to the command <Esc>-y0D.

Hard reset**<Esc>E**

Perform a hard reset and run a self-test.

A hard reset will take approximately one second to complete. During this time, the bar code reader interface will not react to any further commands that may be written to it.

All previous escape sequence commands sent to the bar code interface will be aborted when a hard reset is executed. After the reset has taken place, all of the interface options will revert to their default values.

If the interface has failed its self-test, it will immediately transmit one of the following four messages:

```
ROM SELF TEST FAILED<CR><LF>
PROCESSOR SELF TEST FAILED<CR><LF>
LOWER RAM SELF TEST FAILED<CR><LF>
UPPER RAM SELF TEST FAILED<CR><LF>
```

No message will be issued if the interface has passed the self-test.

Note that the diagnostic messages will always be terminated by a carriage return (0x0d) and line feed (0x0a) character pair, regardless of any previous command to set the termination characters.

All subsequent messages from the interface will be terminated with a (default) single carriage return character.

A *Hard Reset* will not normally be issued as part of a multiple instruction escape sequence, because it will cause all prior commands to be overridden by default settings, and all subsequent commands within the escape sequence to be ignored by the bar code reader interface while it resets itself.

Select bar code symbology**<Esc>-y<code>F**

Set the barcode reader to recognise one or more barcode formats, as indicated in the following table:

<code>	Enable bar code format:
1	Code 39
2	Interleaved 2 of 5
4	UPC/EAN
8	Codabar
16	Code 128

For example, to read Code 39 and UPC/EAN bar codes only, use:

```
<Esc>-y5F
```

If a bar code format has not been enabled, then the bar code reader will ignore all scans in that code format.

The default setting for this option is equivalent to the command <Esc>-y31F.

Note that it may be necessary to set additional interface options to enable the interface to read certain bar code formats.

Check character options**<Esc>-y<code>G**

Enable or disable verification/check digits and check characters within bar code scans. The precise action of this command is dependent upon the bar code format being scanned, as indicated below.

The default state corresponds to the command <Esc>-y0G.

Code 39 and Interleaved 2 of 5

For Code 39 and Interleaved 2 of 5 bar codes, the bar code reader may be programmed to verify, or to ignore check characters in the bar code scan. In addition, the bar code reader may be programmed to transmit, or to omit the verification character from the scan data when it is transmitted to the host computer.

<code>	Option
0	Do not verify check characters
1	Verify Code 39 check characters
2	Verify Interleaved 2 of 5 check characters
8	Transmit Code 39 and Interleaved 2 of 5 check characters

UPC/EAN

The contents of UPC/EAN bar code scans are always checked against the check digit. However, the bar code reader may be programmed to transmit, or to omit the check digit when the scan data is transmitted to the host computer. Note that the check digit in a UPC E bar code is never transmitted to the host computer.

The bar code reader may also be programmed to decode UPC E 0 bar codes, or it may be programmed to automatically discriminate between UPC E 0 and UPC E 1 version bar codes.

<code>	Option
0	Read UPC E 0 only, transmit UPC/EAN check digit
32	Do not transmit UPC/EAN check digit
64	Read both UPC E 0 and UPC E 1

Decoding options**<Esc>-y<code>H**

This command provides several assorted programming options to the programmer. For UPC/EAN bar code scans, the bar code reader may be programmed to accept (zero, two or five) supplemental digits in the code, and to expand UPC E bar codes automatically into a UPC A code format, if required.

The bar code reader may also be programmed using this command to decode Code 39 bar codes in standard, or in extended mode. If the extended option is selected, then the bar code reader will encode each character pair into the corresponding ASCII characters; if standard mode is selected, then each character in the scan data is decoded individually.

The bar code reader may also be programmed to retain (or discard) Codabar start and stop characters in the scan data it transmits to the host computer.

<code>	Option
1	Extended Code 39
0	Standard Code 39
2	Do not transmit Codabar start and stop characters
0	Transmit Codabar start and stop characters
4	Read UPC bar codes only
0	Read both UPC and EAN bar codes
8	Decode UPC/EAN 2 digit supplement data
0	Do not decode UPC/EAN 2 digit supplement data
16	Decode UPC/EAN 5 digit supplement data
0	Do not decode UPC/EAN 5 digit supplement data
32	Expand UPC E bar codes into UPC A bar codes

0	Do not expand UPC E bar codes
64	Auto discriminate UPC/EAN supplementals
0	Require UPC/EAN supplementals
128	Transmit UPC/EAN ID characters
0	Do not transmit UPC/EAN ID characters

The default setting is equivalent to the command <Esc>-y0H.

Notes

When the option is set to decode UPC/EAN 2 digit (or 5 digit) supplement data, then the bar code label **must be scanned in the forwards direction only**.

Only UPC/EAN bar codes containing supplement digits may be read by the interface if either, or both of these options has been enabled. Both the 2 digit and 5 digit options may be enabled together to allow both types of bar code scans to be read by the interface, if required.

If the supplement digit options are not enabled by this command, then bar code labels containing supplement digits may still be scanned, and they may be scanned in both directions (i.e. forwards or backwards). However the supplement digit data is **not** transmitted to the host computer.

Single read mode <Esc>-y<code>J

This command will force the bar code reader to abort any current scan and not transmit that scan data to the HC.

Enabling *Single read mode* means that the bar code reader will read a single bar code label each time a *Single read control* command is issued.

When *Single read mode* is disabled, the bar code reader will attempt to read a bar code label whenever a label is scanned.

<code>	Option
1	Single read mode is enabled
0	Single read mode is disabled

The default value is equivalent to the command <Esc>-y0J.

Single read control <Esc>-y<code>K

This command will force the bar code reader to abort any current scan and not transmit that scan data to the HC.

When *Single read mode* has been enabled (see above), the *Single read control* may be used to enable the bar code reader, so that it then will read one scan.

<code>	Option
1	Read next scan

Set Interleaved 2 of 5 length <Esc>-y<code>M

This command presets the length of Interleaved 2 of 5 bar code read by the bar code reader.

There are three length checking options available:

<code>	Option
0	The bar code may have a variable length, between 4 and 32 digits
1..32	The bar code is <code> digits long (even values only)
33	The bar code may only contain either 6 or 14 digits

The default setting is equivalent to the command <Esc>-y0M.

Notes

Although the Interleaved 2 of 5 bar code length may be set to the minimum setting (i.e. two digits), short Interleaved 2 of 5 bar codes may well appear in other, longer bar codes. As a result, false readings may be given when other formats of bar code label are scanned, which may be misinterpreted as two digit Interleaved 2 of 5 labels. Consequently, the minimum setting of 2 digits in this option is not recommended.

An Interleaved 2 of 5 bar label must contain an even number of digits. If `<code>` is set to an odd number between one and thirty one, then the micro controller will automatically round up the value to the next higher even number.

Set termination string `<Esc>-y<code>O<string>`

This command will force the bar code reader to abort any current scan and not transmit that scan data to the HC.

Append the termination string `<string>` to the end of every bar code data message transmitted to the HC computer by the micro controller, to mark the end of that message.

The default termination string is a single carriage return character (0x0d). However, a user-defined termination string may be defined with this command, which will be used to terminate all subsequent message transmitted by the bar code reader interface. An example escape sequence, which substitutes the characters "stop" as the terminating string in place of the default value, is:

```
<Esc>-y40stop
```

A termination string may contain a maximum of **four** characters and a minimum of **zero** characters. The total number of characters contained within the termination string must be passed as the `<code>` argument of the escape sequence.

The default termination string transmitted by the bar code reader is a single carriage return character (0x0d).

Unlike all other sequence commands described in this document, the ASCII characters for the termination string follow the command identifier character that normally terminates a command. As a result this command must be positioned last, if issued within an escape sequence that contains more than one command (even if the terminating string contains zero characters).

For example, consider the commands:

```
<Esc>-y40stop          Set the termination string to "stop"
```

```
<Esc>-y1D              Insert a 10 ms delay between each data character
```

If these commands are combined in a single escape sequence, that sequence must be:

```
<Esc>-y1d40stop
```

Code ID characters `<Esc>-y<code>Q`

Instruct the bar code reader to add a lower case ID character to the data message transmitted to the HC, to identify the symbology of the bar code. This ID code character is transmitted before the bar label data.

The following values will enable, and disable this option:

<code><code></code>	Option
1	Transmit ID characters
0	Do not transmit ID characters

The table below displays which lower case ASCII character ('a' - 'e') is used to identify each type of bar code label:

ID character	Bar code format
a	Code 39
b	Interleaved 2 of 5
c	UPC/EAN/JAN
d	Codabar
e	Code 128

The default state is equivalent to the command `<Esc>-y0Q`.

Status request**<Esc>-y<code>S**

This command will force the bar code reader to abort any current scan and not transmit that scan data to the HC.

Return a data string to the host computer, containing information about the status of the firmware inside the bar code reader.

<code>	Option
1	Transmit the status message

The status message that is transmitted back to the HC in response to this instruction is:

HBCR-161X Version 15.x

followed by the current termination string, where "15.x" represents the present revision level of the firmware inside the bar code reader interface.

Scanner enable**<Esc>-y<code>W**

This command will force the bar code reader to abort any current scan and not transmit that scan data to the HC.

Enable or disable the bar code reader.

When disabled, the bar code scanner will not read data from any bar code labels.

<code>	Option
1	Enable the bar code scanner
0	Disable the bar code scanner

The default is for the bar code scanner to be enabled.

RS232 port/bar code driver services

The RS232 port/bar code device driver supports all the services that are described in the *Serial Port* chapter of this manual. This section describes only those services that provide modified or additional behaviour when used with the bar code reader interface. These include services to open the port, write data to the bar code interface and read data from a bar code wand connected to the port.

p_open(TTY:)**Open the device**

```
INT p_open(VOID **ppcb, "TTY:B", -1);
```

Open a channel to the RS232 port/bar code device.

The table below displays which device should be accessed to open the RS232 port or the bar code reader interface, when the expansion module has been inserted into either of the top and bottom slots of the HC.

Slot	RS232 port	Bar code port
Top	TTY:A	TTY:C
Bottom	TTY:B	TTY:D

On opening a channel, RS232 port, the bar code reader interface and any connected wand or scanner will all be powered up automatically.

Returns zero if the device was opened successfully, otherwise a negative error.

For example, the following code fragment illustrates the opening of a channel to the bar code interface with the device in the top slot of the HC.

```
LOCAL_D VOID *pHandle;
INT result;

result=p_open(&pHandle, "TTY:D", -1);
```

p_close**Close the channel**

```
INT p_close(VOID *pcb)
```

Close the channel to the RS232 port/bar code device.

Returns zero.

Closing the channel will also cause the RS232 port/bar code interface to be powered down. To save battery power, a device should be closed as soon as it is no longer required.

Note that when the device is closed, or when the HC automatically powers down, all of the information programmed into the bar code interface will be lost. If you do not wish to use the default settings of the bar code reader interface, then you will need to reprogram your custom settings back into the interface when the device is next opened.

P_FSENSE**Sense serial port characteristics**

```
INT p_iow(VOID *pcb, P_FSENSE, P_SRCHAR *pserial);
```

Sense the serial port characteristics, writing them to the P_SRCHAR struct (defined in *p_serial.h*) pointed to by pserial.

The P_FSENSE service cannot fail and returns zero.

If using the bar code interface, the serial port characteristics must be altered once a channel is opened, to allow the interface to communicate successfully with the HC. See the description of the P_FSET service for further details.

P_FSET**Set serial port characteristics**

```
INT p_iow(VOID *pcb, P_FSET, P_SRCHAR *psrchar);
```

Set the serial port characteristics from the P_SRCHAR struct (defined in *p_serial.h*) pointed to by psrchar.

Returns zero if the P_FSET service completed successfully, otherwise returns a negative error.

If using the bar code interface, the serial port characteristics must be altered once a channel is opened, to allow the interface to communicate successfully with the HC. Since the bar code interface does not use RTS/CTS handshaking, the CTS line in the serial port must be disabled.

The following code fragment illustrates the use of the P_FSENSE and P_FSET services to disable RTS/CTS handshaking:

```
#include <p_serial.h>

LOCAL_D P_SRCHAR srChar;
LOCAL_D VOID *pHandle;

f_leave(p_open(&pHandle, "TTY:D", -1));
p_iow(pHandle, P_FSENSE, &srChar); /* read default settings into srChar */
srChar.hand=P_IGN_CTS;           /* suspend RTS/CTS handshaking */
p_iow(pHandle, P_FSET, &srChar);  /* set up port with new settings */
```

Note that this code assumes that it is called under the protection of p_enter.

P_FREAD**Read from device**

```
VOID p_iow (VOID *pcb, INT P_FREAD, VOID *buf, UINT *plen);
VOID p_ioc (VOID *pcb, INT P_FREAD, WORD *pfstat, VOID *buf, UINT *plen);
```

Read up to *plen bytes of data from the device into the buffer pointed to by buf. The buffer is assumed to be of sufficient length to receive the data. If using an asynchronous P_FREAD service, it is the caller's responsibility to preserve the data space pointed to by buf and plen until the service completes.

The result is returned by a call to p_iow and written to *pfstat by the asynchronous call. The result is zero if the service completed successfully, otherwise it is a negative error.

A read operation may take an indefinite time to complete and it should therefore be executed asynchronously within a quality system. For clarity, the following example code uses synchronous reads.

```
#define MAX_BARCODE 128
#define TERMINATE_CHAR 0x0d

VOID *pHandle;
TEXT *p;
INT len;
UINT count;
TEXT buf[MAX_BARCODE+2];

len=0;
count=1;
p=&buf[0];
FOREVER
{
    p_iow(pHandle,P_FREAD,p,&count);
    len+=1;
    if ((*p++==TERMINATE_CHAR)|| (len>=MAX_BARCODE_LEN))
    {
        *--p=0; /* convert to a zero-terminated string */
        len-=1;
        break;
    }
}
p_printf("Bar code is %s, containing %d bytes",&buf[0],len);
```

P_FWRITE

Write to device

```
INT p_iow(VOID *pcb,INT P_FWRITE,VOID *buf,UINT *plen);
VOID p_ioc(VOID *pcb,INT P_FWRITE,WORD *pfstat,VOID *buf,UINT *plen);
```

Write up to *plen bytes of data from the buffer pointed to by buf to the device. If using an asynchronous P_FWRITE service, it is the caller's responsibility to preserve the data space pointed to by buf and plen until the service completes.

The result is returned by a call to p_iow and written to *pfstat by the asynchronous call. The result is zero if the service completed successfully, otherwise it is a negative error.

A write operation may take an indefinite time to complete and should therefore be executed asynchronously within a quality system. For clarity, the following example uses a synchronous write.

The following code fragment transmits the escape sequence <Esc>-y1d24f2H to the bar code reader interface:

```
#define MAX_COMMAND 64

VOID *pHandle;
UINT len;
UBYTE command[MAX_COMMAND+2];

p_scpy(&command[0],"\033-y1d24f2H");
len=p_slen(&command[0]);
p_iow(pHandle,&command[0],&len);
```

An example program

The program listed below will operate a bar code interface connected to either the top, or the bottom of the HC. The program will read scanned bar code data from the reader interface, display that information on the screen and then await a key press. The scanning operation may be repeated as many times as required. Press the X key after a scan to exit the program; pressing the V key will display the firmware status of the software inside the bar code interface.

This program is only given here as a simple example of how to program the micro controller and read scan data: a quality system should carry out more rigorous testing for errors, and should also use asynchronous operations to read and write the data.

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

#define MAX_STRING 128

LOCAL_D VOID *pHandle;

GLDEF_C INT main(void)
{
    P_SRCHAR srChar;
    INT len, onebyte, result, reply;
    TEXT buffer[MAX_STRING+2];
    /* Try to open a port */
    if (p_open(&pHandle, "TTY:D", -1) < 0)
    {
        if ((result = p_open(&pHandle, "TTY:E", -1)) < 0)
            p_panic(result);
    }
    /* Set up serial port characteristics */
    p_iow(pHandle, P_FSENSE, &srChar);
    srChar.hand = P_IGN_CTS;
    p_iow(pHandle, P_FSET, &srChar);
    buffer[0] = '\0';
    onebyte = 1;
    do
    {
        /* Display menu on screen */
        p_printf("      BAR CODE SCANNER");
        p_printf("%s", buffer);
        p_printf("\npress V for Version,");
        p_printf("      X to exit");
        p_printf("      or any other key");
        p_printf("      to scan a label");
        reply = p_tofold(p_getch());

        if (reply == 'V')
            p_write(pHandle, "\033-y1S", 5);
        if (reply != 'X')
        { /* Read input from the bar code interface */
            len = 0;
            do
            {
                p_iow(pHandle, P_FREAD, &buffer[len], &onebyte);
            } while (buffer[len++] != 13 && len < MAX_STRING);
            len--;
            buffer[len] = 0;
            p_sound(5, 300);
        }
    }
```

```
    if (reply!='X')
    {
        if (reply!='V')
            p_printf("CODE %s",buffer);
        else
        {
            p_printf("    Version:\n    %s",buffer);
            p_printf("Press key to continue");
            p_getch();
        }
    }
} while (reply !='X');
p_close(pHandle);
return(0);
}
```


CHAPTER 15

INTRODUCTION TO PSION

INFRARED COMMUNICATIONS

Note: This document is a beta version and may be subject to change.

About this chapter

The use of infrared (IR) as a medium allows devices to communicate without using a connecting cable. The Psion Series 3c (S3c) and Psion Siena both run an Infrared Communications (“Beaming”) application. This application sits on top of a communications protocol stack that implements the standards set by the Infrared Data Association (IrDA), (see <http://www.irda.org/irda/irdabout.html>). These standards specify both the physical and the link layer. This chapter describes the application programming interface to the Psion implementation of the IrDA standards, and gives details of the standards only where relevant. For a full understanding of infrared communications you should read the IrDA standards documents, copies of which are downloadable from the IrDA website, (<http://www.irda.org/irda/standrd.html>). The three relevant standards employed by Psion are tabulated below:

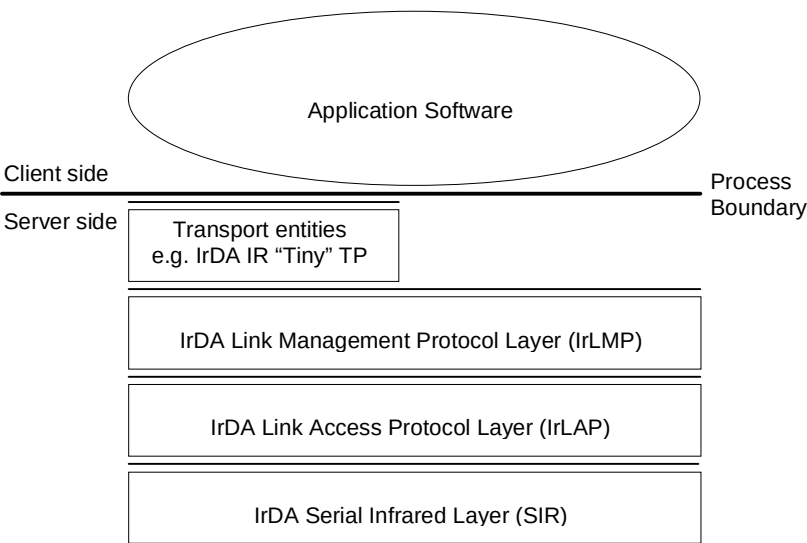
IrDA standard

Serial Infrared Link Management Protocol (IrLMP)
Serial Infrared Link Access Protocol (IrLAP)
Serial Infrared Physical Layer Link Specification (SIR)

Communications layer

Network layer (OSI layer 3)
Data link layer (OSI layer 2)
Physical layer (OSI layer 1)

The IrDA protocol layer model



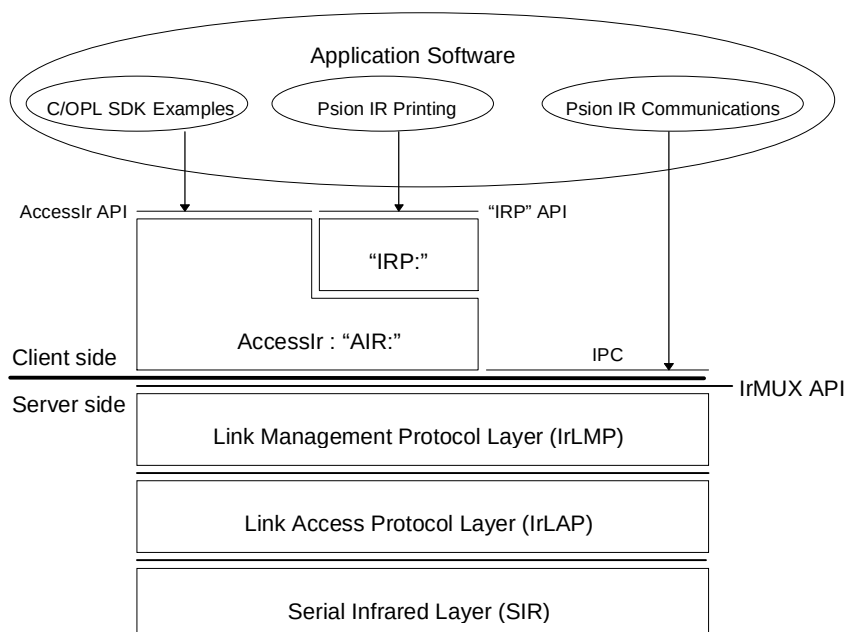
The IrDA protocol stack

Introduction to Psion infrared communications

Important note:

The Psion S3a and Psion Siena IR ports are designed only for IrDA compliant communication. They are not suitable for communication with machines that use other standards.

The Psion protocol layer model



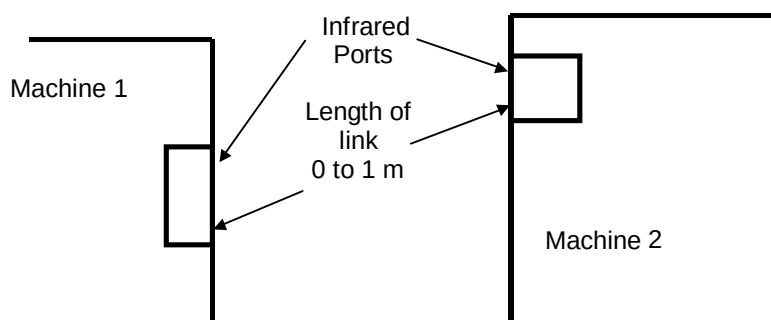
The Psion infrared protocol stack

The physical layer

The Serial Infrared Device Driver (SIR) provides services to the layer above it (IrLAP), and directly controls the Psion computer's hardware.

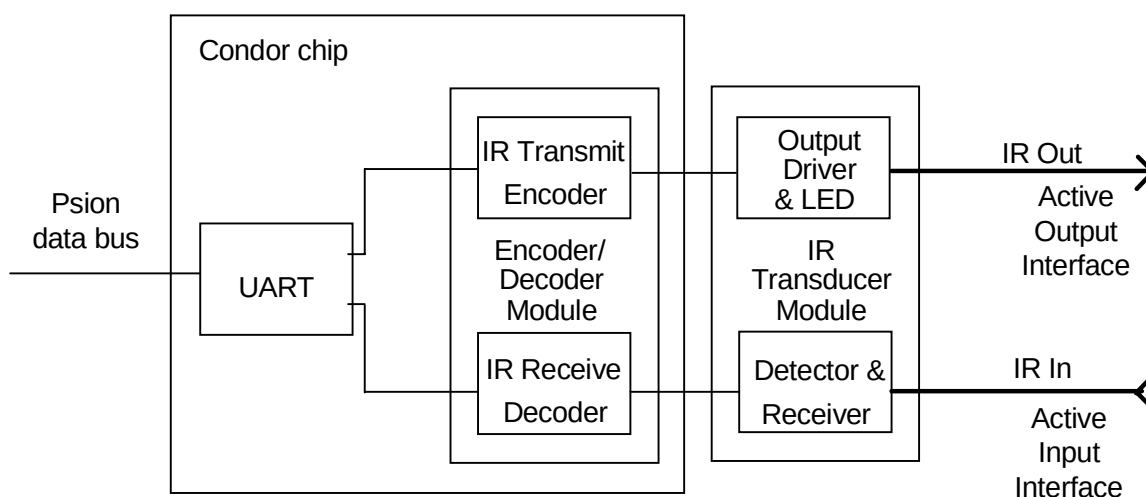
Port geometry

The Psion serial infrared software supports a link between two machines from zero to at least one metre apart as shown below. These machines can be two Psions (S3c or Siena) or a Psion and another device such as a printer with an IrDA compliant interface. The ports on the two machines do not need be aligned perfectly.



The port geometry for two machines using IrDA infrared communications

Internally the Psion IR port is configured as below:



The internal organization of the Psion serial infrared communications port

The Condor chip contains a UART which handles both standard wire based RS232 communications and IR communications. The device is switched between these two modes and cannot handle both at the same time.

Data transfer rate

The Psion serial infrared communications port operates at the standard IrDA speed of 9600 Baud. It can also work at other speeds on the S3c and Siena, (these are optional to the IrDA standard):

Speed (Baud)	S3c	Siena
2,400	Yes	Yes
9,600	Yes	Yes
19,200	Yes	Yes
38,400	Yes	No
57,600	Yes	No
115,200	Yes	No

Data link layer

The IrDA data link layer protocol (IrLAP) is based on the standard asynchronous HDLC and SDLC half duplex protocols.

It has the following extra features, (mostly to handle the mobile, ad-hoc nature of the medium):

- Extended addressing
- Dynamic address conflict resolution
- Extended error recovery mechanisms
- Dynamic station discovery/identification
- Extended connection setup, (so that stations can negotiate to establish the best connection speed that both connecting machines can support)
- Any station can contend to become a primary station
- Extended rules for medium access, (to resolve contention between stations competing for control of the medium).

IrLAP uses services provided by the physical layer (SIR), and provides services to the layer above it (IrLMP).

Primary and secondary stations

At any given instant there may be at most one IrLAP connection between a given pair of stations (IrDA compliant machines). One of the machines must assume the role of IrLAP primary station while the other must assume the role of IrLAP secondary station.

An IrDA primary station is a machine that takes responsibility for data flow control and for unrecoverable data link errors.

An IrDA secondary station is any station that has an IrLAP connection active and that has not assumed the role an IrLAP primary station.

IrLAP services

IrLAP provides two general types of services:

- Connectionless services
- Connection-oriented services

A connectionless service involves either the general broadcast of a message from a station, and/or the general reception of a message by a station, without the necessity of identifying the other station (if any) involved. Connectionless messages (datagrams) are sent with the expectation that they will be received correctly; there is no acknowledgement of correct receipt. If a message is received incorrectly, a higher layer must ask for re-transmission.

A connection-oriented service requires that both stations involved in the message transfer have knowledge of the other's characteristics. A logical session is set up between the primary and secondary station, and there is acknowledgement that messages are correct.

The network layer

The IrDA Link Management Protocol (IrLMP) uses services provided by the data-link layer beneath it (IrLAP) and provides services to clients above (i.e., transport entities and/or applications).

IrLMP consists of two parts, both of which are accessible by the Psion IrMUX API using IPC:

- LM-IAS (Link Management Information Access Service)
- LM-MUX (Link Management Multiplexer)

AccessIr is an API added by Psion to simplify IR communications - it is not part of the IrDA protocol standard - and sits above IrLMP on the client side. It uses the AIR: device driver.

An application developer can choose to use either the API to AccessIr (recommended) or the API to IrMUX. Whichever one is used the sequence of operations required to set up, use and close down an IR link is similar.

For a connection-oriented service the link management process involves several stages:

- IAS application logging (secondary station)
- discovery (primary station)
- multiplexing the link (primary station)
- controlling the link (primary station)
- closing down the link (primary station)

IAS application logging

The IAS works by each IrDA device keeping a database of the services (IR enabled applications) that the device currently has available. Applications must register with the IAS server on their station so that they are added to this database. Software on one machine can then find out the services available on another machine by querying the database of services on that machine. Applications can also remove themselves from the IAS database.

Discovery

Discovery is the process whereby the IrDA primary station sets up a database of the secondary stations that are currently within IR communication range. This database is distinct from the IAS database.

Each record in the discovery log is a `DISCOVERY_LOG` struct:

```
typedef struct
{
    UINT solicited;
    UINT sniff;
    ULONG devAddr;
    UINT lapVersion;
    UBYTE info[32];
} DISCOVERY_LOG;
```

The `DISCOVERY_LOG` fields are:

<code>solicited</code>	If TRUE, indicates that the discovered information was solicited by an explicit discovery request by the host station. If FALSE, indicates that the discovered information was unsolicited (i.e. responding to a previously received discovery XID CMD frame).
<code>sniff</code>	If TRUE, indicates that the remote (discovered) device is a sniffing device.
<code>devAddr</code>	The remote (discovered) station's 32-bit device address.
<code>lapVersion</code>	The 16-bit IrDA version number of the remote (discovered) IrLap layer. Should hold 0x00 indicating that the remote machine conforms to the IrDA IrLAP 1.00 specification
<code>info</code>	32 bytes of user-defined data passed from the remote (discovered) station to the host station. This discovery information is used by peer MUX layers for identification purposes. At the current default baud of 9600, only 23 bytes of the discovery info field can actually be used. For a Psion machine <code>info</code> will contain the non-zero-terminated string "Psion Series3" or "Psion Siena" for an S3c or Siena respectively.

The `info` field also contains the following bytes:

byte	Character set	Indicates the character set supported by the remote machine; 0x00 here indicates ASCII
2 bytes	Service hint	Give an indication as to the level of IrLMP support provided by the discovered device.

The service hint bitfields are as follows:

BYTE1:		BYTE2:	
Bit	Function	Bit	Function
0	PnP Compatible	0	Telephony
1	PDA/Palmtop	1	File Server
2	Computer	2	IrCOMM
3	Printer	3	reserved
4	Modem	4	reserved
5	Fax	5	reserved
6	LAN Access	6	reserved
7	Extension	7	Extension

Full details can be found in the IrDA IrLAP specification.

Multiplexing

Multiplexing the link allows independent applications (services) to exchange data using a single IrLAP link. Software on one machine can discover the services available on another machine by querying the database of services on that machine.

Link Control

Managing the use of the multiplexed link is called Link Control. This includes allowing certain client applications to take exclusive control of the IrLAP link connection, (*Exclusive mode*).

The Psion IR Communications application

The Psion IR Communications application (sometimes called IR Beaming) uses the Psion Link Paste mechanism when sending data, (see the *Link Paste* chapter in the *OLIB Reference* manual). The IR Communications application takes whatever data was highlighted in the most recent application to declare itself to be the Link Paste Server (or the selected file in a System Screen file list or directory), and uses that as the data/file to send. Receiving data over IR works independently of the Link Paste mechanism.

Third party applications

Any existing application that supports the Link Paste mechanism will work with the Psion IR Communications application to send data. In general this means that it must be an HWIM program, because it uses the ATS mechanism. To modify an HWIM program so that it can use IR Receive:

- add the Bring keycode (21) to the accelerator list
- map the Bring keycode to the command manager method used to implement “bring” functionality

It is much more involved to add IR Receive capability to non-HWIM applications, such as HWIF ones. They could be made to explicitly look for the IPC messages relating to the Bring keycode, but this is not so easy.

System resources

A `SYS_IR_POWER_LEVEL` resource has been added, giving the text “High” and “Low” for Infrared power levels.

A `SYS_PRINTER_IR` resource has been added, giving the text “Infrared” as a printer type.

Keypresses

Normally, IR-related keys are captured by the built-in IR data transfer application. If the user disables IR, by selecting another mode of communication, these keypresses are absorbed by the `ws_process_key` method of any application’s instance of `WSERV`. As a general service to the user, the method also displays an “Infrared not enabled” information message.

The `ws_process_key` method of the `WSERV` class performs the following additional action on the Series 3c:

- if the keypress is `W_KEY_IR_LINK`, 0x142 (which would normally be captured by the infrared file/record transfer software) display an information message to the effect that Infrared is disabled

The `ws_process_key` method of the `WSERV` class performs the following additional action on the Siena:

- if the keypress is one of `W_KEY_IR_BRING`, 0x140 or `W_KEY_IR_SEND`, 0x141 (which would normally be captured by the infrared file/record transfer software) display an information message to the effect that Infrared is disabled.

The IR printer port device driver

Psion Series 3c machines are capable of point and shoot printing using IR, to IrDA compliant printers. Infrared enabled printers on the market at the time of writing include:

- HPLaserJet 5P
- HP DeskJet 340
- Citizen PN60

The Psion IR printer port device driver `IRP:` is an example of a device driver that sits logically above `AIR:` on the client side. It works in exactly the same way as the normal `PAR:` parallel port device driver. The `IRP:` device driver automatically starts the IR protocol stack.

Siena models do not have the driver in ROM. The driver `IRLPT.LDD` is installed in `\SIBOSDK\LIB` as part of the LDD component of the optional disk, to be loaded onto a Siena if required.

CHAPTER 16

THE ACCESSIR API

Note: This document is a beta version and may be subject to change.

Using the AccessIr API

Prerequisites

It is essential that the *Introduction to Psion Infrared Communications* chapter has been read before this chapter.

Introduction to using the AccessIr API

The AccessIr API facilitates Infrared communication between two Psion machines (Series 3c and/or Siena), using the AIR: device driver, *accessir.ldr*. This API is much easier to use than the IrMUX API, but gives less flexibility. It is therefore recommended that the AccessIr API is used for all simple IR communication applications. It is not present in the Siena ROM, but is installed in the *\SIBOSDK\LIB* directory as part of the LDD component of the SIBO C SDK Optional disk, to be loaded onto a Siena if required.

The AIR: device driver is not present in the ROM of Siena machines, and is 1.7KB in size. It must be loaded first (after transfer onto the machine's internal drive) by using:

```
p_loadldr("ACCESSIR.LDR")
```

The AIR: device driver must be unloaded at the end on the Siena using:

```
p_devdel("ACCESSIR", E_LDR)
```

Otherwise there are no differences between using the AccessIr API on the Siena and on the Series 3c.

Only a connection-oriented option is available.

AccessIr requires the IR protocol stack to be started first (as it must be when using the IrMUX API).

There are then seven steps in the communication process for the primary station:

1. Open a channel to the AccessIr device driver
2. Discover and log the remote machines
3. Select a remote machine
4. Connect to the selected remote machine
5. Send or accept data
6. Disconnect
7. Close the channel to its IR device

There are five steps in the process for the secondary station:

1. Open a channel to the AccessIr device driver
2. Wait for the primary station to connect
3. Send or accept data
4. Optional Disconnect
5. Close the channel to the AccessIr device driver

In all the code examples and function call prototypes below, either `p_iow()`, `p_ioa()` or `p_ioc()` may be used where `p_iow()` is given. The constants used in the AccessIr API are defined in the `p_file.h` PLIB include file.

Initialising the IR protocol stack

Before the IR protocol stack can be used it must be initialized, by doing a `p_execc`. The protocol stack is started as a separate process.

The argument `*name` is `SYSSIRDA.IMG` in the example code segment below:

```
LOCAL_C HANDLE loadIRDAserver(TEXT *name)
{
    INT err;
    TEXT serverPath[P_FNAME_SIZE];
    HANDLE server_pid=NULL;
    /* Start the IR process & store its server_pid in property */
    if (E_FILE_NXIST==(server_pid=p_pidfind(name)))
    { /* Load from M, then from wherever we are... */
        f_fparse(name, "LOC::M:\\", &serverPath[0], NULL);
        if ((server_pid=p_execc(&serverPath[0],
                               (UBYTE*)&server_pid,
                               sizeof(server_pid)))<0)
        {
            f_fparse(name, "ROM::\\", &serverPath[0], NULL);
            if ((server_pid=p_execc(&serverPath[0],
                                   (UBYTE*)&server_pid,
                                   sizeof(server_pid)))<0)
            { /* Error...no irda! */
                return(0);
            }
        }
        if (0!=(err=p_presume(server_pid)))
            return(0);
    }
    p_sleep(5L); /* Give IRDA a chance to get going! */
    return (server_pid);
}
```

Opening and closing a channel to the IR device

Before a connection can be made, a channel must first be opened to the IR device using `p_open("AIR:")`. This channel must be closed when IR operations have ceased, using `p_close()`.

Example

```
if (!p_open(&pIR, "AIR:", 0))
{
    ...
    p_close(pIR);
}
```

Discovery

Secondly, a log must be obtained of all remote machines that it is possible to communicate with, using `p_iow(P_FIRDISCOVER)`.

Selection

The machine to connect to is then selected from the log, using `p_iow(P_FIRSELECT)`.

Connection

Connection is then made by the primary station to the selected secondary station, using `p_iow(P_FIRMAKECONNECT)`. The secondary station sends a zero terminated string of up to 56 bytes of connect data to the primary station, when the connection is accepted.

The remote machine must be awaiting connection; it is put in this state by using `p_iow(P_FIRAWAITCONNECT)`. The primary station then sends a zero terminated string of up to 56 bytes of connect data to the secondary station when connection is successful.

Sending and accepting data

The primary station may then either send data, using `p_iow(P_FWRITE)`, or accept data from the secondary station using `p_iow(P_FREAD)`.

Likewise the secondary station may either accept data from the primary station, using `p_iow(P_FREAD)`, or send data using `p_iow(P_FWRITE)`.

Disconnection

Once data transfer is complete the connection between the two stations must be discontinued by the primary station or secondary station, using `p_iow(P_FIRDISCONNECT)`.

Constants

The constants used in the function calls below have the following numerical equivalents:

<code>P_FREAD</code>	1
<code>P_FWRITE</code>	2
<code>P_FIRDISCONNECT</code>	4
<code>P_FIRAWAITCONNECT</code>	5
<code>P_FIRDISCOVER</code>	6
<code>P_FIRSELECT</code>	7
<code>P_FIRMAKECONNECT</code>	8

Opening and closing a channel to the IR device

`p_open("AIR:")` Open a channel to the IR device

```
INT p_open(VOID *pcb, "AIR:", 0);
```

Open a channel to the IR device, (IrDA Open function).

If successful the port will be powered up.

Arguments

<code>VOID *pcb</code>	Handle
<code>"AIR:"</code>	Device name
<code>0</code>	Mode

Return value

`INT` Zero if the channel was opened successfully; else a negative error code

Possible errors include:

<code>E_GEN_INUSE</code>	Port in use by another client
<code>E_GEN_NOMEMORY</code>	Insufficient memory to make connection

Example

```
if (!p_open(&pIR, "AIR:", 0))
{
    ...
    p_close(pIR);
}
```

p_close **Close a channel to the IR device**

```
INT p_close(VOID *pcb);
```

Close an open channel to the IR device, (IrDA Close function).

The internal receive data buffer is flushed. If a frame of data is currently being transmitted then transmission is completed before closure. The transmit data buffer becomes empty. Any outstanding IR driver read or write requests are cancelled.

Arguments

VOID *pcb	Handle
-----------	--------

Return value

INT	Zero
-----	------

Example

```
p_close(pcb);
```

Discovery, selection and connection**p_iow(P_FIRDISCOVER)** **Discover IR enabled machines**

```
INT p_iow(VOID *pcb, P_FIRDISCOVER, VOID *buf, UWORD *pNslots);
```

Obtain the log of remote machines, (IrDA Discover function).

Arguments

VOID *pcb	Handle
P_FIRDISCOVER	IR service
VOID *buf	Buffer to receive discovery list; on return contains the list of machines found
UWORD *pNslots	Pointer to word containing the number of slots; on return contains the number of machines found

Return value

INT	Maximum number of machines to find
-----	------------------------------------

Example

```
p_iow(pcb, P_FIRDISCOVER, Buffer, pNslots);
```

p_iow(P_FIRSELECT) **Select remote machine to connect to**

```
INT p_iow(VOID *pcb, P_FIRSELECT, VOID *Address);
```

Select a machine to connect to, (IrDA Select function).

Arguments

VOID *pcb	Handle
P_FIRSELECT	IR service
VOID *Address	Pointer to long containing machine address

Return value

INT Zero

Example

```
p_iow(pcb,P_FIRSELECT,&Address);
```

p_iow(P_FIRMAKECONNECT) Connect to selected machine

```
INT p_iow(VOID *pcb,P_FIRMAKECONNECT,TEXT *pname,VOID *buf[0]);
```

Connect to the selected remote machine, (IrDA Make Connect function), where pname points to a string containing the name of the application to connect to.

Arguments

VOID *pcb	Handle
P_FIRMAKECONNECT	IR service
TEXT *pname	Name of application to connect to. This is a zero terminated string, maximum 25 characters long (excluding terminator).
VOID *buf[0]	Zero terminated string. On return contains up to sixty bytes of connect data from remote machine, if successful

Return value

INT Maximum data send size or negative error

Example

Either p_iow(), p_ioc() or p_ioa() can be used, e.g.:

```
p_iow(pcb,P_FIRMAKECONNECT,"Bob's Chat program",&Buffer[0]);
IOW(Handle%,8,#UADD(ADDR(Name$),1),#UADD(ADDR(Buffer$),1))
```

Note: contents of Name\$ and Buffer\$ must be zero terminated

p_iow(P_FIRAWAITCONNECT) Wait for remote connect (primary)

```
INT p_iow(VOID *pcb,P_FIRAWAITCONNECT,TEXT *pname,VOID *buf[0]);
```

Wait for a remote machine to connect as the primary station, (IrDA Accept Connection function), where pname points to a string containing the name of the application to connect to on the local machine.

Arguments

VOID *pcb	Handle
P_FIRAWAITCONNECT	IR service
TEXT *pname	Name of your application
VOID *buf[0]	Zero terminated string. On return contains up to 56 bytes of connect data from remote machine, if successful

Return value

INT Maximum data send size or negative error

Example

Either p_iow(), p_ioc() or p_ioa() can be used, e.g.:

```
p_iow(pcb,P_FIRAWAITCONNECT,"Bob's Chat program",&Buffer[0]);
```

Send and accept data

p_iow(P_FREAD) Accept data from selected remote machine

```
INT p_iow(VOID *pcb, P_FREAD, VOID *Data[0], UWORD *Length);
```

Accept data from the remote machine (IrDA Read function).

Arguments

VOID *pcb	Handle
P_FREAD	IR service
VOID *Data[0]	Buffer to receive data
UWORD *Length	Pointer to word to receive data length. On return contains the amount of data received

Return value

INT	Zero or negative error,
-----	-------------------------

Example

Either p_iow(), p_ioc() or p_ioa() can be used, e.g.:

```
p_iow(pcb, P_FREAD, &Data[0], &Length);
```

p_iow(P_FWRITE) Send data to the selected remote machine

```
INT p_iow(VOID *pcb, P_FWRITE, VOID *Data[0], UWORD *Length);
```

Send data to the remote machine (IrDA Write function).

Arguments

VOID *pcb	Handle
P_FWRITE	IR service
VOID *Data[0]	Buffer of data to be sent
UWORD *Length	Pointer to word to containing data length

Return value

INT	Zero or negative error
-----	------------------------

Example

Either p_iow(), p_ioc() or p_ioa() can be used, e.g.:

```
p_iow(pcb, P_FWRITE, &Data[0], &Length);
```

Disconnection

p_iow(P_FIRDISCONNECT) Disconnect from remote machine

```
INT p_iow(VOID *pcb, P_FIRDISCONNECT);
```

Disconnect from the connected remote machine and cancel all requests, (IrDA Disconnect function).

Arguments

VOID *pcb	Handle
P_FIRDISCONNECT	IR service

Return value

INT	Zero or negative error,
-----	-------------------------

Example

```
p_iow(pcb, P_FIRDISCONNECT);  
IOW(Handle%, 4, #0, #0)
```

Example application

The following application allows one Psion machine to act like a sending teletype (constantly queueing IR write calls) and another Psion to act like a receiving teletype (constantly queueing IR read calls).

```

/*  MODULE - ACCESSIR.C
Copyright (C) Psion PLC 1995/6
*/

#include <plib.h>

#ifndef EPOC
    GLREF_D P_DEVICE p_serial,p_file,p_keyb,p_timer;
#endif
LOCAL_D VOID *pIr;
LOCAL_D UBYTE buf[256];
LOCAL_D TEXT connData[64]=" ";
LOCAL_D TEXT printstring[24]="Welcome to IR printing!";
LOCAL_D UWORD slots;
LOCAL_D INT readlen,writelen,ret,i;
LOCAL_D UBYTE devName[24];
LOCAL_D ULONG devAddr;

LOCAL_C HANDLE LoadIrdaServer(TEXT *name)
{
    INT err;
    TEXT serverPath[P_FNAME_SIZE];
    HANDLE serverPid=NULL;

    /* Start the IRDA stack as a separate process and store its pid */
    p_printf("Trying to start %s",name);
    if ((serverPid=p_pidfind(name))==E_FILE_NXIST)
    {
        /* Try to load irda from M first */
        f_parse(name,"LOC:M:\\",&serverPath[0],NULL);
        if ((serverPid=p_execc(&serverPath[0],
            (UBYTE *)&serverPid,sizeof(serverPid)))<0)
        {
            /* Try to load irda from ROM */
            f_parse(name,"ROM:\\",&serverPath[0],NULL);
            if ((serverPid=p_execc(&serverPath[0],
                (UBYTE *)&serverPid,sizeof(serverPid)))<0)
            {
                /* Error ! No irda */
                return(0);
            }
        }
        if ((err=p_presume(serverPid))<0)
            return(0);
    }
    p_sleep(5L); /* Give irda a chance to get going */
    return(serverPid);
}

LOCAL_C VOID PrintMenu()
{
    p_printf("*****");
    p_printf("Press \'p\' to make station primary");
    p_printf("Press \'s\' to make station secondary");
    p_printf("Press \'q\' to quit");
    p_printf("*****");
}

```

```
LOCAL_C INT DiscoverDevices()
{
    p_printf("Doing a discovery");
    slots=6;
    p_iow(pIr,6,&buf[0],&slots); /* P_FIRDISCOVER */
    p_printf("Devices found = %d",slots);
    if (slots>0)
    {
        /* We have discovered at least one device */
        devAddr= *((ULONG *)&buf[4]);
        /* Discovery nickname is last 23 bytes of info */
        p_bcpy(devName,&buf[12],23);
        i=0;
        while (devName[i]>=0x20 && i<23)
            i++;
        devName[i]='\0';
        p_printf("Remote devAddr = 0x%08X",devAddr);
        p_printf("Remote device = %s",devName);
        return(0);
    }
    else
    {
        p_printf("Trying again");
        return(-1);
    }
}

LOCAL_C INT ConnectToFirstDevice()
{
    p_printf("Selecting machine");
    ret=p_iow(pIr,7,&devAddr); /* P_FIRSELECT */
    p_printf("Connecting to machine %08X",devAddr);
    if ((ret=p_iow(pIr,8,"Test",&connData[0]))<0) /* P_FIRMAKECONNECT */
    {
        p_printf("Connect failed with ret = %d",ret);
        p_getch();
        return(0);
    }
    p_printf("Successfully connected machine");
}

LOCAL_C VOID TransmitData()
{
    INT ret;
    p_printf("Writing keypresses to IR port - ESC to cancel");
    writelen=1;
    while ((ret=p_getch())!=27)
    {
        /* OK as long as not escape */
        buf[0]=ret;
        p_iow(pIr,2,&buf[0],&writelen); /* P_FWRITE */
        p_print("%s",&buf[0]);
    }
}

LOCAL_C INT WaitForConnect()
{
    INT ret;
    p_printf("Connecting as a secondary");
    if ((ret=p_iow(pIr,5,"Test",&connData[0]))<0) /* P_FIRAWAITCONNECT */
    {
        p_printf("Wait for connect failed with ret = %d",ret);
        p_getch();
        return(-1);
    }
    p_printf("Accepted a connection");
}
```

```

LOCAL_C INT ReadData()
{
    readlen=1;
    if ((ret=p_iow(pIr,1,&buf[0],&readlen))<0) /* P_FREAD */
    {
        p_printf("Error on reading");
        if (ret==DisconnectErr)
            p_printf("Primary has disconnected");
        return(ret);
    }
    p_print("%s",&buf[0]);
    return(1);
}

LOCAL_C VOID Disconnect()
{
    p_printf("Disconnecting");
    p_iow(pIr,4); /* P_FIRDISCONNECT */
}

GLDEF_C INT main(VOID)
{
    TEXT servername[20]="SYS$IRDA.IMG";

#ifdef EPOC
    p_inst(&p_file,&p_serial,&p_keyb,&p_timer,NULL);
#endif
    p_printf("Psion Software (c) October 1996");
    p_printf("*****");
    p_printf("Starting simple AccessIr beaming app");
    if ((ret=LoadIrdaServer(servername))==NULL)
    {
        p_printf("Problem loading Irda stack");
        p_getch();
        return(0);
    }
    p_printf("Successfully kicked open the irda stack as separate process");
    if ((ret=p_open(&pIr,"AIR:",-1))<0)
    {
        p_printf("Cannot open AccessIr driver ret = %d",ret);
        p_getch();
        return(0);
    }
    p_printf("Successfully opened AccessIr driver");
startLoop:
    PrintMenu();
    ret=p_getch();

    switch (ret)
    {
        case 'p':
        case 'P':
            /* PRIMARY STATION - TRANSMITTER */
            if ((ret=DiscoverDevices())<0)
                goto startLoop;
            ConnectToFirstDevice();
            TransmitData();
            Disconnect();
            break;
    }
}

```

```
case 's':
case 'S':
    /* SECONDARY STATION - RECEIVER */
    if ((ret=WaitForConnect())<0)
        break;
    while ((ret=ReadData())>0)
        /* Keep reading until failure */ ;
    break;
case 'q':
case 'Q':
    p_printf("Terminating program");
    break;
default:
    goto startLoop;
    break;
}
p_close(pIr); /* This also kills the IrDA process */
p_printf("End of program - Hit key to esc");
p_getch();
return(0);
}
```

CHAPTER 17

THE IRMUX API

Using the IrMUX API

Prerequisites

It is essential that the *Introduction to Psion Infrared Communications* chapter has been read before this chapter.

Introduction to using the IrMUX API

The IrMUX API facilitates Infrared communication between two Psion machines (Series 3c and/or Siena).

This section explains how to use each call that can be made to the IrMUX server using IPC on a SIBO (EPOC/16) based computer system. This API is more difficult to use than the AccessIr API, but potentially gives more flexibility. For simple IR communication applications it is recommended that the AccessIr API is used.

With the IrMUX API connectionless or connection-oriented calls are available.

This set of protocols uses machine addresses and port IDs (rather than application names as for the AccessIr API).

The packets of data used must be exactly the right size, there is no error correction and there is no guarantee of transfer.

Initialising the IR protocol stack

Before the IR protocol stack can be used it must be initialized, by doing a p_execc. The protocol stack is started as a separate process. This is done in exactly the same way as when using the AccessIr API. When the last application has logged off, the protocol stack will clean up and terminate itself automatically.

The argument `*name` is `SY$IRDA.IMG` in the example code segment below:

```
LOCAL_C HANDLE loadIRDAserver(TEXT *name)
{
    INT err;
    TEXT serverPath[P_FNAME_SIZE];
    HANDLE server_pid=NULL;
    /* Start the IR process & store its server_pid in property */
    if (E_FILE_NXIST==(server_pid=p_pidfind(name)))
    { /* Load from M, then from wherever we are... */
        f_fparse(name,"LOC::M:\\",&serverPath[0],NULL);
        if ((server_pid=p_execc(&serverPath[0],
                               (UBYTE*)&server_pid,
                               sizeof(server_pid)))<0)
        {
            f_fparse(name,"ROM::\\",&serverPath[0],NULL);
            if ((server_pid=p_execc(&serverPath[0],
                                   (UBYTE*)&server_pid,
                                   sizeof(server_pid)))<0)
            { /* Error...no irda! */
                return(0);
            }
        }
        if (0!=(err=p_presume(server_pid)))
            return(0);
    }
    p_sleep(5L); /* Give IRDA a chance to get going! */
    return (server_pid);
}
```

Logging on to and logging off from the IrMUX server

Before an application can use IrMUX services it must log on. When it has finished it must log off.

Logging on to the IrMUX server

The `LM_Logon` function call is used to log on to the IrMUX server.

Logging off from the IrMUX server

The `LM_Logoff` function call is used to log off from the IrMUX server.

Registering/unregistering applications with the LM-IAS server

This is independent of the process of sending or receiving data. An application can either:

- register itself in the LM-IAS database
- unregister itself from the LM-IAS database

Registering the port with the LM-IAS server

The `LM_RegisterPort` function call is used to register the application with the LM-IAS server.

Unregistering the port with the LM-IAS server

The `LM_UnRegisterPort` function call is used to unregister the application with the LM-IAS server.

LM-IAS services

The `LM_GetValueByClass` message has not been implemented as a direct IPC service, but an IAS Get Value By Class frame must be passed from the LM-IAS server of the primary station to the LM-IAS server of the secondary station. The secondary station replies with another specially formatted frame of data.

The LM-IAS server on a remote machine is reached by starting a connection to the remote machine, (using the `LM_ConnectRequest` function), with a remote port of zero.

The IAS Get Value By Class service can only be used with a connection to a remote LM-IAS server. Direct read and write access to a remote LM-IAS port has not been blocked in order to allow future expansion. A client should not attempt to send frames directly to a remote LM-IAS connection (other than the Get Value By Class message) without a full understanding of the implications of such actions. Developers should study the IrDA IrLMP specification.

The IAS Get Value By Class message frame

The format of the frame that must be sent from the primary station is:

```
byte      0x84
byte      name length
n bytes   application name, not including zero terminator
byte      0x12
byte      "IrDA:IrLMP:LsapSel"
```

The IAS Get Value By Class reply frame

The format of the frame returned by the LM-IAS server on the secondary station is:

```
byte      AND with 0x3F and compare with 4; if not equal to 4, failed
byte      0
byte      non-zero
byte      don't care
byte      don't care
byte      don't care
byte      1
byte      0
word      0
byte      port number
```

Connectionless calls

Data may either be sent or received in a connectionless manner. This process involves one step:

- Queue a connectionless read or write request

The first application to read the transmitted data is the one that gets it.

Reading and writing data

The `LM_CLReadRequest` function call is used to queue a connectionless read request.

The `LM_CLWriteRequest` function call is used to queue a connectionless write request.

Connection-oriented calls

The steps in the process for a secondary station are:

1. Register the application with the secondary station's LM-IAS server
2. Wait for the primary station to connect

The steps in the process for a primary station are:

1. Discover and log the remote machines
2. Connect to the LM-IAS server on the selected remote machine
3. Ask for a particular registered application (by preparing and sending a `LM_GetValueByClass` message)
4. Wait for the reply (the secondary station returns a port ID for the required application)
5. Close the LM-IAS connection

6. Reconnect using the port ID previously returned for the required application on the secondary station
7. Send or accept data (either reliably or unreliably)
8. Disconnect

Discovery

The `LM_DiscoverDevicesRequest` function call is used by the primary station to return information about machines within transmission range.

Connection - first time

The `LM_ConnectRequest` function call is used by the primary station (with a remote port of zero) to attempt to connect to the LM-IAS server of the remote machine.

If successful the primary station sends an IAS Get Value By Class frame to the secondary station. The secondary station sends back a frame containing the ID of the port for the application required (previously registered by the secondary station with its LM-IAS server).

Disconnecting - first time

The `LM_DisconnectRequest` function call is used by the primary station to disconnect the two communicating machines, before subsequent reconnection.

Connection - second time

The `LM_ConnectRequest` function call is used by the primary station to attempt to re-connect to the remote machine, using the port ID for the required application, returned previously.

Reading and writing data

Reading and writing data may be done either reliably or unreliably. If unreliable read/write is used then the data is only sent once.

The `LM_ReadRequest` function call is used to queue a connection-oriented read request.

The `LM_WriteRequest` function call is used to queue a connection-oriented write request.

The `LM_URadRequest` function call is used to queue an unreliable connection-oriented write request.

The `LM_UWriteRequest` function call is used to queue an unreliable connection-oriented write request.

Disconnecting - second time

The `LM_DisconnectRequest` function call is used by the primary station to disconnect the two communicating machines.

Using Exclusive mode

If an application on the primary station wants to be sure of control over the link then it should use Exclusive mode. A call to `LM_AccessModeRequest` gives exclusive access to IrMUX. This means that only that one application can talk, and all other applications are blocked.

If you want to stop other applications from placing IrMUX into Exclusive mode, then the connection (which always starts in Idle mode) can be put into Active mode, by using the `LM_IdleRequest` function call. This should only be done during critical periods of communication. The connection should be put back in Idle mode when the critical period has finished, by using another call to `LM_IdleRequest`.

When in Exclusive mode, if the client application is not responding to incoming data quickly enough, the number of retries to be used on each data frame can be changed (from the default of 1), using the `LM_SetHandshakingLevel` function call. The number of retries must be set back to 1 when IrMUX leaves Exclusive mode.

When exclusive use of IrMUX is no longer required, another call to `LM_AccessModeRequest` should be made.

The current status of the link can be found with a call to `LM_StatusRequest`.

The IrMUX API

IrMUX message format

The MUXMESSAGE struct has the format shown below.

```
typedef struct
{
    E_MESSAGE Mess;           // Message Control
    VOID * MessConn;         // Connection Handle
    VOID * MessArg1;         // Argument 1
    VOID * MessArg2;         // Argument 2
    UINT MessArg3;           // Argument 3
} MUXMESSAGE
```

The E_MESSAGE struct has the format shown below.

```
typedef struct message
{
    struct message next;
    UBYTE *status;
    UINT type;                // Message Number
    HANDLE pid;
} E_MESSAGE;
```

The Message Number specified for each function below should be used as the type in an E_MESSAGE struct, itself used as the mess field of a MUXMESSAGE struct.

Argument 1, Argument 2 and Argument 3 refer to the three MUXMESSAGE fields MessArg1, MessArg1 and MessArg1.

For a full description of how to use these structs with the server function p_mreceive, see the chapter *Processes and Inter-Process Messaging* in the *PLIB Reference* manual, and also the chapter *Inter-Process Communication* in the *OLIB Reference* manual.

Unless specified below the argument values are unchanged when the function call returns.

Errors from IrLAP (returned in the status element of the E_MESSAGE struct) are documented in the IrDA *Serial Infrared Link Access Protocol (IrLAP)* standard document.

LM_Login

Log on to the IrMUX server

Message Number 1

Use this function call to log on to the IrMUX server.

LM_Logoff

Log off from the IrMUX server

Message Number 2

Use this function call to log off from the IrMUX server. This will stop any connections still held open by the client.

LM_RegisterPort

Register a port number with the LM-IAS server

Message Number 10

Use this function call to register an application with the LM-IAS server.

The application name must be less than twenty five characters in length and zero terminated.

Arguments

Connection Handle	Pointer to a <code>VOID*</code> ; on return contains the LM-IAS entry handle
Argument 1	Pointer to a buffer containing the application name
Argument 2	Irrelevant
Argument 3	The port number
Value of status in the <code>E_MESSAGE</code> struct on return:	
<code>OKAY</code>	Register successful
<code>E_GEN_NOMEMORY</code>	No memory to complete request
<code>E_GEN_RANGE</code>	Errors from IrLAP

LM_UnRegisterPort Free registered port with LM-IAS server

Message Number 11

Use this function call to free a registered port number with the LM-IAS server.

Arguments

Connection Handle	Entry handle
Argument 1	Pointer to buffer containing class name
Argument 2	Irrelevant
Argument 3	Irrelevant

LM_CLReadRequest Queue a connectionless read request

Message Number 4

Use this function call to queue a connectionless read request.

It should be noted that in the case of all reads, if a frame is received which is longer than the requested data size the frame will be truncated and data will be lost. If a received frame is smaller than the length requested the actual amount of data received will be placed into the `UINT` pointed to by Argument2.

Arguments

Connection Handle	Irrelevant
Argument 1	Pointer to a buffer to receive the data; on return contains the received data
Argument 2	Pointer to a <code>UINT</code> containing the number of bytes required; on return the <code>UINT</code> contains the actual amount of data returned
Argument 3	Irrelevant
Value of status in the <code>E_MESSAGE</code> struct on return:	
<code>OKAY</code>	Read complete
<code>E_GEN_INUSE</code>	Read already queued
<code>E_MUX_ABORT</code>	IrMUX has been destroyed

LM_CLWriteRequest Queue a connectionless write request

Message Number 5

Use this function call to queue a connectionless write request.

Arguments

Connection Handle	Irrelevant
Argument 1	Pointer to a buffer containing the data to send;
Argument 2	Pointer to a <code>UINT</code> containing the number of bytes to send
Argument 3	Irrelevant
Value of <code>status</code> in the <code>E_MESSAGE</code> struct on return:	
<code>OKAY</code>	Write successful
<code>E_GEN_NOMEMORY</code>	No memory to complete request
(various)	Errors from IrLAP

LM_DiscoverDevicesRequest Return info on in-range machines

Message Number 6

Use this function call to return information about machines within transmission range.

IrMUX will either instruct IrLAP to perform a discovery operation or will return the results of a previous discovery.

Arguments

Connection Handle	Irrelevant
Argument 1	Pointer to a buffer of size at least <code>slots*sizeof(DISCOVERY_LOG)</code> ; on return contains the discovery log
Argument 2	Pointer to a <code>UINT</code> containing the number of slots; on return contains the number of discoveries
Argument 3	Irrelevant
Value of <code>status</code> in the <code>E_MESSAGE</code> struct on return:	
<code>MUX_DISCOVERY_COMPLETE</code>	Discovery complete
<code>MUX_CACHE_USED</code>	Results of previous discovery used
<code>E_GEN_NOMEMORY</code>	Could not complete request
<code>E_MUX_ABORT</code>	IrMux has been destroyed

LM_ConnectRequest Attempt to connect to a remote machine

Message Number 9

Use this function call to attempt to connect to a remote machine. Supplying a home Port number of -1 will cause IrMUX to generate one for you. A remote port number of zero is used to connect to the remote LM-IAS server. See the appropriate IrDA IrLAP document for details of the `CONNECT_STRUCT` and how to set port numbers and other connection parameters.

Arguments

Connection Handle	Pointer to a <code>VOID*</code> to receive the connection handle on return
Argument 1	Pointer to a <code>CONNECT_STRUCT</code> containing connect parameters; on return the <code>CONNECT_STRUCT</code> contains the actual connection parameters
Argument 2	Pointer to a buffer of 60 bytes of connection data; on return the buffer contains connection data from the other machine
Argument 3	Irrelevant
Value of status in the <code>E_MESSAGE</code> struct on return:	
<code>OKAY</code>	Connection complete
<code>E_GEN_NSUP</code>	Unsupported parameters or port number
<code>E_GEN_INUSE</code>	Port number in use by another client
<code>E_MUX_BUSY</code>	A discovery is in process
<code>E_MUX_EXCLUSIVE</code>	Another connection already has exclusive access to IrMUX
<code>E_GEN_NOMEMORY</code>	Insuffient Memory to make connection
<code>E_MUX_REMOTEDISCONNECT</code>	Remote forced disconnection
<code>E_MUX_TIMEOUTDISCONNECT</code>	Forced disconnection, timeout
<code>E_MUX_LOGOFFDISCONNECT</code>	Forced disconnection, client logoff
<code>E_MUX_USERDISCONNECT</code>	User cancelled connection
<code>E_MUX_LAP</code>	IrLAP cannot complete this request

LM_WaitForConnection Wait for remote machine to connect

Message Number 8

Use this function call to get a remote machine to wait for connection through a specified port number.

Arguments

Connection Handle	Pointer to <code>VOID*</code> to receive connection handle on return
Argument 1	Pointer to a <code>CONNECT_STRUCT</code> containing connect parameters; on return the <code>CONNECT_STRUCT</code> contains the actual connection parameters
Argument 2	Pointer to a buffer of 60 bytes of connection data; on return the buffer contains connection data from the other machine
Argument 3	Irrelevant

Value of status in the `E_MESSAGE` struct on return:

<code>OKAY</code>	Connection complete
<code>E_GEN_NSUP</code>	Unsupported parameters or port number
<code>E_GEN_INUSE</code>	Port number in use by another client
<code>E_GEN_NOMEMORY</code>	Insufficient memory to make connection
<code>E_MUX_REMOTEDISCONNECT</code>	Remote forced disconnection
<code>E_MUX_TIMEOUTDISCONNECT</code>	Forced disconnection, timeout
<code>E_MUX_LOGOFFDISCONNECT</code>	Forced disconnection, client logoff
<code>E_MUX_USERDISCONNECT</code>	User cancelled connection

LM_StatusRequest

Return the status of the link

Message Number 21

Use this function call to return the status of the link and whereabouts IrLAP is holding un-ACKed data.

Arguments

Connection Handle	Connection handle
Argument 1	Pointer to an <code>INT</code> to hold the link quality on return
Argument 2	Pointer to an <code>INT</code> to hold the number of un-ACKed frames on return
Argument 3	Irrelevant

Value of status in the `E_MESSAGE` struct on return:

<code>OKAY</code>	Request complete
<code>E_MUX_NOCONNECTION</code>	Connection handle invalid or connection removed by other end

LM_ReadRequest

Queue a read request on a connection

Message Number 24

Use this function call to queue a read request on a connection.

This function will panic the client process if a read is already queued on that connection.

Arguments

Connection Handle	Connection handle
Argument 1	Pointer to a buffer to receive the data on return
Argument 2	Pointer to a <code>UINT</code> containing the number of bytes required; on return the <code>UINT</code> contains the amount of data returned
Argument 3	Irrelevant

Value of status in the E_MESSAGE struct on return:

OKAY	Read complete
E_MUX_REMOTEDISCONNECT	Remote forced disconnection
E_MUX_TIMEOUTDISCONNECT	Forced disconnection, timeout
E_MUX_LOGOFFDISCONNECT	Forced disconnection, client logoff
E_MUX_USERDISCONNECT	User cancelled connection
E_MUX_NOCONNECTION	Connection handle invalid or connection removed by other end
E_MUX_ABORT	IrMUX has been destroyed

LM_WriteRequest **Queue a write request on a connection**

Message Number 25

Use this function call to queue a write request on a connection.

The maximum length of a data frame that can be sent is returned as part of the connection parameters from a connect request. Do not exceed this length.

Arguments

Connection Handle	Connection handle
Argument 1	Pointer to a buffer containing the data to send
Argument 2	Pointer to a UINT containing the number of bytes to send
Argument 3	TRUE if more data is to follow
Value of status in the E_MESSAGE struct on return:	
OKAY	Write successful
E_GEN_NOMEMORY	No memory to complete request
E_MUX_NOCONNECTION	Connection handle invalid or connection removed by other end
(various)	Errors from IrLAP

LM_UReadRequest **Queue an unreliable read request**

Message Number 26

Use this function call to queue an unreliable read request on a connection.

In unreliable reading, data is sent only once.

This function will panic the client process if a read is already queued on that connection.

Arguments

Connection Handle	Connection handle
Argument 1	Pointer to a buffer to receive the data on return
Argument 2	Pointer to a UINT containing the number of bytes required; on return the UINT contains the actual amount of data returned
Argument 3	Irrelevant

Value of status in the `E_MESSAGE` struct on return:

<code>OKAY</code>	Read complete
<code>E_MUX_REMOTEDISCONNECT</code>	Remote forced disconnection
<code>E_MUX_TIMEOUTDISCONNECT</code>	Forced disconnection, timeout
<code>E_MUX_LOGOFFDISCONNECT</code>	Forced disconnection, client logoff
<code>E_MUX_USERDISCONNECT</code>	User cancelled connection
<code>E_MUX_NOCONNECTION</code>	Connection handle invalid or connection removed by other end
<code>E_MUX_ABORT</code>	IrMUX has been destroyed

LM_UWriteRequest **Queue an unreliable write request**

Message Number 27

Use this function call to queue an unreliable write request on a connection.

In unreliable writing, data is sent only once.

Arguments

Connection Handle	Connection handle
Argument 1	Pointer to a buffer containing the data to send
Argument 2	Pointer to a <code>UINT</code> containing the number of bytes to send
Argument 3	<code>TRUE</code> if more data to follow
Value of status in the <code>E_MESSAGE</code> struct on return:	
<code>OKAY</code>	Write successful
<code>E_GEN_NOMEMORY</code>	No memory to complete request
<code>E_MUX_NOCONNECTION</code>	Connection handle invalid or connection removed by other end
(various)	Errors from IrLAP

LM_AccessModeRequest **Obtain/release exclusive access**

Message Number 22

Use this function call to obtain/release exclusive access to IrMUX.

Arguments

Connection Handle	Connection handle
Argument 1	<code>LM_ExclusiveMode</code> to obtain exclusive access or <code>LM_MultiplexMode</code> to release exclusive access to IrMUX
Argument 2	Irrelevant
Argument 3	Irrelevant

Value of status in the `E_MESSAGE` struct on return:

<code>OKAY</code>	Request complete
<code>E_GEN_MEMORY</code>	Not enough memory to complete the request
<code>E_GEN_INUSE</code>	Unable to enter exclusive mode
<code>E_MUX_BUSY</code>	Another client is attempting to enter exclusive mode
<code>E_MUX_EXCLUSIVE</code>	Another client already has exclusive access
<code>E_MUX_NOCONNECTION</code>	Connection handle invalid or connection removed by other end
<code>E_MUX_ABORT</code>	IrMUX is being destroyed
<code>E_GEN_NSUP</code>	Argument 1 does not contain either <code>LM_ExclusiveMode</code> or <code>LM_MultiplexMode</code>
<code>E_MUX_REMOTEDISCONNECT</code>	Remote forced disconnection
<code>E_MUX_TIMEOUTDISCONNECT</code>	Forced disconnection, timeout
<code>E_MUX_LOGOFFDISCONNECT</code>	Forced disconnection, client logoff
<code>E_MUX_USERDISCONNECT</code>	User cancelled connection

LM_IdleRequest **Place the connection into Idle/Active mode**

Message Number 23

Use this function call to place the connection into Idle/Active mode.

This implementation does not follow the IrMUX specification in that communication can be carried out in both Active and Idle modes. Being in active mode simply acts as a block to other clients placing IrMUX into Exclusive mode.

A connection always starts in Idle mode. To stop other clients gaining exclusive access to IrMUX by entering Active mode, a client must put its own connection into Active mode. It is recommend that Active mode is only used during critical periods of data communication.

Arguments

Connection Handle	Connection handle
Argument 1	Irrelevant
Argument 2	Irrelevant
Argument 3	TRUE for Idle mode, FALSE for Active mode

Value of status in the `E_MESSAGE` struct on return:

<code>OKAY</code>	Mode change successful
<code>E_MUX_BUSY</code>	Connection busy
<code>E_MUX_NOCONNECTION</code>	Connection handle invalid or connection removed by other end

LM_SetHandshakingLevel **Set retries on each data frame**

Message Number 28.

Use this function call to set the number of retries on each data frame, if the client application is not responding to incoming data quickly enough.

This should only be enabled when in Exclusive mode. When use of exclusive mode has finished `LM_SetHandshakingLevel` should be used to reset the number of retries to 1.

Arguments

Connection Handle	Connection handle
Argument 1	Irrelevant
Argument 2	Irrelevant
Argument 3	The number of times to resend each frame, 0 (zero) for inactive.

LM_DisconnectRequest**Disconnect**

Message Number 20

Use this function call to disconnect. This will cancel any requests on the connection.

Arguments

Connection Handle	Connection handle
Argument 1	Irrelevant
Argument 2	Irrelevant
Argument 3	Irrelevant

Value of status in the E_MESSAGE struct on return:

OKAY	Disconnect complete
E_MUX_NOCONNECTION	Connection handle invalid or connection removed by other end
E_MUX_BUSY	IrMUX is busy. The application should wait a short time and try again. May be ignored if logging off.

CHAPTER 18

FAST CHARGER

Introduction

This chapter describes Fast Charger services for the HC and *Workabout* docking stations.

In this document the term ‘Docking Station’ refers both to the HC docking station, (which connects to an HC LIF interface fitted to the bottom of the HC), and to the *Workabout* docking station. The term ‘computer’ refers to an HC or *Workabout*.

Docking Station services

There are two variants of the docking station: one model has been designed for the HC computer and one model has been designed for the *Workabout* computer. The docking station may be used to fast charge the battery inside the "docked" computer.

The docking station FCG: device driver may be used to charge (or discharge) the main battery under the direct control of an HC/*Workabout* computer which has been inserted in the docking station holster. The device driver may also be used to supply that computer with information about the battery charge status.

Notes:

1. the HC/*Workabout* docking station expansion port **cannot** be used whilst using the (FCG:).
2. the HC/*Workabout* docking station will charge batteries without any external software control from a docked computer.

See also the *Cradle and Docking Station* chapter for details of the cradle device driver CRD:.

Using the Docking Station

Use of the Docking Station is complicated by the fact that one high speed serial channel is used to communicate to three possible interfaces.

1. The interface in the bottom of the HC (i.e. Barcode, RS232 etc.)
2. The electronics in the device
3. Any expansion fitted to the device.

As a result of this, only one interface at a time can be opened on this channel, i.e. either the RS232 in the HC bottom expansion (port TTY:B) or the RS232 in the Docking Station expansion slot (port TTY:C).

HC/*Workabout* docking station fast charger services

The FCG: device driver will allow a C application running on a machine in the docking station holster to monitor and control the state of the docking station fast charger circuitry. The docking station is capable of charging, discharging and performing a capacity check of the main battery inside the computer in a variety of operating modes. It may also be used to charge (but not discharge) a spare battery installed in the spare battery compartment of the docking station.

Important note: the fast charger circuit in the docking station will charge batteries automatically **without requiring any explicit calls to the FCG: device driver**; only in a few circumstances should it actually be

necessary to use FCG: (as described in this chapter) to give implicit instructions to the docking station. In normal circumstances, it is **not** necessary to run software on the SIBO machine to control or monitor the battery charging/discharging process.

There are two variants of the fast charger device driver, where each driver is tailored for use with one specific SIBO computer and docking station. The fast charger device driver which may be used with an HC computer is called `sys$chgh.ldr`; the fast charger device driver which may be used with a *Workabout* computer is called `sys$fchg.ldr`. Both device drivers may be found in the path `\sibosdk\lib\` on installation of the relevant SIBO C Software Development Kit optional components onto your PC (from the `ldr.zip` file).

The library file `fcharge.h` (located in the path `\sibosdk\include\`) contains C constant declarations which may be used with either the `sys$chgh.ldr` or the `sys$fchg.ldr` device driver.

The fast charger does not allow a direct measurement of the status of each battery. As a result, the FCG: device drive will determine the status of each battery indirectly by measuring the charge pulses applied by the charger circuitry to the battery pack over a short period of time. For this reason, some of the FCG: I/O operations may take a number of seconds to complete.

The fast charger device driver will only run if the Psion computer is resting in the docking station holster.

Fast charging batteries

The docking station may be used to charge the battery inside the "docked" computer, or a battery located in the spare battery compartment on the top of the docking station. Only one battery pack can be charged at any one time.

The fast charge circuitry possesses four modes of charging:

fast charge	charge the battery pack with a preset current, and for a preset charge period (or shorter, depending upon the charge status of the battery).
top off	the battery pack is now 90-95% fully-charged; supply a lower current (typically C/10) after completion of the fast charge, to fully-charge the cells
trickle charge	supply a small charge current (typically C/30) to maintain the charge in a fully-charged battery pack
discharge	discharge the main battery at a fixed discharge current

If you wish to control the fast charger explicitly from a C program running on the "docked" Psion computer, then the program should carry out the following steps:

1. open a channel to the fast charger device driver with the call `p_open(VOID **ppcb, "FCG:A", -1);`
2. set up the charge current and maximum charging time period with a call to `FCHG_SETCARGEMODE` (if this step is omitted, then default settings will be used).
3. instruct the device driver to charge the main battery (or the spare battery) with `FCHG_FASTCHARGE`. This command will commence the charging of the battery at a default current and over a default time period.
4. close the channel to FCG: with `p_close(VOID *pcb)`

Whilst the FCG: channel is open, the C program may also monitor the progress of the charging process, if required. Calls to `FCHG_FASTCHARGE` may be omitted if the C program wishes only to monitor the progress of the charging process.

The maximum charge time period set with a `FCHG_SETCARGEMODE` call is used as a backup safety feature to ensure that the fast charging process terminates before any damage can occur to the battery pack. For this reason, **it is essential that the charge time is set correctly, in relation to the charge rate which you wish to use**. The two main terminators used by the fast charge circuitry to detect the end of the charging process are voltage and temperature slope. These two terminators are managed by the fast charger hardware.

IMPORTANT: Never charge a standard Psion battery pack with a charge rate that exceeds 1C without first contacting the battery supplier for advice. High charge rates in excess of 1C will reduce the life of the battery pack and may create a risk of explosion or fire.
--

Measuring battery capacity

The FCG: device driver may be used to determine the charge capacity of a battery pack. If the time taken to fully-discharge a fully-charged battery pack is measured, then the relative battery charge capacity may be determined from the following equation:

$$M = (10 * T_m * I_r) / (6 * C_r)$$

where:

- M = the reduction in battery charge capacity of the battery under test (%)
- T_m = measured discharge time (minutes)
- I_r = discharge current (mA)
- C_r = rated battery capacity of battery under test (mAh)

Rated charge capacity of standard Psion battery packs

The rated charge capacity for the standard Psion HC and *Workabout* battery packs is displayed in the table below. This table also contains the standard discharge time and the standard discharge current for each variant of the docking station. .

Docking station	Rated capacity C_r (mAh)	Discharge time T_m (mins)	Discharge current I_r (mA)
HC	650	130	300
<i>Workabout</i>	845	175	290

Example calculations

Example 1: If a fully-charged HC battery pack takes 100 minutes to discharge, at the (fixed) discharge current of 300mA, then the measured capacity of that HC battery pack is 77%.

$$\text{i.e. } M = (10 * T_m * I_r) / (6 * C_r) = (10 * 100 * 300) / (6 * 650) = 77\%$$

Example 2: If a fully-charged *Workabout* battery pack takes 120 minutes to discharge, at the (fixed) discharge current of 290mA, then the measured capacity of that *Workabout* battery pack is 69%.

$$\text{i.e. } M = (10 * 120 * 290) / (6 * 845) = 69\%$$

The charge capacity of custom batteries may also be determined using the same equation. Care must be taken to use the correct value of I_r in the equation, where I_r is defined by the docking station variant that is being used to discharge the custom battery pack.

p_open(FCG:A)

Open the fast charger device

```
INT p_open(VOID **ppcb, "FCG:A", -1);
```

Open a channel to the docking station fast charger device.

Returns zero if the device is opened successfully, otherwise a negative error. The p_open operation will fail if the docking station is not present.

If the channel is opened successfully, the charging current and charging time will be preset to the default settings. The device driver will then immediately begin polling the status of the battery. This polling operation may take up to forty seconds to complete.

Errors include:

E_GEN_NOMEMORY	failed to allocate memory for control block
E_FILE_LOCKED	port is already open
E_GEN_INUSE	port in use
E_FILE_DEVICE	device does not exist

p_close**Close the channel**

```
INT p_close(VOID *pcb);
```

Close the fast charger device channel.

Any outstanding FCHG_XXX operations will be cancelled by p_close.

Returns zero.

FCHG_SETCARGEMODE**Set battery charge mode**

```
#include "fcharge.h"
INT p_iow(VOID *pcb, FCHG_SETCARGEMODE, WORD *pmode);
```

Set the charge current and the maximum charging time period.

Two pieces of information must be passed to the docking station fast charger circuitry (in pmode):

- the current which will be used to charge the battery
- the maximum period of time which the battery will be charged.

The contents of pmode should contain a value which is constructed from two integers. The three least significant bits in pmode should identify the charge time period; the four most significant bits in pmode should identify the charge current. (The present charge mode may be read by calling the FCHG_READCHARGEMODE function.)

This information may be selected from the fixed options that are supported by the docking station. The minimum charging current that one could select should not be less than a quarter of the battery capacity (unless the battery pack is known to be already partially charged).

Example: The capacity of the standard *Workabout* battery is 850mA. The minimum charging current that one could use is $0.25 \times 850 = 212\text{mA}$ (approx. $200\text{mA} = \text{FCHG_CCLOW2}$).

Please refer to the tables below for information about the discrete charge currents which may be supplied by the docking station. This table also contains the names of the C constants, defined in *fcharge.h*, that are associated with each charge current setting for the docking station.

Charging Psion battery packs

The following tables display suggested combinations of charge current and charge times that are suitable for the standard Psion HC and *Workabout* battery packs:

Charge period	Charge current for standard Psion HC battery pack	Charge current for standard Psion <i>Workabout</i> battery pack
1C	700mA	850mA
C/2	350mA	500mA
C/4	175mA	200mA

The above charge period/charge currents correspond to the following C constants in *fcharge.h*:

Charge period	Charge current for standard Psion HC battery pack	Charge current for standard Psion <i>Workabout</i> battery pack
FCH_CTIME1C	FCHG_DEFAULTCC	FCHG_DEFAULTCC
FCH_CTIMEC2	FCH_CC350mA	FCH_CC500mA
FCH_CTIMEC4	FCH_CCLOW2	FCH_CCLOW2

High charging currents should **only** be selected in situations where special, high capacity, fast charge battery packs are being used. Lower charging currents are recommended in circumstances where the ambient temperature is above 35°C, or where lower capacity battery packs are used.

The minimum charging current that may be selected should not be less than a quarter of the battery capacity (unless the battery pack is known to be already partially-charged). Incorrect settings will result in under-charge or damage to batteries.

Charging custom battery packs

The docking station may also be used to charge custom battery packs with different charge-current requirements.

The fast charger device driver and docking station support the following charge times:

Charge period	<i>fcharge.h</i> constant	Maximum charge period (minutes)
2C	FCHG_CTIME2C	45
1C	FCHG_CTIME1C	90
C/2	FCHG_CTIMEC2	180
C/4	FCHG_CTIMEC4	360

The fast charger device driver and docking station support the following charge currents:

<i>fcharge.h</i> constant	HC docking station current	Workabout docking station current
FCHG_CCL0W1	100mA	130mA
FCHG_CCL0W2	175mA	200mA
FCHG_CC350mA	350mA	350mA
FCHG_CC500mA	500mA	500mA
FCHG_CC650mA	650mA	650mA
FCHG_DEFAULTCC	700mA	850mA
FCHG_CC850mA	850mA	850mA
FCHG_CC1000mA	1A	1A
FCHG_CC1100mA	1.1A	1.1A
FCHG_CCHIGH1	1.2A	1.3A
FCHG_CCHIGH2	1.4A	1.5A

IMPORTANT: Never charge a standard Psion battery pack with a charge rate that exceeds 1C without first contacting the battery supplier for advice. High charge rates in excess of 1C will reduce the life of the battery pack and may create a risk of explosion or fire.

Notes:

1. The standard power supply that is provided with the Workabout docking cradle has a maximum current rating of 1A. As a result, battery charge currents in excess of 850 mA **cannot** be supported by the standard power supply provided with the docking station.

If you wish to fast charge batteries using currents greater than 850 mA, a 2A power supply must be used with the Workabout docking station. Suitable power supplies are available from Psion; for further details please contact the Psion Sales staff.

The HC docking station **is** provided with a 2A power supply and hence is capable of handling the entire charge current range supported by the device driver.

2. Charging currents below or equal to 200mA have a tolerance of $\pm 30\%$ and currents greater than 300mA have a tolerance of $\pm 15\%$.

FCHG_READCHARGEMODE**Read battery charge mode**

```
#include "fcharge.h"
INT p_iow(VOID *pcb, FCHG_READCHARGEMODE, WORD *pmode);
```

Read the charge current and the maximum charging time period settings.

The present charge rate and the required charging period will be returned to *pmode*.

The contents of *pmode* will contain a value which is constructed from two integers. The three least significant bits in *pmode* will identify the charge time period; the four most significant bits in *pmode* will identify the charge current. The charge mode may be altered by calling the `FCHG_SETCARGEMODE` function.

The table that appears in the `FCHG_SETCARGEMODE` section (above) contains all of the valid combinations of charge current/charge period settings for HC and *Workabout* machines.

Zero is returned if the `FCHG_SETCARGEMODE` request completed successfully, or a negative value if an error has occurred.

FCHG_READSTATUS**Read the battery status**

```
#include "fcharge.h"
INT p_iow(VOID *pcb, FCHG_READSTATUS, WORD *pstate1, WORD *pstate2);
```

Read the present charge status of each battery from the docking station device, writing the new charge states to *pstate1* and *pstate2*.

The status of the main battery inside the computer held in the holster is returned in *pstate1*; if a spare battery has been inserted in the spare battery compartment of the docking station, then the status of this battery will be returned to *pstate2*.

On completion, *pstate1* and *pstate2* may contain one of the following integer values:

<code>FCHG_UNKNOWN</code>	the battery is in an unknown state
<code>FCHG_NOTCHARGING</code>	the battery is not charging. If the docking station has been instructed to discharge the main battery, then <code>FCHG_READSTATUS</code> will return a <code>FCHG_NOTCHARGING</code> value to indicate that the battery is indeed being discharged
<code>FCHG_FASTCHARGING</code>	The battery is fast charging
<code>FCHG_TOPPINGOFF</code>	The battery is now charged to 90-95% of its maximum capacity and it is now being topped off by the fast charger circuitry
<code>FCHG_TRICKLE</code>	The battery is being trickle charged

The `FCG` device driver calculates the present charge status of a battery by measuring the duration of charge pulses that the battery receives from the docking station. A charge pulse from the docking station may last 40 seconds; as a result, it is possible that a call to `FCHG_READSTATUS` may not be able to determine the present charge status of a battery at the precise moment in time when the call is made.

In this situation, a `FCHG_UNKNOWN` status message will be returned for the status of that battery. This does not mean that the battery charge status is indeterminate; another `FCHG_READSTATUS` call in the future may, however, be able to return a valid status value for the charge status of that battery.

Zero is returned if the `FCHG_READSTATUS` request completed successfully, or a negative value if an error has occurred.

FCHG_ASYNCREAD Read the battery status asynchronously

```
#include "fcharge.h"
INT p_ioc(VOID *pcb, FCHG_ASYNCREAD, WORD *pstate1, WORD *pstate2);
```

Read the present charge status of each battery from the docking station device asynchronously, writing the new charge states to `pstate1` and `pstate2` when the charge status of either battery alters.

The `FCHG_ASYNCREAD` operation may be cancelled at any time by a call to `FCHG_CANCEL`.

`FCHG_ASYNCREAD` will return immediately after initiating a request to receive battery status information when the status of either battery changes. At some time in the future, the new status of the main battery inside the computer in the holster will be returned in `pstate1`; if a spare battery has been fitted inside the compartment in the top of the docking station, then the status of this battery will be returned to `pstate2`.

Panics if `pcb` is not a valid channel handle, or if an `FCHG_ASYNCREAD` operation is outstanding.

Returns zero if the `FCHG_ASYNCREAD` request completed successfully, or a negative value if an error has occurred.

FCHG_CANCEL Cancel an asynchronous read

```
#include "fcharge.h"
INT p_iow(VOID *pcb, FCHG_CANCEL);
```

Cancel any outstanding `FCHG_ASYNCREAD` request. Performing a cancel is harmless if no read request is outstanding.

Returns zero.

FCHG_FASTCHARGE1 Fast charge the main battery

```
#include "fcharge.h"
INT p_iow(VOID *pcb, FCHG_FASTCHARGE1);
```

Begin fast charging the main battery that is inside the computer fitted to the docking station holster.

The present settings for the charge current and charge time period (as set by `p_open` or `FCHG_SETCHARGE MODE`) will be used by the fast charger circuitry to charge the battery.

`FCHG_FASTCHARGE1` will return zero if it completes successfully, or a negative value if an error occurs.

Please note: only one battery may be charged at any one time; i.e. the main battery cannot be charged while a battery in the spare battery compartment is being charged.

If you wish to charge the main battery *and* the spare battery, then the software which calls `FCG`: should charge one battery, sense when that battery is charged sufficiently, and *then* charge the second battery.

FCHG_FASTCHARGE2 Fast charge the spare battery

```
#include "fcharge.h"
INT p_iow(VOID *pcb, FCHG_FASTCHARGE2);
```

Begin fast charging the battery that is in the spare battery compartment of the docking station.

The present settings for the charge current and charge time period (as set by `p_open` or `FCHG_SETCHARGE MODE`) will be used by the fast charger circuitry to charge the battery.

`FCHG_FASTCHARGE2` will return zero if it completes successfully, or a negative value if an error occurs.

Please note: only one battery may be charged at any one time; i.e. a battery in the spare battery compartment cannot be charged while the main battery is being charged.

If you wish to charge the main battery *and* the spare battery, then the software which calls `FCG`: should charge one battery, sense when that battery is charged sufficiently, and *then* charge the second battery.

Note: if the main battery is charging and the spare battery is waiting, then the `FCHG_FASTCHARGE2` function will over-ride the status of the main battery and start charging the spare battery instead.

FCHG_DISCHARGE1**Discharge the main battery**

```
#include "fcharge.h"
INT p_iow(VOID *pcb, FCHG_DISCHARGE1);
```

Begin discharging the main battery fitted inside the computer fitted to the docking station holster.

The FCHG_DISCHARGE1 command will initiate a sequence of operations which will take a few seconds to complete. If any additional calls are made to FCG: in this period, those commands will return a negative completion code and the instructions will be ignored.

The discharging current is fixed by the fast charger circuitry and may therefore not be altered by software. The discharge current for the HC docking station is set to 300 mA $\pm 3\%$. The discharge current for the *Workabout* docking station is set to 290 mA $\pm 3\%$.

FCHG_DISCHARGE1 will return zero if it completes successfully, or a negative value if an error occurs.

Notes:

1. when the docking station has been instructed to discharge a battery, subsequent FCHG_READSTATUS operations will each return a FCHG_NOTCHARGING status code - which should be interpreted as meaning that the battery is discharging (as expected).
2. The spare battery cannot be discharged by FCG:.

Example program

An example C application may be found in the file path `\sibosdk\wkdemo\`, on installation of this SDK onto your PC.

charger.c is a *Workabout* computer program which monitors and controls the fast charger circuitry in the docking station. This program allows the main battery, or spare battery to be charged, or the main battery to be discharged, and the status of both batteries to be monitored.

Note: This program will only run if the *Workabout* computer has been placed into the holster of a docking station.

```
/* Fastcharger for Workabout */
/* Example application */
/* (C) Copyright Psion Software PLC 1997 */

#include <p_std.h>
#include <p_file.h>
#include <p_sys.h>
#include <p_keyb.h>
#include <epoc.h>

#include "fcharge.h"

GLREF_D VOID *winHandle;

LOCAL_C VOID State(WORD Val)
{
    switch (Val)
    {
        case 0:
            p_printf(" Unknown");
            break;
        case 1:
            p_printf(" Not Charging");
            break;
        case 2:
            p_printf(" Fast Charging");
            break;
        case 3:
            p_printf(" Topping Off");
            break;
        default:
            p_printf(" Trickle");
    }
}
```

```

GLDEF_C VOID main(VOID)
{
    VOID *pcb;
    WORD One,OldOne;
    WORD Two,OldTwo;
    WORD KStat,Disch;
    WORD LogOne[100];
    WORD LogTwo[100];
    WORD P1=0,P2=0,Temp;
    ULONG TTime;
    P_CON_KBREC Key;
    TEXT X1Str[11];
    WORD X1=0;
    ULONG XE=0,DischTime;
    TEXT *X1Ptr;
    INT Item;
    for(Item=0;Item<100;Item++)
    {
        LogOne[Item]=0;
        LogTwo[Item]=0;
    }

    p_devdel("FCG",E_LDD);
    p_printf("Loading");
    if(p_loadldd("m:\\sys$fchg. ldd")<0)
        p_panic(0);
    p_printf("Opening test");
    if(p_open(&pcb,"FCG:A",-1)<0)
        p_panic(1);
    p_printf("Closing test");
    if(p_close(pcb)<0)
        p_panic(2);
    p_printf("Opening test part II");
    if(p_open(&pcb,"FCG:A",-1)<0)
        p_panic(3);
    p_printf("Cant remove test");
    if(p_devdel("FCG:",E_LDD)>=0)
        p_panic(4);
    p_printf("Setting the charge Mode");
    p_iow(pcb,P_FCHG_READCHARGEMODE,&One);
    if (One!=(P_FCHG_DEFAULTCC | P_FCHG_CTIME1C))
        p_panic(5);
    p_printf("Test the charge Mode");
    if (p_iow(pcb,P_FCHG_SETCARGEMODE,&One)!=0)
        p_panic(6);
    p_printf("Set Rubbish charge Mode");
    One=-1;
    if (p_iow(pcb,P_FCHG_SETCARGEMODE,&One)==0)
        p_panic(7);
    p_iow(pcb,P_FCHG_READSTATUS,&One,&Two);
    p_print("Battery one:");
    State(One);
    p_print("Battery two:");
    State(Two);

    /*
        p_printf("Asynch test1");
        p_iow(pcb,P_FCHG_ASYNCREAD,&One,&Two);
        p_print("Battery one:");
        State(One);
        p_print("Battery two:");
        State(Two);
        p_printf("Asynch test2");
        p_iow(pcb,P_FCHG_ASYNCREAD,&One,&Two);
        p_print("Battery one:");
        State(One);
        p_print("Battery two:");
        State(Two);*/

```

```
p_ioc4(winHandle,P_FREAD,&KStat,&Key);
TTime=p_date();
Disch=0;
DischTime=0;
FOREVER
{
    if (KStat!=E_FILE_PENDING) /* Was it the keyboard */
    {
        p_iowait(); /* not really the way to do this -oh well */
        if (Key.keycode=='Q' || Key.keycode=='q' || Key.keycode==27)
            p_exit(0);
        else if (Key.keycode=='D' || Key.keycode=='d')
        {
            p_printf("Start discharge");
            TTime=p_date();
            if (p_iow(pcb,P_FCHG_DISCHARGE))
                p_panic(8);
            Disch=1;
        }
        else if (Key.keycode=='F' || Key.keycode=='f')
        {
            p_printf("Start Fastcharge1");
            TTime=p_date();
            if (p_iow(pcb,P_FCHG_FASTCHARGE1))
                p_panic(9);
        }
        else if (Key.keycode=='G' || Key.keycode=='g')
        {
            p_printf("Start Fastcharge2");
            TTime=p_date();
            if (p_iow(pcb,P_FCHG_FASTCHARGE2))
                p_panic(9);
        }
        else if (Key.keycode=='C' || Key.keycode=='c')
        {
            p_getl("Current: ",&X1Str[0],10);
            X1Ptr=&X1Str[0];
            p_stoi(&X1Ptr,&X1);
        }
        else if (Key.keycode=='/' || Key.keycode=='?')
        {
            if (Disch==1)
                p_printf("Discharging: %d", (p_date()-TTime));
            else
            {
                p_printf("Discharge Time: %dl",DischTime);
                if (X1>0 && DischTime>0)
                {
                    XE=(850*60*60*10)/X1;
                    XE=(DischTime*100*10)/XE;
                    p_printf("Capacity (Cell) %d%%", (UWORD)(XE));
                }
            }
        }
    }

    else if (Key.keycode=='1')
    {
        p_printf("Battery one Log");
        Temp=0;
        while (Temp<P1)
            if (LogOne[Temp++]>0)
                State(LogOne[Temp-1]);
    }
    else if (Key.keycode=='2')
    {
        p_printf("Battery two Log");
        Temp=0;
        while (Temp<P2)
            if (LogTwo[Temp++]>0)
                State(LogTwo[Temp-1]);
    }
}
```

```

else if (Key.keycode=='H' || Key.keycode=='h' || Key.keycode==290 ||
        Key.keycode==291)
{
    p_printf("");
    p_printf("C...Enter Discharge current");
    p_printf("D...Start Discharge");
    p_printf("F...Fastcharge Battery1");
    p_printf("G...Fastcharge Battery2");
    p_printf("?...Show Discharge time");
    p_printf("1...Show Battery1 Log");
    p_printf("2...Show Battery2 Log");
    p_printf("Q...Quit");
    p_printf("");
}
p_ioc4(winHandle,P_FREAD,&KStat,&Key);
}
OldOne=One;
OldTwo=Two;
p_iow(pcb,P_FCHG_READSTATUS,&One,&Two);
if (OldOne!=One || OldTwo!=Two)
{
    if (OldOne==1 && Disch==1 && One!=1)
    {
        Disch=0;
        DischTime=(p_date()-TTime);
        p_printf("Discharge Time: %d",DischTime);
    }
    p_printf("Time: %d", (p_date()-TTime));
    if (One!=OldOne)
    {
        if (P1>50)
            P1=0;
        LogOne[P1++]=One;
    }

    if (Two!=OldTwo)
    {
        if (P2>50)
            P2=0;
        LogTwo[P2++]=Two;
    }
    p_print("Battery one:");
    State(One);
    p_print("Battery two:");
    State(Two);
}
}
}

```


INDEX

- .wve files
 - digital sound files, 5-1
- A_FTIMITED
 - alarm device, 6-2
- A_FTIMITED_X
 - alarm device, 6-3
- A_FUNTIMITED
 - alarm device, 6-3
- A_FUNTIMITED_X
 - alarm device, 6-4
- AccessIr
 - infrared API, 16-1
- AccessIr API
 - infrared, 15-4
- AIR:
 - infrared device, 15-4, 16-1, 16-2
 - loading on Siena, 16-1
- alarm
 - device I/O introduction, 6-1
 - note sequence SND: device, 5-3
- alarm device
 - A_FTIMITED, 6-2
 - A_FTIMITED_X, 6-3
 - A_FUNTIMITED, 6-3
 - A_FUNTIMITED_X, 6-4
 - ALM: introduction, 6-1
 - p_close, 6-2
 - P_FCANCEL, 6-2
 - p_open(ALM:), 6-2
 - panics, 6-2
 - services additional S3a, 6-3
 - services S3 S3a & MC, 6-2
- alarm services
 - time application S3 S3a, 6-2
- ALM: device
 - introduction, 6-1
 - see alarm device, 6-1
- asynchronous
 - I/O functions, 1-1
- bar code
 - Codabar start and stop characters, 14-5
 - maximum scanning rate, 14-4
 - supplement digits, 14-4
- bar code command syntax
 - HC bar code/RS232 module, 14-11
- bar code commands
 - HC bar code/RS232 module, 14-11
- bar code formats UPC/EAN
 - HC bar code/RS232 module, 14-6
- bar code interface
 - comms settings, 14-2
 - HC bar code/RS232 module, 14-3
- bar code symbologies
 - HC bar code/RS232 module, 14-4
- bar code wand connector
 - HC bar code/RS232 module, 14-2
- BAR: device
 - see HC bar code reader, 13-1
- battery capacity
 - docking station fast charger, 18-3
 - HC, 18-3
 - Workabout, 18-3
- baud rates
 - infrared, 15-3
- beaming
 - infrared communications application, 15-6
- bottom slot
 - HC bar code/RS232 module, 14-2
- CLIB
 - programs CON: device use of, 2-1
- Codabar
 - start and stop characters, 14-5
- communications
 - infrared application, 15-6
- CON: device
 - automatic opening, 2-1
 - CLIB based programs, 2-1
 - example code, 2-14
 - explicit opening, 2-3
 - handle of, 2-2
 - introduction, 2-1
 - OPL programs, 2-1
 - PLIB based programs, 2-2
 - see console device, 2-1
- Condor chip
 - serial infrared, 15-3
- connection
 - second time (LM-IAS server)
 - infrared, 17-4
- connection first time (LM-IAS server)
 - infrared, 17-4
- connectionless calls
 - infrared, 17-3
- connectionless read request
 - infrared, 17-3
- connectionless write request
 - infrared, 17-3
- connection-oriented calls
 - infrared, 17-3
- console device
 - automatic opening of CON: device, 2-1
 - example code, 2-14
 - explicit opening of CON: device, 2-3
 - handle of, 2-2
 - I/O, 2-1
 - p_close, 2-4
 - P_EVENT_READ, 2-9
 - P_EVENT_TEST, 2-10
 - P_FCANCEL, 2-5
 - P_FEDIT, 2-6
 - P_FFLUSH, 2-6
 - P_FINQ, 2-10
 - P_FREAD, 2-5
 - P_FSENSE, 2-6

- P_FSET service call convention, 2-3
- P_FTEST, 2-5
- P_FWFLUSH, 2-10
- p_open(CON:), 2-3
- P_SCR_ATTRB, 2-11
- P_SCR_CANCEL_CAPTURE_KEY, 2-13
- P_SCR_CAPTURE KEY, 2-13
- P_SCR_CLIENT_FOREGROUND, 2-12
- P_SCR_CLR, 2-7
- P_SCR_COMPATIBILITY, 2-9
- P_SCR_CSET, 2-10
- P_SCR_CURSOR, 2-8
- P_SCR_DISABLE_READS, 2-12
- P_SCR_ESCAPE, 2-9
- P_SCR_FLUSH, 2-12
- P_SCR_FONT, 2-11
- P_SCR_GREY, 2-9
- P_SCR_LAST_LINE_WRAP, 2-12
- P_SCR_NEL, 2-8
- P_SCR_POSA, 2-8
- P_SCR_POSR, 2-8
- P_SCR_SCROLL, 2-7
- P_SCR_SLOCK, 2-8
- P_SCR_WLOCK, 2-8
- P_SCR_WSET, 2-7
- p_write, 2-4
- panics, 2-3
- services, 2-3
- services additional, 2-9
- cradle device
 - introduction, 11-1
 - introduction HC, 11-1
 - p_close(CRD:), 11-2
 - P_FCANCEL, 11-2
 - P_FREAD, 11-2
 - P_FSENSE, 11-2
 - P_FSET, 11-2
 - p_open(CRD:), 11-1
 - services, 11-1
- CRC
 - XYmodem device, 9-1
- CRD: device
 - introduction, 11-1
 - see cradle device, 11-1
 - services, 11-1
- data transfer
 - link I/O, 10-1
- database
 - world application services S3 S3a, 8-2
- device driver
 - FCG: docking station, 18-2
 - infrared accessir.ldd, 16-1
 - serial port LDD, 4-1
 - serial port PDD, 4-1
 - Siena infrared printing, 15-7
- devices
 - AIR: infrared, 15-4, 16-1, 16-2
 - alarm I/O introduction, 6-1
 - ALM: introduction, 6-1
 - BAR: introduction, 13-1
 - CON: introduction, 2-1
 - console I/O, 2-1
 - CRD: introduction, 11-1
 - FCG: docking station, 18-1
 - FRC: I/O introduction, 7-1
 - FRC: introduction, 7-1
 - HC bar code reader I/O introduction, 13-1
 - HC bar code/RS232 module, 14-1
 - HC cradle I/O introduction, 11-1
 - HC docking station I/O introduction, 11-1
 - HC intelligent bar code Reader, 14-1
 - HC magnetic card reader I/O, 12-1
 - I/O introduction, 1-1
 - I/O Introduction, 1-1
 - IRP: infrared printing, 15-7
 - MCR: introduction, 12-1
 - NCP I/O introduction, 10-1
 - PAR: introduction, 3-1
 - parallel port I/O, 3-1
 - serial port I/O introduction, 4-1
 - SIR: infrared, 15-2, 15-3
 - SND: introduction, 5-1
 - sound I/O introduction, 5-1
 - TTY: introduction, 4-1
 - WLD: introduction, 8-1
 - Workabout docking station I/O introduction, 11-1
 - world database I/O introduction, 8-1
 - XMD: introduction, 9-1
 - Xmodem I/O introduction, 9-1
 - Ymodem I/O introduction, 9-1
- dial tones
 - sound device DTMF, 5-4
- disconnecting first time (LM-IAS server)
 - infrared, 17-4
- disconnecting second time
 - infrared, 17-4
- discovery
 - infrared, 15-5, 17-4
- DISCOVERY_LOG
 - structure infrared, 15-5
- docking station
 - battery capacity calculations, 18-3
 - device introduction HC, 11-1
 - device introduction Workabout, 11-1
 - fast charger, 18-1
 - FCG: device, 18-1
 - FCG: device driver, 18-2
 - FCHG_ASYNCREAD, 18-7
 - FCHG_CANCEL, 18-7
 - FCHG_DISCHARGE, 18-8
 - FCHG_FASTCHARGE1, 18-7
 - FCHG_FASTCHARGE2, 18-7
 - FCHG_READCHARGEMODE, 18-6
 - FCHG_READSTATUS, 18-6
 - FCHG_SETCARGEMODE, 18-4
 - p_close, 18-4
 - p_open, 18-3
- docking station device
 - p_close(CRD:), 11-2
 - P_FCANCEL, 11-2
 - P_FREAD, 11-2
 - P_FSENSE, 11-2
 - P_FSET, 11-2
 - p_open(CRD:), 11-1

- services, 11-1
- DTMF
 - sound device dial tones, 5-4
- E_FALARM
 - sound device, 5-3
- E_FDIAL
 - sound device, 5-4
- E_FSSOUNDCHANNELn
 - sound device, 5-3
- E_MESSAGE struct
 - structure infrared, 17-5
- example application
 - infrared transfer, 16-7
- example program
 - fast charger Workabout, 18-8
 - infrared transfer, 16-7
- exclusive mode
 - infrared, 17-4
- expansion module
 - HC bar code/RS232 module, 14-2
- extension file
 - world database, 8-9
- fast charger
 - docking station, 18-1
 - example program Workabout, 18-8
 - FCHG_ASYNCREAD, 18-7
 - FCHG_CANCEL, 18-7
 - FCHG_FASTCHARGE1, 18-7
 - FCHG_FASTCHARGE2, 18-7
 - FCHG_FASTDISCHARGE1, 18-8
 - FCHG_READCHARGEMODE, 18-6
 - FCHG_READSTATUS, 18-6
 - FCHG_SETCARGEMODE, 18-4
 - introduction, 18-1
 - p_close(FCG:A), 18-4
 - p_open(FCG:A), 18-3
 - services, 18-1
- FCG:
 - device, 18-1
- FCHG_ASYNCREAD
 - fast charger, 18-7
- FCHG_CANCEL
 - fast charger, 18-7
- FCHG_FASTCHARGE1
 - fast charger, 18-7
- FCHG_FASTCHARGE2
 - fast charger, 18-7
- FCHG_FASTDISCHARGE1
 - fast charger, 18-8
- FCHG_READCHARGEMODE
 - fast charger, 18-6
- FCHG_READSTATUS
 - fast charger, 18-6
- FCHG_SETCARGEMODE
 - fast charger, 18-4
- file format
 - world database, 8-9
 - world database extension, 8-10
- FRC: device
 - I/O introduction, 7-1
 - introduction, 7-1
 - p_close, 7-2
 - P_FCANCEL, 7-2
 - P_FREAD, 7-2, 7-3
 - P_FSTART, 7-2
 - p_open(FRC:), 7-1
 - services S3a & Workabout, 7-1
- free running counter
 - see FRC: device, 7-1
- get value by class frame
 - infrared, 17-2
 - infrared IAS, 17-3
- get value by class reply frame
 - infrared IAS, 17-3
- handshaking level
 - infrared, 17-4
- HC
 - battery capacity calculation, 18-3
- HC bar code device
 - p_close, 13-2
 - P_FCANCEL, 13-2
 - P_FREAD, 13-2
 - p_open(BAR:), 13-2
- HC bar code reader
 - device driver, 13-1
 - hardware description, 13-1
 - interface module, 13-1
 - services, 13-2
 - wand, 13-1
- HC bar Code reader
 - device I/O introduction, 13-1
- HC bar code/RS232 module
 - 2-pin header, 14-3, 14-4
 - 5v regulated power supply, 14-4
 - auto power-off HC, 14-4
 - bar code command syntax, 14-11
 - bar code commands, 14-11
 - bar code formats UPC/EAN, 14-6
 - bar code symbologies, 14-4
 - bar code wand connector, 14-2
 - bottom slot, 14-2, 14-3
 - charging the main battery, 14-3
 - CTS input, 14-2
 - DCD input, 14-2, 14-3
 - device introduction, 14-1
 - diode isolation, 14-3
 - DSR auto wakeup switch, 14-3
 - DSR input, 14-2, 14-3
 - DTR output, 14-2, 14-3
 - example code, 14-20
 - expansion module, 14-2
 - HBCR-1610 series micro controller, 14-3
 - idle current of interface, 14-4
 - IGN_CTS serial port characteristic, 14-3
 - interface comms settings, 14-3
 - losing micro controller program data, 14-4
 - maximum output current, 14-2
 - optional Vsup connection, 14-2
 - p_close, 14-18
 - P_FREAD, 14-18
 - P_FSENSE, 14-18
 - P_FSET, 14-18
 - P_FWRITE, 14-19
 - p_open(TTY:), 14-17
 - power consumption, 14-2

- power consumption wand, 14-4
- powering an external device, 14-4
- programming, 14-1
- RI pin of the RS232 port, 14-2
- RS232 port pinout table, 14-2
- RTS output, 14-2
- RTS/CTS handshaking protocol, 14-3
- RX input, 14-2
- scanning current for the interface, 14-4
- services, 14-17
- switched 5v regulated output, 14-3
- switched Vsup output, 14-3
- switching off the external unit, 14-2
- tmask serial port characteristics, 14-3
- top slot, 14-2, 14-3
- TTY:A, 14-2
- TTY:B, 14-2
- TTY:D, 14-2, 14-3
- TTY:E, 14-2, 14-3
- TX output, 14-2
- undecoded laser scanners, 14-3
- Vsup input voltage supply range, 14-3
- wand emulator, 14-1
- Xon/Xoff handshaking, 14-2
- HC cradle device
 - introduction, 11-1
 - see cradle device, 11-1
- HC docking station
 - device introduction, 11-1
- HC intelligent bar code reader
 - device introduction, 14-1
 - see HC bar code/RS232 module, 14-1
- HC magnetic card device
 - introduction, 12-1
 - p_close, 12-1
 - P_FCANCEL, 12-2
 - P_FREAD, 12-1
 - P_FSET, 12-2
 - p_open(MCR:), 12-1
 - services, 12-1
- HC magnetic card reader
 - device I/O, 12-1
- HWIM
 - infrared applications, 15-6
- I/O devices
 - introduction, 1-1
 - see devices, 1-1
- I/O function
 - asynchronous, 1-1
 - p_ioc, 1-1
 - p_ioca, 1-1
 - p_iow, 1-1
 - synchronous, 1-1
- IAS
 - infrared, 15-4
- idle mode
 - infrared, 17-4
- include file
 - infrared constants p_file.h, 16-2
- infrared
 - AccessIr API, 15-4, 16-1
 - AIR: device, 15-4, 16-1, 16-2
 - AIR: device on Siena, 16-1
 - baud rates, 15-3
 - beaming communications application, 15-6
 - communications AccessIr API, 16-1
 - communications application, 15-6
 - communications IrMUX API, 17-1
 - Condor serial chip, 15-3
 - connection first time (LM-IAS server), 17-4
 - connection second time (LM-IAS server), 17-4
 - connectionless calls, 17-3
 - connectionless read request, 17-3
 - connectionless services IrLAP, 15-4
 - connectionless write request, 17-3
 - connection-oriented calls, 17-3
 - connection-oriented services IrLAP, 15-4
 - constant P_FIRAWAITCONNECT, 16-3
 - constant P_FIRDISCONNECT, 16-3
 - constant P_FIRDISCOVER, 16-3
 - constant P_FIRMAKECONNECT, 16-3
 - constant P_FIRSELECT, 16-3
 - constant P_FREAD, 16-3
 - constant P_FWRITE, 16-3
 - constants p_file.h, 16-2
 - device driver accessir.1dd, 16-1
 - disconnecting first time (LM-IAS server), 17-4
 - disconnecting second time, 17-4
 - discovery, 15-5, 17-4
 - example application, 16-7
 - exclusive mode, 17-4
 - handshaking level, 17-4
 - HWIM applications, 15-6
 - IAS, 15-4
 - IAS get value by class frame, 17-2, 17-3
 - IAS get value by class reply frame, 17-3
 - idle mode, 17-4
 - IPCS methods, 17-1
 - IrDA protocol model, 15-1
 - IrDA standard, 15-1
 - IrLAP, 15-1
 - IrLAP layer, 15-2, 15-3
 - IrLAP services, 15-4
 - IrLMP, 15-1, 15-4
 - IrLMP layer, 15-3
 - IrMUX API, 15-4, 17-1
 - IrMUX message format, 17-5
 - IrMUX server logoff, 17-2
 - IrMUX server logon, 17-2
 - ISO OSI layers, 15-1
 - link control, 15-6
 - link paste, 15-6
 - LM_AccessModeRequest, 17-4, 17-11
 - LM_CLReadRequest, 17-6
 - LM_CLWriteRequest, 17-7
 - LM_ConnectRequest, 17-2, 17-4, 17-7
 - LM_DisconnectRequest, 17-4, 17-13
 - LM_DiscoverDevicesRequest, 17-4, 17-7
 - LM_GetValueByClass, 17-2
 - LM_IdleRequest, 17-12
 - LM_Logoff, 17-2, 17-5
 - LM_Logon, 17-2, 17-5

- LM_ReadRequest, 17-4, 17-9
- LM_RegisterPort, 17-2, 17-5
- LM_SetHandshakingLevel, 17-4, 17-12
- LM_StatusRequest, 17-9
- LM_UnRegisterPort, 17-2, 17-6
- LM_UReadRequest, 17-4, 17-10
- LM_UWriteRequest, 17-4, 17-11
- LM_WaitForConnection, 17-8
- LM_WriteRequest, 17-4, 17-10
- LM-IAS, 15-4
- LM-IAS server registering, 17-2
- LM-IAS server unregistering, 17-2
- LM-MUX, 15-4
- loadIRDAserver() example code, 16-2
- message number 1 (IrMUX), 17-5
- message number 10 (IrMUX), 17-5
- message number 11 (IrMUX), 17-6
- message number 2 (IrMUX), 17-5
- message number 20 (IrMUX), 17-13
- message number 21 (IrMUX), 17-9
- message number 22 (IrMUX), 17-11
- message number 23 (IrMUX), 17-12
- message number 24 (IrMUX), 17-9
- message number 25 (IrMUX), 17-10
- message number 26 (IrMUX), 17-10
- message number 27 (IrMUX), 17-11
- message number 28 (IrMUX), 17-12
- message number 4 (IrMUX), 17-6
- message number 5 (IrMUX), 17-7
- message number 6 (IrMUX), 17-7
- message number 8 (IrMUX), 17-8
- message number 9 (IrMUX), 17-7
- multiplexing, 15-6
- MUXMESSAGE structure, 17-5
- OSI layers, 15-1
- p_close, 16-2, 16-4
- P_FIRAWAITCONNECT, 16-3, 16-5
- P_FIRDISCONNECT, 16-3, 16-6
- P_FIRDISCOVER, 16-2, 16-4
- P_FIRMAKECONNECT, 16-3, 16-5
- P_FIRSELECT, 16-2, 16-4
- P_FREAD, 16-3, 16-6
- P_FWRITE, 16-3, 16-6
- p_mreceive, 17-5
- p_open(AIR:), 16-2, 16-3
- primary station, 15-4
- printer IRP: device, 15-7
- printer IRP: device Siena, 15-7
- protocol stack - initializing, 17-1
- protocol stack example code, 17-2
- protocol stack SYS\$IRDA.IMG, 16-2, 17-2
- Psion protocol model, 15-2
- read request, 17-4
- read request unreliable, 17-4
- retries, 17-4
- secondary station, 15-4
- SIR, 15-1
- SIR: device, 15-2, 15-3
- SYS_IR_POWER_LEVEL, 15-6
- SYS_PRINTER_IR, 15-6
- unreliable read request, 17-4
- unreliable write request, 17-4
- write request, 17-4
- write request unreliable), 17-4
- infrared communication
 - introduction to Psion IR, 15-1
- IPCS
 - infrared methods, 17-1
- IrDA
 - infrared protocol model, 15-1
 - standard, 15-1
- IrLAP
 - infrared, 15-1
 - infrared layer, 15-2, 15-3
- IrLAP services
 - infrared, 15-4
- IrLMP
 - infrared, 15-1, 15-4
 - infrared layer, 15-3
- IrMUX
 - infrared API, 17-1
- IrMUX API
 - infrared, 15-4
- IrMUX message format
 - infrared, 17-5
- IrMUX server
 - infrared logoff, 17-2
 - infrared logon, 17-2
- IRP:
 - infrared printer device, 15-7
- ISO OSI layers
 - infrared, 15-1
- key code
 - W_KEY_IR_BRING, 15-6
 - W_KEY_IR_LINK, 15-6
 - W_KEY_IR_SEND, 15-6
- Link
 - data transfer, 10-1
 - example code, 10-10
 - inter-process messages, 10-3
 - panics, 10-1
 - process, 10-1
 - protocol, 10-1
 - protocol I/O introduction, 10-1
- LINK
 - process, 10-3
- link control
 - infrared, 15-6
- link paste
 - infrared, 15-6
- LM_AccessModeRequest
 - infrared, 17-4, 17-11
- LM_CLReadRequest
 - infrared, 17-6
- LM_CLWriteRequest
 - infrared, 17-7
- LM_ConnectRequest
 - infrared, 17-2, 17-4, 17-7
- LM_DisconnectRequest
 - infrared, 17-4, 17-13
- LM_DiscoverDevicesRequest
 - infrared, 17-4, 17-7
- LM_GetValueByClass
 - infrared, 17-2

- LM_IdleRequest
 - infrared, 17-12
- LM_Logoff
 - infrared, 17-2, 17-5
- LM_Logon
 - infrared, 17-2, 17-5
- LM_ReadRequest
 - infrared, 17-4, 17-9
- LM_RegisterPort
 - infrared, 17-2, 17-5
- LM_SetHandshakingLevel
 - infrared, 17-4, 17-12
- LM_StatusRequest
 - infrared, 17-4, 17-9
- LM_UnRegisterPort
 - infrared, 17-2, 17-6
- LM_UReadRequest
 - infrared, 17-4, 17-10
- LM_UWriteRequest
 - infrared, 17-4, 17-11
- LM_WaitForConnection
 - infrared, 17-8
- LM_WriteRequest
 - infrared, 17-4, 17-10
- LM-IAS
 - infrared, 15-4
- LM-IAS server
 - infrared registering, 17-2
 - infrared unregistering, 17-2
- LM-MUX
 - infrared, 15-4
- loadIRDAserver()
 - infrared example code, 16-2
- magic static
 - CON: device handle winHandle, 2-2
- magnetic card device
 - see HC magnetic card device, 12-1
- main world file
 - location, 8-9
- maximum scanning rate
 - bar codes, 14-4
- MCR: device
 - introduction, 12-1
 - see HC magnetic card device, 12-1
- multiplexing
 - infrared, 15-6
- MUXMESSAGE structure
 - infrared, 17-5
- NCP
 - panics, 10-1
 - process SYS\$NCP, 10-1
- NCP device
 - I/O introduction, 10-1
- NCP: device
 - example code, 10-10
 - p_close, 10-4
 - P_FCANCEL, 10-7
 - P_FCONNECT, 10-4
 - P_FDISCONNECT, 10-6
 - P_FINQ, 10-9
 - P_FREAD, 10-6
 - P_FRISUPER, 10-7
 - P_FSENSE, 10-10
 - P_FSTOP, 10-10
 - P_FWRITE, 10-7
 - p_open(NCP:), 10-3
 - process SYS\$NCP, 10-2
 - services, 10-3
- note sequence
 - SND: device, 5-3
- OPL
 - CON: device use of, 2-1
- OSI layers
 - infrared, 15-1
- p_close
 - alarm device, 6-2
 - console device, 2-4
 - FRC: device, 7-2
 - HC bar code device, 13-2
 - HC bar code/RS232 module, 14-18
 - HC magnetic card device, 12-1
 - infrared, 16-2, 16-4
 - NCP: device, 10-4
 - parallel port, 3-1
 - serial port, 4-6
 - sound device, 5-2
 - world device, 8-2
 - XYmodem device, 9-8
- p_close(CRD:)
 - close device, 11-2
- p_close(FCG:A)
 - fast charger device, 18-4
- P_EVENT_READ
 - console device, 2-9
- P_EVENT_TEST
 - console device, 2-10
- P_FCANCEL
 - alarm device, 6-2
 - cancel read from device, 11-2
 - console device, 2-5
 - FRC: device, 7-2
 - HC bar code device, 13-2
 - HC magnetic card device, 12-2
 - NCP: device, 10-7
 - parallel port, 3-2
 - serial port, 4-8
 - sound device, 5-2
 - world device, 8-2
- P_FCONNECT
 - NCP: device, 10-4
 - XYmodem device, 9-8
- P_FCTRL
 - serial port, 4-9
- P_FDISCONNECT
 - NCP: device, 10-6
 - XYmodem device, 9-10
- P_FEDIT
 - console device, 2-6
- P_FFLUSH
 - console device, 2-6
 - serial port, 4-9
- p_file.h
 - infrared constants, 16-2

- P_FINQ
 - console device, 2-10
 - NCP: device, 10-9
 - serial port, 4-10
- P_FIRAWAITCONNECT
 - infrared, 16-3, 16-5
 - infrared constant, 16-3
- P_FIRDISCONNECT
 - infrared, 16-3, 16-6
 - infrared constant, 16-3
- P_FIRDISCOVER
 - infrared, 16-2, 16-4
 - infrared constant, 16-3
- P_FIRMAKECONNECT
 - infrared, 16-3, 16-5
 - infrared constant, 16-3
- P_FIRSELECT
 - infrared, 16-2, 16-4
 - infrared constant, 16-3
- P_FREAD
 - console device, 2-5
 - FRC: device, 7-2, 7-3
 - HC bar code device, 13-2
 - HC bar code/RS232 module, 14-18
 - HC magnetic card device, 12-1
 - infrared, 16-3, 16-6
 - infrared constant, 16-3
 - NCP: device, 10-6
 - read from device, 11-2
 - serial port, 4-7
 - XYmodem device, 9-11
- P_FRSUPER
 - NCP: device, 10-7
- P_FSENSE
 - console device, 2-6
 - HC bar code/RS232 module, 14-18
 - NCP: device, 10-10
 - parallel port, 3-2
 - sense the device type, 11-2
 - serial port, 4-9
 - sound device, 5-2
- P_FSET
 - HC bar code/RS232 module, 14-18
 - HC magnetic card device, 12-2
 - parallel port, 3-2
 - serial port, 4-9
 - service call conversion, 2-3
 - set the device type, 11-2
 - sound device, 5-2
- P_FSTART
 - FRC: device, 7-2
- P_FSTOP
 - NCP: device, 10-10
- P_FTEST
 - console device, 2-5
 - serial port, 4-9
- P_FWFLUSH
 - console device, 2-10
- P_FWRITE
 - HC bar code/RS232 module, 14-19
 - infrared, 16-3, 16-6
 - infrared constant, 16-3
 - NCP: device, 10-7
 - parallel port, 3-2
 - serial port, 4-8
 - XYmodem device, 9-12
- p_ioc
 - I/O function, 1-1
- p_ioca
 - I/O function, 1-1
- p_iow
 - I/O function, 1-1
- p_mreceive
 - infrared, 17-5
- p_open(AIR:
 - infrared)
 - infrared, 16-2
- p_open(AIR:)
 - infrared, 16-3
- p_open(ALM:)
 - alarm device, 6-2
- p_open(BAR:)
 - HC bar code device, 13-2
- p_open(CON:)
 - console device, 2-3
- p_open(CRD:)
 - open device, 11-1
- p_open(FCG:A)
 - fast charger device, 18-3
- p_open(FRC:)
 - FRC: device, 7-1
- p_open(MCR:)
 - HC magnetic card device, 12-1
- p_open(NCP:)
 - NCP: device, 10-3
- p_open(PAR:)
 - parallel port, 3-1
- p_open(SND:)
 - sound device, 5-2
- p_open(TTY:)
 - HC bar code/RS232 module, 14-17
 - serial port, 4-6
- p_open(WLD:)
 - world device, 8-2
- p_open(XMD:)
 - XYmodem device, 9-8
- P_SCR_ATTRB
 - console device, 2-11
- P_SCR_CANCEL_CAPTURE_KEY
 - console device, 2-13
- P_SCR_CAPTURE KEY
 - console device, 2-13
- P_SCR_CLIENT_FOREGROUND
 - console device, 2-12
- P_SCR_CLR
 - console device, 2-7
- P_SCR_COMPATIBILITY
 - console device, 2-9
- P_SCR_CSET
 - console device, 2-10
- P_SCR_CURSOR
 - console device, 2-8
- P_SCR_DISABLE_READS
 - console device, 2-12
- P_SCR_ESCAPE
 - console device, 2-9

- P_SCR_FLUSH
 - console device, 2-12
- P_SCR_FONT
 - console device, 2-11
- P_SCR_GREY
 - console device, 2-9
- P_SCR_LAST_LINE_WRAP
 - console device, 2-12
- P_SCR_NEL
 - console device, 2-8
- P_SCR_POSA
 - console device, 2-8
- P_SCR_POSR
 - console device, 2-8
- P_SCR_SCROLL
 - console device, 2-7
- P_SCR_SLOCK
 - console device, 2-8
- P_SCR_WLOCK
 - console device, 2-8
- P_SCR_WSET
 - console device, 2-7
- p_write
 - console device, 2-4
- panics
 - alarm device, 6-2
 - console device, 2-3
 - Link, 10-1
 - NCP, 10-1
 - parallel port device, 3-1
 - serial port device, 4-1
 - sound device, 5-1
- PAR: device
 - example code, 3-3
 - introduction, 3-1
 - see parallel port, 3-1
- parallel port
 - device I/O, 3-1
 - example code, 3-3
 - p_close, 3-1
 - P_FCANCEL, 3-2
 - P_FSENSE, 3-2
 - P_FSET, 3-2
 - P_FWRITE, 3-2
 - p_open(PAR:), 3-1
 - panics, 3-1
- parity errors
 - HC bar code/RS232 module, 14-3
- PC-AT style RS232 port
 - HC bar code/RS232 module, 14-1
- PLIB
 - programs CON: device use of, 2-2
- power consumption
 - HC bar code/RS232 module, 14-4
- primary station
 - infrared, 15-4
- printer device
 - IRP: infrared, 15-7
- protocol
 - Link introduction, 10-1
- protocol stack
 - infrared - initializing, 17-1
- Psion infrared
 - protocol model, 15-2
- read request
 - infrared, 17-4
- read request unreliable
 - infrared, 17-4
- retries
 - infrared, 17-4
- RI pin of the RS232 port
 - HC bar code/RS232 module, 14-2
- RS232 device
 - see serial port, 4-1
- RS232 port pinout table
 - HC bar code/RS232 module, 14-2
- secondary station
 - infrared, 15-4
- serial infrared
 - Condor chip, 15-3
- serial port
 - baud rate, 4-2
 - character frame, 4-2
 - control flags, 4-5
 - device driver LDD, 4-1
 - device driver PDD, 4-1
 - device I/O introduction, 4-1
 - errors, 4-5
 - example code, 4-11
 - handshaking, 4-3
 - HC TTL levels, 4-1
 - p_close, 4-6
 - P_FCANCEL, 4-8
 - P_FCTRL, 4-9
 - P_FFLUSH, 4-9
 - P_FINQ, 4-10
 - P_FREAD, 4-7
 - P_FSENSE, 4-9
 - P_FSET, 4-9
 - P_FTEST, 4-9
 - P_FWRITE, 4-8
 - p_open(TTY:), 4-6
 - panics, 4-1
 - parameters, 4-1
 - parity, 4-3
 - services, 4-6
 - terminator characters, 4-5
- Siena infrared
 - printer IRP: device driver, 15-7
- SIR
 - infrared, 15-1
- SIR:
 - infrared device, 15-2, 15-3
- SND: device
 - example code, 5-5
 - introduction, 5-1
 - see sound device, 5-1
- sound
 - buzzer emulator, 5-1
 - buzzer piezo, 5-1
 - device I/O introduction, 5-1
 - digital files, 5-1
 - HC machines, 5-1
 - MC machines, 5-1

- S3 machines, 5-1
- S3a machines, 5-1
- sound device
 - E_FALARM, 5-3
 - E_FSSOUNDCHANNELn, 5-3
 - example code, 5-5
 - p_close, 5-2
 - P_FCANCEL, 5-2
 - P_FSENSE, 5-2
 - P_FSET, 5-2
 - p_open(SND:), 5-2
 - panics, 5-1
 - services, 5-2
 - services additional HC MC S3a, 5-3
 - services additional S3 S3a, 5-4
 - SND: introduction, 5-1
- structures
 - DISCOVERY_LOG infrared, 15-5
 - E_MESSAGE infrared, 17-5
 - MUXMESSAGE infrared, 17-5
- supplement digits
 - bar codes, 14-4
- synchronous
 - I/O functions, 1-1
- SY\$IRDA.IMG
 - infrared protocol stack, 16-2, 17-2
- SY\$NCP
 - process, 10-1, 10-2
- SYS_IR_POWER_LEVEL
 - infrared, 15-6
- SYS_PRINTER_IR
 - infrared, 15-6
- time application
 - alarm services S3 S3a, 6-2
- top slot
 - HC bar code/RS232 module, 14-2
- TTY: device
 - example code, 4-11
 - introduction, 4-1
 - parameters, 4-1
 - see serial port, 4-1
- TTY:A
 - HC bar code/RS232 module, 14-2
- TTY:B
 - HC bar code/RS232 module, 14-2
- TTY:D
 - HC bar code/RS232 module, 14-2
- TTY:E
 - HC bar code/RS232 module, 14-2
- unreliable read request
 - infrared, 17-4
- unreliable write request
 - infrared, 17-4
- W_KEY_IR_BRING
 - key code, 15-6
- W_KEY_IR_LINK
 - key code, 15-6
- W_KEY_IR_SEND
 - key code, 15-6
- wand emulator
 - HC bar code/RS232 module, 14-1
- winHandle
 - magic static CON: device handle, 2-2
- WLD: device
 - introduction, 8-1
 - see world device, 8-1
- Workabout
 - battery capacity calculation, 18-3
- Workabout docking station
 - device introduction, 11-1
 - see docking station device, 11-1
- world application
 - database S3 S3a, 8-2
- world database
 - contents, 8-1
 - file extension, 8-9
 - file format, 8-9
 - file format extension file, 8-10
 - file types and locations, 8-9
 - see also world device, 8-1
- world database device
 - see world device, 8-1
- world device
 - database contents, 8-1
 - file extension, 8-9
 - file format, 8-9
 - file format extension file, 8-10
 - file types and locations, 8-9
 - I/O introduction, 8-1
 - introduction, 8-1
 - mode, 8-1
 - p_close, 8-2
 - P_FCANCEL, 8-2
 - p_open(WLD:), 8-2
 - services, 8-2
 - WR_BACK, 8-3
 - WR_CALC, 8-8
 - WR_EXTRA, 8-5
 - WR_FIND_CITY, 8-2
 - WR_FIND_COUNTRY, 8-2
 - WR_FIND_EXACT, 8-3
 - WR_GET_CITY_DATA, 8-6
 - WR_GET_COUNTRY_DATA, 8-7
 - WR_GET_DEFAULT_COUNTRY, 8-4
 - WR_GET_DIAL_STRING, 8-4
 - WR_GET_HOME, 8-3
 - WR_NEXT, 8-3
 - WR_NEXT_LOCK, 8-9
 - WR_SET_DEFAULT_COUNTRY, 8-4
 - WR_SET_EXTRA, 8-5
 - WR_SET_HOME, 8-3
- world file
 - extension, 8-9
 - format, 8-9
 - format extension file, 8-10
 - main, 8-9
- world.dat
 - location, 8-9
- WR_BACK
 - world device, 8-3
- WR_CALC
 - world device, 8-8
- WR_EXTRA
 - world device, 8-5
- WR_FIND_CITY
 - world device, 8-2

- WR_FIND_COUNTRY
 - world device, 8-2
- WR_FIND_EXACT
 - world device, 8-3
- WR_GET_CITY_DATA
 - world device, 8-6
- WR_GET_COUNTRY_DATA
 - world device, 8-7
- WR_GET_DEFAULT_COUNTRY
 - world device, 8-4
- WR_GET_DIAL_STRING
 - world device, 8-4
- WR_GET_HOME
 - world device, 8-3
- WR_NEXT
 - world device, 8-3
- WR_NEXT_LOCK
 - world device, 8-9
- WR_SET_DEFAULT_COUNTRY
 - world device, 8-4
- WR_SET_EXTRA
 - world device, 8-5
- WR_SET_HOME
 - world device, 8-3
- write request
 - infrared, 17-4
- write request unreliable
 - infrared), 17-4
- XMD: device
 - introduction, 9-1
 - see XYmodem device, 9-1
- Xmodem device
 - I/O introduction, 9-1
 - see XYmodem device, 9-1
- Xon/Xoff handshaking
 - HC bar code/RS232 module, 14-2
- XYmodem device
 - checksum CRC, 9-1
 - checksum one byte, 9-1
 - device driver, 9-1
 - device I/O, 9-1
 - example connect code, 9-10
 - example disconnect code, 9-10
 - example file receive code, 9-11
 - example file recieve code, 9-12
 - example file send code, 9-10, 9-13
 - p_close, 9-8
 - P_FCONNECT, 9-8
 - P_FDISCONNECT, 9-10
 - P_FREAD, 9-11
 - P_FWRITE, 9-12
 - p_open(XMD:), 9-8
 - protocol problems, 9-7
 - protocols, 9-1
 - services, 9-8
 - Xmodem protocol, 9-2
 - Ymodem protocol, 9-4
- Ymodem device
 - I/O introduction, 9-1
 - see XYmodem device, 9-1