SIBO 'C' Software Development Kit

HC PROGRAMMING GUIDE

Version 2.30

March 1, 1999

Contents

CHAPTER 1

INTRODUCTION TO THE HC

The HC concept

Combining modular hardware design and the most modern software techniques, the Psion HC range of computers represents a new approach to computing in the field. HC computers can extend existing computer networks away from the office, right up to the "front line" - whether that's in a warehouse, on a sales call, on a maintenance visit, or wherever. Rugged and powerful, HC computers are the mobile elements of a computer system, ensuring that information held "at base" in the office is timely and accurate by putting the base directly in touch with the point of action.

The HC has been designed to be integrated into any computer system and to meet any application requirement: assisting with the making of deliveries, taking of orders, collecting or distributing information, servicing equipment, and so on.

Every element of the hardware is configurable, from the plug-in megabyte-sized Solid State Disks, to the internal expansion slots for peripheral devices such as bar code scanners, modems, and magnetic card readers.

Equally important is the multi-tasking operating system with full graphics and windowing capability. Applications can make productive use of the various fonts and emphases available, and can even display and manipulate diagrams, maps, and pictures. The result: software applications that are highly informative and intuitive to use, and which consequently improve operator acceptance and efficiency.

The multi-tasking facilities - unique to the HC range among handheld computers - significantly shorten software development times and greatly simplify otherwise complex issues ranging from the simultaneous monitoring of several peripherals - a bar code scanner and a modem, for example - through to sophisticated process control applications.

Switching on and off

The HC can be switched on or off by means of the ON/OFF key near the top left corner of its front face. Typically, this key is salmon-coloured - though colour configuration is one of many customisation measures possible for the HC.

There is no need to "exit" programs before switching the HC off. When the HC is next switched on, all current programs continue from their previous state. The contents of the internal RAM memory are preserved throughout the period of being switched off, without any significant current being drawn in the meantime.

Switching on for the first time

The first time an HC is switched on or immediately following a reset, it will probably display the *"Insert Pack and press enter"* message.

This indicates that the HC is searching for a configuration file called *autoexec.btf*.

To by-pass this message and hence accept the default configuration, press PSION+ESC (the PSION key has the familiar "cup and saucer" logo: it is usually located near the bottom left of the keyboard). For keyboards that do not have an ESC key, SHIFT+C should be typed instead.

The HC in due course presents a `$` prompt to indicate that its Command Shell is ready to receive commands.

The basic hardware

All aspects of the hardware of the HC have been designed with the following goals in mind:

- portability
- ruggedness
- data security
- ease of use
- adaptability
- long battery life.

Processor

The HC has an industry standard 80C86-compatible 16-bit processor, the NEC V30H, that runs at a clock rate of 3.84MHz.

The HC also contains a number of proprietary-designed custom-built chips called *ASIC*s, which are responsible for many of its more exclusive features. See the *Hardware Reference* manual for more details.

Internal memory

The amount of internal RAM memory on an HC varies from model to model. The basic model, the HC100, has 128k of RAM; the HC110 has 256k, and the HC120 has 512k.

All models have 256k internal Flash ROM. Because the ROM is Flash rather than "OTP" (one-time programmable) or "masked", it is possible for its contents to be altered by special techniques, facilitating additional ROM-based customisation - even down to the level of individual HCs.

Solid state disks (SSDs)

The standard HC has two solid state disk drives, which are the equivalent of disk drives on a PC. To access them, open the rear cover by pressing the catch on the left side of the HC (if the catch is locked, turn it through 180°.) SSDs can be inserted into the disk drives in the top third and the bottom third of the area enclosed by the cover.

SSDs should be inserted with their upper faces (containing large writing) nearest to the rear cover of the HC. If you try to insert them upside down, by accident, you will find they don't fit properly into their slots - so there is no risk of any untoward damage.

The SSD drive near the top of the HC is drive A:, and that near the bottom is drive B:.

SSDs give open-ended capacity for data storage in a highly secure and compact form. SSDs can also be read by other computers in the SIBO range, as well as by PCs equipped with an SSD drive.

The speed of data transfer to and from SSDs is *enormously* faster than with floppy disks, and compares favourably, at 320 kBytes/sec, with even the fastest of hard disks.

There are no moving parts in any SSD, nor in any SSD drive. This is one reason why, notwithstanding the high performance statistics for SSD data transfer, HC batteries last for as long as they do.

Note that you should never open the rear cover of the HC while any SSD is being accessed by the HC. Opening the rear cover switches the machine off immediately, and data loss could occur. In any case of doubt, switch the HC off manually (use the ON/OFF key) before opening the rear cover: this method of powering down the HC is guaranteed not to lose any data between the HC and its SSDs.

Types of SSD

There are two types of SSD: *Flash* and *Ram*. Either type of SSD can be used in either HC drive.

RAM SSDs can be overwritten selectively making them ideal for storing frequently altered information.

RAM SSDs when not pugged into an HC require a backup battery to preserve their data. The battery is a standard miniature lithium cell, with a guaranteed in-use life of one year. It is easily replaced by the user.

A RAM SSD when plugged into an HC will preserve its data *indefinitely*; the one year battery lifetime refers only to periods in which the SSD is *not* plugged into an HC - only then does a RAM SSD draw current from its own battery.

Flash SSDs are a highly secure medium requiring no battery to maintain data integrity. They are ideal for storing data not intended for frequent editing or revision.

When files on a Flash SSD are deleted or modified the original data is simply marked as "inaccessible". The result is rather like crossing out entries in a filofax: the entries still occupy physical space. In due course, the disk may become full up with out-of-date entries. However a Flash SSD can easily be reset to its original pristine state by "formatting" it. If a Flash SSD is full because unwanted "erased" files are still occupying space, but the disk also contains some data still wanted, copy all files to another disk (using eg the `copy *.*` command of the Command Shell). The erased files are ignored by any such `copy` command, so only the data wanted is copied across. The original disk can then be cleared, by formatting it, before relevant files are copied back on to it.

At the time of writing, Flash SSDs are available up to 2 Mbyte in size and Ram SSDs up to 1 Mbyte. By the time you read this, Flash SSDs up to 8 Mbyte in size may be available.

Each SSD has a switch so that the data on it can be write-protected. While an SSD is write-protected:

- nothing it contains can be altered or deleted
- the data held on it can only be read.

The write-protection can be removed by setting the switch back to the 'Write' position.

Expansion modules

There is an expansion port at either end of the HC. These can hold a wide variety of interface devices. Possibilities include:

- RS232/parallel printer port
- barcode reader (complete with wand or CCD/Laser scanner)
- magnetic card reader ("MCR")
- modem
- "combination" devices such as RS232/MCR/scanner.

The two ports are identical, except for their names: "Port A" (at the top end of the HC) and "Port B".

To remove a module from either expansion port, release the rear cover, in the same way as for the SSDs. Slide the release button next to the module to the UNLOCK position and pull the module out. To replace, push the module right in and lock the module into position by pushing the catch into the locked position. The rear cover cannot be closed unless this catch has been set to LOCK.

It is even possible for the contents of an expansion module to be exchanged "in the field". There is no need to reset the HC before doing this.

The Fast Serial port and the Cradle

The Psion Cradle has been designed to satisfy requirements for:

- secure mounting for the HC
- "hands-free" operation
- battery recharge
- high speed data transfer with a PC.

The Cradle incorporates a security lock to ensure that the HC is held reliably. A trigger loaded spring release and hand recess guarantees easy insertion and removal.

There is an additional i/o port, the Fast Serial port, on the right side of the machine, for data exchange and battery charging. It is designed to be connected directly to a Cradle. The high reliability contacts automatically engage when the HC is placed in the Cradle - no user-made connections are required.

Data is exchanged via the Fast Serial port at up to 1.5Mbits/sec.

The Cradle contains an expansion slot provided to accommodate a high-speed connection to a PC. This slot can be used, alternatively, for RS232, MCR, or modem modules (among others). See the chapter *The HC in the Cradle* for more details.

Power supply

The HC can be powered using rechargeable nickel-cadmium batteries or an optional mains adaptor.

The HC will not switch on if there is no power source, if the batteries are too low, or if the rear door is open. Power is needed to operate the HC and to maintain the data stored in internal memory. Data stored on SSDs, however, doesn't rely on the main power source.

On the right side of the machine, under the rubber plug, is a socket labelled POWER. Plug the mains adaptor into this socket. The red power indicator light will come on. This light indicates that the HC is being powered by an external source, such as the mains adaptor - even if the HC itself is not switched on.

The HC is also supplied with a small round lithium battery. This is the backup battery. It is essential because it keeps the internal memory secure if the main batteries are being changed. It should be fitted before the main batteries. However, the HC cannot be run using only the backup battery.

To see where to fit the backup battery, remove the expansion module at the base of the HC. The positive side of the battery should face upwards (towards the rear of the HC).

The backup battery should last for approximately one year, provided the HC doesn't spend long periods with no other power supply. It is recommended that a new backup battery is fitted yearly (if the HC is left powered only by the backup battery, the battery will last for approximately one month).

The main battery cartridge is stored in the back of the HC, between the two SSD drives under the rear cover. It contains the rechargeable batteries. *Do not attempt to disassemble the battery cartridge.*

To remove the cartridge, switch the machine off and release the back cover as for SSDs, then push and lift the cartridge. To fit the battery cartridge back into the HC, slide it into place and close the rear cover; the machine can now be switched on.

The nickel-cadmium batteries can be recharged in several ways:

- leave the sealed cartridge in the HC while powered from the mains - the batteries will be trickle charged
- remove the sealed cartridge from the HC and plug a mains adaptor into it to recharge the batteries directly from the mains
- trickle recharge by a standard Cradle
- fast recharge by a Cradle supporting this facility.

The HC can be configured so that, when either battery is low, a warning message will appear. Independently of this, there are a variety of software methods to monitor the voltages of the batteries.

When the main battery is low, the HC may have enough power to display the screen and accept input from the keyboard, but not enough to write to Flash disk or access expansion devices. The HC will turn off if an operation is attempted for which it does not have enough power. New batteries should be fitted (or the existing batteries recharged) before the operation is tried again.

In order to save power, the HC will by default switch itself off automatically, if left alone for 5 minutes. The "auto-switch-off" time can be changed to another value, if desired, or the HC set so that it does not auto-switch-off at all.

Caution regarding lithium batteries

Note that there is a risk of explosion if lithium batteries are fitted incorrectly. Be sure that the backup battery is fitted so that, if the bottom expansion port is removed, the face of the battery containing the plus symbol is the (partly) visible one. (This is the flatter of the two faces.)

Lithium batteries should be replaced only with the same or equivalent type, as recommended by Psion. Used lithium batteries should be disposed of according to the manufacturer's instructions.

Screen

The normal HC screen is a retardation film LCD 160 pixels wide by 80 pixels deep. In a standard font, this allows for the display of 9 lines each with around 30 characters. If fewer characters are required to be displayed, a larger font can be used, to achieve a more striking screen image.

Changing the font is only one example of the graphics support supplied by the resident software.

By default, the screen is illuminated by reflected light, using (as throughout the HC) state-of-the-art technology. In case additional lighting is required, a variant is available with a factory-fitted backlight. This backlight can be switched on or off whenever the user requires (bearing in mind that there is an inevitable additional drain on the batteries whenever the backlight is used). Alternatively, the HC can be configured to switch off the backlight automatically once a given time period has elapsed.

Keyboard

The keyboard features positive travel dished keys with durable legends.

Various keyboard layouts are available, depending on how the HC is to be used. For example,

- a full alphanumeric keyboard (53 standard-sized keys)
- a more limited, number-oriented keyboard (31 larger keys)
- the alphanumeric keyboard can be augmented with special characters used in Scandinavian countries - these extra characters being accessed via the PSION modifier key
- alternatively, the alphanumeric keyboard can be augmented with special characters used in mainland European countries.

The following special keys may also be present:

ON/OFF	switches the HC on and off
BACKLIGHT	switches the backlight on and off (if one is present)
LCD	controls the contrast of the LCD display
MENU	under application control
TASK	accessed via the SHIFT key: allows for switching between tasks
INFO	accessed via the SHIFT key: under application control (by default, the voltage levels of the main and backup batteries are displayed)
F1 through F4	extra keys under application control
LOCK	forces the keyboard into upper case
DEL	used to edit typing
ESC or C (CLEAR)	used to clear a line of input or cancel an entry
ENTER	terminates a line of input.
PSION	an extra modifier key (analogous to ALT on a PC), recognisable by its familiar "cup and saucer" Psion logo.

The basic software

The software running on an HC at any one time is a mixture of

- ROM resident core software (the "operating system")
- ROM resident utilities, such as the MS-DOS like Command Shell and the Link communications software
- application software, from an SSD or internal memory
- library software, again from an SSD or internal memory.

Library software is software that can be re-used by more than one application. It may be written by Psion, by the application writer, or by a third party.

The effectiveness of library software and application software can be increased considerably by informed use of the ROM resident software - this software sets the HC apart from its competitors just as much as its unique hardware does.

See the following chapter, *Writing Software for the HC*, for some initial guide-lines on how to write applications or library code for the HC.

Versions of the HC software

Whilst the bulk of the material in this manual holds true for HCs with ROM version numbers less than 1.50, parts of the manual presuppose that the ROM software running on the HC has version number at least 1.50.

To see which version of ROM software is contained in any HC, type `ver` at the `$` prompt in the Command Shell. (Alternatively, the version number is displayed following any reset.)

Machines with ROM version numbers less than 1.50 can easily be upgraded, using the *Repro* procedure discussed later in this chapter, in conjunction with a suitable Master SSD.

The terms Epoc and Plib explained

What counts as the operating system of the HC and what counts as an application depends on your point of view. The services in the HC ROM software that applications programmers can call upon actually consist of many layers - as the following few sections make clear.

The kernel of the operating system of the HC is known as *Epoc*. See the *Introduction* chapter of the *Plib Reference* manual for a detailed list of the essential characteristics of Epoc.

Epoc contains code to implement the *Plib* function library - Psion's version of the standard C programming library, as modified and extended for use by HC programs.

The core ROM of the HC contains considerably more than just the Plib library: for example the *Window Server* which is responsible for the screen and the keyboard is a completely separate process the code for which is resident in the ROM.

Hence in response to the `ver` command at the `$` prompt of the Command Shell, versions numbers will be given for the HC ROM version number, the Epoc operating system and the Command Shell.

Initially applications programmers will have little need to distinguish between the various components of the HC operating system. However distinctions do exist and it is necessary to understand them in order to write more advanced applications.

Graphics window server

The Window Server is ultimately responsible for all graphics output on the HC and is also responsible for channelling all keyboard input to the appropriate application(s).

The Window Server includes support for basic text printing functions of the `puts` and `gets` variety - as used in the Command Shell. However, it is expected that most applications will go beyond this level and hence take advantage of at least some of the graphics enhancements supported by the Window Server:

- text display in a variety of fonts and font styles (eg bold, italic), including fonts that are proportional as well as some that are monospaced
- display of characters (or other small icons) in a *custom-designed* application-specific font
- line, box, and poly-line drawing
- area clearing, filling, inverting, and greying
- flashing cursors and other animated displays, including clocks that are automatically updated
- general bitmap and icon manipulation, eg involving maps, markers, and diagrams
- alert dialogs, information status messages, and flashing "busy" indicators.

It is possible to achieve screen displays which update themselves without any annoying flicker, which scroll smoothly, and which redraw quickly whenever required - in marked contrast to some other graphics systems.

See the *Window Server Reference* manual for a complete description of the Window Server.

Multi-tasking kernel

From the beginning Epoc was designed as a pre-emptive multi-tasking operating system. It is multi-tasking in that multiple processes can run concurrently and exchange data dynamically. It is pre-emptive in that a lower priority process is always interrupted when a higher priority process is ready to run. Routine processing can always be *sent into background* if the user has something more urgent to attend to.

Often an application is best written as two or more components each of which implements part of the applications overall functionality. Under Epoc these processes can run concurrently and exchange data as and when required. While one process is sitting idle waiting for an event another process is being run. When the first process receives the event it too starts running immediately with no idle waiting for the second process to complete.

Often the user will require two or more applications to be running at the same time. Under Epoc the user can ask one application to print a large text file and then use another application for editing a second text file. The only restriction is that no more than one application may access a given hardware device at a given time. In the example the first application is accessing the serial (or parallel) port. The second is accessing the screen and the keyboard. Thus there is no conflict.

To switch between applications provided with a user interface the user can press the TASK key. This simply brings the application into the foreground. Applications software may provide additional mechanisms for bringing applications into the foreground.

Writing various programs separately and then giving the user the opportunity to combine them as required - depending on circumstance - naturally adds to the attractiveness of a suite of software. For example the *shell* component of the operating system - the Command Shell which is supplied with the HC - can easily be replaced with a third party shell as long as it is given the appropriate name (*sys$shll.img*) and fulfils a few basic functions. The Window Server may also be replaced although this would be a very complex task and is thus not recommended. Less radically the applications programmer should consider supplying processes that run alongside the in-built ones and which add to the overall functionality of the HC.

Support for asynchronous i/o

A central concept that underlies user friendly interfaces is the idea that the computer should not be held up indefinitely, waiting for an event to complete. For example, the user should always be able to cancel an aberrant data transfer, or a mistaken print request, without having to resort to resetting the computer.

Part of the support Epoc provides for this is its multi-tasking capabilities (see above). Another part is its large range of *asynchronous* i/o services. Rather than just having a request, for example, to print a line (and to wait until the line has indeed been printed), there is a request to print a line *and to notify the program when the line has been printed* leaving the program free to process other data input in the meanwhile.

Another reason why asynchronous services are of fundamental importance is that programs often cannot tell which of two events will be the next one to occur - where the events include not just input from the user, but also a variety of communications data and other peripheral input. Again, a subprocess may report that it has finished some lengthy activity, such as scanning a large database; a supervising program would have to be ready to respond to this notification, as well as being ready for any other kind of data input. Programs ought to be structured to cope with any of these events being the next one to occur.

Traditionally, function libraries offer poor support for asynchronous services. Not so the Plib function library that is built into the HC ROM.

Database support functions

The Plib file i/o functions can be used for any variety of data formats on file, and HC programmers can choose whatever they feel most comfortable with.

However, much can be said in favour of the Dbf file format:

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

For larger or more complicated databases, programmers may consider using the *ISAM* (*Indexed Sequential Access Method*) library that is separately available to support program development. The ISAM routines mesh closely with the Dbf services in Plib. See the *ISAM Reference* manual for more details.

Support for remote file access

In contrast to just supporting remote file *transfer* - a notion familiar to most users of computers - the HC operating system supports the more radical and far-reaching notion of remote file *access*. In many situations, remote file access is altogether the more convenient way for software on one computer to interact with some data stored on another computer.

To clarify the distinction between remote file transfer and remote file access, consider some software on an HC, that from time to time accesses a database stored on a central PC. One method to achieve this would involve the following steps:

- transfer a copy of the database from the PC to the HC
- have the HC software operate on the local copy of the database
- finally transfer the copy of the database back from the HC to the PC.

Two separate pieces of software are involved in this:

- the database application running on the HC
- some communications software, implementing the file transfer.

However, with remote file *access*, the database software on the HC directly accesses the database file on the remote computer. There is no need for some independent software to copy the *whole* database from PC to HC and then, later, back again. Instead, the operating system of the HC automatically transfers only that small part of the data in the database that the software on the HC needs to access.

At one level, the way this works is by an extension of the concept of a filename. Traditionally, the *full* specification of the location of a file on a computer would have been something like

```
a:\project\library\backup.c
```

However, in the view of the HC operating system, this name is actually incomplete (though it suffices for many purposes); strictly, speaking, the full specification of the location of a file on an HC would be something like

```
loc::a:\project\library\backup.c
```

with the leading *loc::* indicating that the file is on the *local* computer. To gain direct access over a file on a remote PC, a filename such as

```
rem::c:\hc\backup.c
```

should be specified - with the leading *rem::* indicating that the file is on the *remote* computer. Given that the computers are connected appropriately, observing the correct naming conventions is all that an application needs to do to gain direct access to files on the remote computer.

In some ways, the remote file access facility of the HC operating system can be compared to the way that networking software provides additional drives on desk top computers. Thus a PC which ordinarily has drives *A:* and *C:* may gain drives *N:* and *U*: when connected to a network - these additional drives allowing access to files stored on the network server or on other computers linked together by the network.

But in another way, the remote file access in Plib is considerably more general; this is why the additional drives appear *as another filing system*. The point is that access is permitted not only to a PC connected to the HC, but also to one of many other types of computers, such as Apple Macs.

For example, to specify a file on an Apple Mac connected to an HC, the following filename might be given:

```
rem::hd40:mike's folder:november:results
```

where it should be noted that the form of the filename is quite different from that allowed by MS-DOS (eg containing spaces and having more than eight letters in a directory name).

For more details, see the section below on *Connecting to other computers*.

Other ROM-based library services

In order to fully appreciate the Plib library it is necessary to read the documentation in the *Plib Reference* manual.

Features worth noting include:

- a full range of mathematical and scientific functions.
- file management and filename manipulation functions.
- support for reading and writing environment variables.
- support for dynamic memory allocation inside and outside the native data segment of an application.
- support for absolute and relative timers. An application could for example switch on an HC and perform a preassigned task at a preset time.
- control over the HC system set-up. Thus an application could for example set the auto-switch-off time, change the language used, and adjust the LCD contrast and backlight setting.
- sophisticated support for error handling.
- special support for advanced object oriented programming methods.

Other ROM components

Additional files in the ROM include

custom$.dat	a specially customisable file that can be written to once and once only
sys$shll.img	the Command Shell, as described in detail in a separate chapter
sys$ntfy.img	the basic Notifier process, used by default to report error conditions (such as missing SSDs)
sys$ctry.cfo	the location of all the language-dependent text strings used by the operating system, as well as keyboard layout information
opl.dyl	allows programs written in Opl to be run (see the *Opl Development Kit* for more information)
olib.dyl	provides additional services that object oriented programmers can access (see the *Olib Reference* manual)
batchk.img	displays information about battery voltage levels
pprint.img	prints a specified file via a nominated peripheral (likely to be omitted from future versions of the ROM)
ttest.img	tests the status of the serial port (likely to be omitted from future versions of the ROM).

Additionally, the ROM contains a variety of programs and device drivers to facilitate communication with other computers - be they PCs, Macs, computers in the SIBO range, or whatever. Chief amongst these is the Link program, described in more detail later in this chapter.

Finally, the ROM also contains a number of built-in fonts (**.fon* files).

Footnote: to obtain a listing of all the files in the ROM, type `dir /p rom::` at the `$` prompt of the Command Shell. Note that no file corresponding to Epoc itself appears in this listing. Epoc is the kernel of the operating system, and not a file in *rom::*.

Customising an HC

This section describes some of the many ways an HC can be customised, to make it ideally suited to some particular set of needs.

Hardware customisation

The HC can be customised to suit customer requirements.

Simple examples of hardware customisation include changing the labels and branding, changing the colour scheme and replacing the keyboard legends.

More complex examples of hardware customisation include changes in the keyboard layout, changes in the size of the LCD, and changes in the assembly of the LCD allowing operation in more extreme ranges of temperature (the cold of the arctic for example).

Further details of hardware customisation are beyond the scope of this manual, which focuses mainly on *software customisation.*

Replacing the built-in Shell

The Window Server will when the HC is first switched on (or following a reset) search for the *shell* program stored in the file *sys$shll.img*. The search is carried out as follows:

- first in the root of *a:*
- then in the root of *b:*
- then in the root of *m:*
- finally in the ROM.

On finding the shell program the Window Server will start it running.

Unless specially customised to the contrary, the Window Server also looks for a program of this name, along the same path, whenever the shell terminates (either normally or abnormally) - so as never to leave the HC without a shell running on it.

The importance of this is that the shell program in the ROM can be over-ridden by one on an SSD. HCs *in the field* will typically be running a shell from an SSD, rather than that from the ROM.

The ROM shell is more suited to *development* work, supporting a rich variety of file management, task management, system configuration, and batch file processing commands. However, this functionality brings its own cost in RAM consumption that may well be undesirable for HCs running application software.

By and large, applications writers will most of the time use an alternative shell, switching back to the built-in Command Shell only when the need arises during program development.

Switching from the Command Shell to a custom shell on an SSD involves

- inserting the SSD containing the custom shell
- tasking to the Command Shell
- typing `term sys$shll` (`term` is short for `terminate`).

Switching back to the Command Shell from a custom shell involves

- removing the SSD on which the custom shell resides
- terminating the shell, either by a command supported by the custom shell, or by resetting the HC
- the Window Server, in restarting the shell, will no longer find the custom shell, and hence will start the Command Shell from the ROM instead.

One reason why, even during development work, the Command Shell may not be required, is that most of the basic functionality the Command Shell provides can be duplicated by commands transmitted down a serial connection from the PC to the HC. These commands can be invoked either using the SIBO Debugger, or using MCLink.

A program developed as an alternative shell would typically have another name during development, such as *hcshell.img*. It would be renamed to *sys$shll.img* only at the last minute. Otherwise, other SIBO programs, such as the SIBO Debugger, might fail to work, on account of finding this alternative shell and attempting to run it instead of the appropriate shell that is built into their own ROM.

Resetting the HC

It should rarely be necessary to reset the HC. Even if, during development, an application contains some dreadful bug, this is most unlikely to cause the entire HC system to hang.

For example, any illegal attempt by an application to write to data outside its own data segment will lead to the operating system terminating the application forthwith, in a so-called *panic*. Likewise should an application leave interrupts disabled for too long.

However, the worst may come to the worst and a reset may prove necessary. Alternatively, it may be required to reset the HC, just in order to terminate one shell process and to cause a new one (say that in the ROM) to be started instead.

Before resetting, it is wise to first terminate all applications and save any important data to an SSD or a PC.

To carry out a *soft* reset of an HC, insert the end of an opened paper-clip into the reset hole (located just to the right of the microphone). This will re-boot the HC forcing the abandonment of all programs running at the time and the consequent loss of the associated data. The files in the internal memory (*m:*) will *not* be lost.

To carry out a hard reset hold the ON/OFF key down while pressing the paper clip into the reset hole. This will erase all internal memory including environment variables.

Reproing the HC

For some applications, an alternative mix of files on the ROM may be required for special customisation purposes:

- programs run out of ROM have less of a RAM overhead than those run from an SSD.
- programs in the ROM are physically more secure than those on an SSD, in the sense that an SSD can be removed by a user but the ROM cannot.
- programs in the ROM may be able to take advantage of special software features inaccessible to programs on an SSD - for example, the fact that ROM code and data segments always remain at a fixed address.
- programs in the ROM are easier to copy protect.

All the different files comprising an HC ROM need to be assembled on a PC, and then combined into a special *master* file, with extension *.mas*. This process is described in the chapter *Customising the HC Rom*, later in this manual.

The process of transferring a *.mas* file into the ROM of an HC is called *reprogramming*, or *reproing* for short. Reproing can be used, not only to produce a specially customised version of the ROM software, but also to upgrade an earlier ROM to a more recent one (say to ROM version 1.50).

Reproing requires a *master SSD*, which contains both the *.mas* file and the repro software itself. Note that, counter-intuitively, reproing will not work if the master SSD is write-protected.

During reproing, the HC should be powered from the mains. As a precaution, it is wise also to have a charged battery in the HC: if the power fails during reprogramming, the HC will need to be sent back to Psion before it works again.

The master SSD should be placed in either of the SSD drives and the rear cover of the HC closed. Type `repro` followed by ENTER at the `$` prompt of the Command Shell. Once the HC has displayed the new master details, press ENTER again to confirm reprogramming.

On HCs with the 31 key numeric keyboard, it is impossible to type `repro`, so type the following instead:

YES 0 NO 0 ENTER

The message `Y0N0` will appear on the screen, and then *repro* will run as normal.

During reproing, the HC displays large characters on the top line of the screen, starting with `A000`. In all, four rows of figures will eventually be displayed. On completion, the HC will emit beeps, and an automatic reset should occur. Occasionally, the automatic reset may fail to occur, in which case a hard reset should be executed (as described above).

As usual following a reset, the automatic search that takes place in these circumstances for a file *autoexec.btf* can be circumvented by pressing PSION+ESC (or, on keyboards with no ESC key, SHIFT+C).

Once reprogramming has completed, the new ROM version number can be determined using the command `ver` in the Command Shell.

Note that (in contrast with the case of reproing the laptop MC computers) no additional hardware *repro enabler* is required in order to repro an HC.

Master SSDs and mastcpy

The master SSDs used during reprogramming cannot be duplicated using ordinary software (such as the `copy` command in the Command Shell). More precisely, only *part* of the contents of a master SSD can be copied in this way.

However, a special tool is available, called `mastcpy`, which *can* make a copy of a master SSD.

The `mastcpy` program runs on a PC with an external SSD drive.

Once-off ROM customisation using Romwrite

As an alternative to reproing an HC with a specially customised ROM, it is possible to customise it by overwriting, in a special way, the contents of the file *custom$.dat* that is in the ROM. Typical uses of this mechanism include

- adding special information such as serial numbers or details of the owner.
- loading an alternative set of *language text*, containing versions of such operating system messages as "Low main battery" and "No system memory" in a foreign language.

In order to write to this file, the special tool *romwrite.img* (available as part of the SDK) has to be used.

Note that *romwrite* can only be used with HC ROMs with version number 1.50 and above. Mains must be present for *romwrite* to operate.

Romwrite copies the contents of a file supplied by the user, with the name *custom$.ref*, into the ROM file *custom$.dat*. The file *custom$.def* must be placed in the same directory as *romwrite.img*. A maximum size of 4608 bytes is allowed for *custom$.ref*. If this file is larger than 4608 bytes, only the first 4608 bytes will be copied into ROM, and no error will be reported.

Romwrite appends two further bytes to the end of the copied information. These are required checksum information.

To invoke *romwrite*:

- copy *romwrite.img* and a suitable file *custom$.ref* to an SSD.
- place the SSD in the HC.
- connect a mains adaptor to the HC.
- type `romwrite` at the `$` prompt of the Command Shell.

An error message will be given if the file *custom$.dat* in the ROM has already been written to, or if there were any problems in writing to the internal ROM.

Warning: if a write error occurs during *romwrite*, the HC must be reproed from a master SSD before being used any further. *Do **not** reset the HC.*

Once the contents of *custom$.dat* have been overwritten using *romwrite*, they cannot be overwritten again until the ROM has been reproed.

Reproing entirely loses the contents of this customised file. However, the contents are unaffected by any reset, *even a hard reset*.

Customisation for copy-protection

One additional use of the file *custom$.dat* would be to frustrate illicit copying of software (see also the chapter *Copy-Protecting Software* in the *General Programming Manual* for discussion of alternative methods with the same end).

Briefly, when an application is started, it could read the contents of *custom$.dat*, looking for a pre-defined byte-stream signature. If this signature is not present, the application would refuse to run.

The signature would have to be written beforehand, into *custom$.dat*, by means of a special installation program. Possibly, the software company producing the application would make a special charge to administer the installation program (whose details would need to be kept secret).

Connecting to other computers

Connections between an HC and another computer, such as a PC or Mac, can be divided into two sorts:

- *high speed connections*, which require the HC to be located in a Cradle and which generally also require the PC to be fitted with an ASIC-2 expansion card
- *standard connections*, which simply require a standard serial cable between the HC and the other computer (no expansion card is required in this case).

High speed connections are discussed more fully in the chapter *The HC in the Cradle*. The remainder of this section focuses primarily on standard connections. See also the chapter *Mclink, Mcprint, and Slink* in the *Additional System Information* manual.

Basics of serial connections to an HC

Any connection between two computers involves a *hardware connection* and a *software connection*.

When an HC is connected to another computer, the software connection will generally be via Epoc *Link* software. A version of this software has to be running on each of the two computers.

The Link software can be started on the HC simply by typing `link` into the Command Shell. One way to start it on a PC is to invoke the executable *mclink.exe* (similar programs also exist for other types of computer, such as Apple Macs).

The hardware connection between a PC and an HC, when Link software is running, can involve either a custom RS232 cable plugged into the PC at one end and the HC at the other, or a High Speed Serial connection via an HC Cradle.

RS232 connections

Modern PCs have 9-pin sockets on serial ports; older ones have 25-pin sockets. If you only have one serial port on your PC, it is called *COM1*, although it is common for PCs to have a second serial port called *COM2* (*COM3* and even *COM4* are also possible, but the Link software does not support these).

Connect the appropriate socket at the PC end of your cable to COM1 if is available - otherwise, use COM2. Link software on the PC sees COM1 as `TTY:A` and COM2 as `TTY:B`. Alternatively, these can also be referenced simply as "p1" and "p2" (for *port*s 1 and 2).

To specify that MCLink uses port 1, type

```
mclink -p1
```

at the MS-DOS command line. Likewise type `mclink -p2` to specify port 2.

The socket at the HC end of the cable plugs straight into the serial port in expansion modules of the HC.

Summary of straightforward usage of Link on the HC

The Link software on the HC can be started by typing simply `link` at the Command Shell `$` prompt.

To terminate the Link software at some later date, type `term link`.

To discover whether or not Link software is running, type `lproc link`.

If the `link` command is issued while Link is already running, a second copy of Link will be launched briefly, but will quickly exit with the error number -32 (or 224), meaning that a process *link.** already exists. No harm will ensue as a result.

Link allows a HC to open or save files on a remote computer in the same way as it opens and saves files on its internal memory and SSDs. Conversely MCLink allows a remote computer to open and save files on an HC in the same manner. Note that *all* HC applications automatically possess the ability to access remote files in this way - no special "comms software" has to be added into the applications. All that is required is a degree of agnosticism regarding the structure of filenames: eg it must not be assumed that directory names end in '\' characters, nor that the core parts of filenames are restricted to eight letters in length. Provided appropriate Plib library routines are used to manipulate ("parse") filenames, remote file access comes free.

The user should note that the Link software must be left running all the time that files on the other computer are being accessed.

Why not MS-DOS?

Some would-be HC applications developers may be put off by the fact that the operating system of the HC is not MS-DOS but Epoc. On the face of things, this poses two problems:

- applications written presupposing MS-DOS have to be rewritten before working on the HC
- there is a learning curve that has to be negotiated, in coming to terms with the differences between Epoc and MS-DOS.

With regard to the first point, there is, frankly, no way standard MS-DOS applications can transfer over to the smaller screen of a handheld computer without *some* amount of re-writing. The reduced screen size of hand held computers actually means more than just "compressing" the screen display from say 80 columns to around 30; it means having to rethink some of the user interface completely (as many displays simply won't work in their original form, if they are compressed by such factors); the *quantitative* change in screen size is such that it in turns leads to a *qualitative* change in the user interface supported.

However, this consideration is incidental to the main point, which is that Epoc is simply an operating system far better suited to the particular needs of computers such as the HC.

Some of the special advantages of Epoc over any version of MS-DOS are:

- a much more sophisticated power distribution system can be managed, resulting in significantly longer battery lifes than could ever be achieved under MS-DOS
- pre-emptive multi-tasking is natural to Epoc, but is artificial (and hence expensive) to MS-DOS
- Epoc supports remote file access in a way that, again, is expensive to emulate in MS-DOS
- Epoc implements address trapping (on an 8086 chip!), amongst other measures, to prevent aberrant processes from causing a system crash: just consider how many times PC developers have to recourse to the "big red switch" when an aberrant MS-DOS application results in fatal damage to PC RAM contents, and compare this with how few times a corresponding measure is required during HC development
- Epoc allows a change in which device drivers are loaded, without the computer having to be reset.

Briefly, Epoc results in smaller programs which execute more efficiently and in a manner more in line with the intuitive expectations of end users.

This applies for the programs built into the ROM as well as those developers might write. As a result (and this may well be the bottom line), HCs end up considerably cheaper than any corresponding MS-DOS computer.

What actually lies behind the initial hesitation of many would-be HC developers is concern over the extent to which files written by MS-DOS programs on PCs can be read and updated by Epoc programs on an HC. Understandably developers are unwilling to upset an existing successful PC setup, even if they are prepared to learn a new programming system for the HC parts of the overall computer system.

However, developers can rest assured that there is no inherent difference in file structure between MS-DOS programs and Epoc programs. Epoc is fully *file-compatible* with MS-DOS.

Furthermore, it should be re-emphasised that many existing programs *will* transfer fairly smoothly from an MS-DOS environment to an Epoc environment. This is the role of the Clib library, discussed in more detail in the *General Programming Manual*.

Finally, bear in mind what some experienced HC developers have said: that it is actually *quicker* to develop programs for the HC than it is for the PC. In part, this is due to the rich Software Development Kit (with high-powered libraries) available for the HC. But it is also in part due to the fact that Epoc is for many purposes a superior operating system. Accordingly, the Epoc learning curve is one that is well worth climbing!

CHAPTER 2

WRITING SOFTWARE FOR THE HC

Basic programming choices

Choice of programming language

The two main high-level programming languages for the HC are Opl and C.

Whilst Opl has many points in its own favour for smaller projects (discussed in the *Opl Development Kit*), the following points are likely to sway any competent programmer to use C for any more substantial application on the HC:

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

Occasionally, some code may have to be written in assembly language (for example, when writing a device driver).

Standard C (Clib) or Psion C (Plib)

A significant proportion of C code that companies have already written for other target computers can be transferred almost straightaway to run on an HC. All that is necessary to do is to recompile and re-link the code.

To take a very simple example, the program *simple.c*

```
#include <stdio.h>

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

together with a project file *simple.pr*

```
#system epoc img
#model small jpi
#compile simple.c
#link simple
```

will run on an HC without any difficulty whatsoever (see the chapter *Building an Application* in the *General Programming Manual* for further discussion of TopSpeed *.pr* project files and their usage).

However, it is recommended that HC programmers rewrite the above program as follows:

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

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

with the project file changed to

```
#system epoc img
#set epocinit=iplib
#model small jpi
#compile simple.c
#link simple
```

The latter is said to be the "Plib" version of the former, which is a "Clib" program (the "P" of "Plib" stands for "Psion").

The following differences will be noticed between the two programs:

- the Plib program uses Psion-proprietary header files.
- the Plib program uses Psion-proprietary function calls (`p_xxx` functions).
- the Plib program links with a different library (this is the significance of the `epocinit` line in the project file).

Code written with Plib calls is considerably more compact than code written with Clib calls. For example the image file for the example Plib program (see above) has size 576 bytes compared with the image file for the equivalent Clib program that has size 4480 bytes - an increase in size of almost seven hundred per cent.

The reason for the greater compactness of compiled Plib code is that Plib functions provide only *very thin shells* for functionality already present in the HC's ROM. Thus Plib calls make more efficient use of the HC ROM software than do the equivalent Clib calls. Being tailored to the particular needs of computers like the HC, Plib evolved with very different constraints and objectives from standard C libraries. In many cases, Plib functions can be claimed to "improve" upon the specification of their nearest Clib equivalents.

The use of Plib calls does not always lead to such large space savings as seen in the example programs (see above) - the reduction in the size of the compiled code depends on the number and types of library function calls made.

Sometimes it will be desirable to write an application using both Clib and Plib calls simply because this can ease the process of converting large programs to run on the Sibosdk system. The reduced development time will thus outweigh the disadvantages of using the Clib calls.

However it is recommended that an application use the Plib library for at least some of its function calls. Although it takes time to become familiar with the Plib library this will repay itself in the form of more compact and powerful applications. Furthermore use of Plib functions is essential for accessing many features of the Sibosdk ROM software - the enhanced graphics facilities of the Window Server for example.

Writing the user interface

A SIBO interface can be written in one of the following ways:

- using console service functions such as `p_printf`, `p_getl`, and `p_puts` or their Clib equivalents. These functions can only produce simple graphical output. They can be extremely useful when debugging an application.
- using functions in the Window Server library with the contents of each window backed up with a bitmap. This method is capable of producing a high quality graphical display.
- using functions in the Window Server library with the contents of each window explicitly redrawn. This method is capable of creating a high quality graphical display. Use of window redraws is more efficient than use of bitmap backups.

The applications programmer does not have to learn to write applications that use the window redrawing mechanism: for many applications backing up the window with a bitmap is sufficient (the penalties of windows with backup bitmaps are much less on the HC screen than on the larger screens of some of the other SIBO computers).

The applications programmer who subsequently goes on to learn about window redrawing will not have wasted his/her time learning about window bitmap backups. The latter provide an excellent foundation for the more complex concepts behind window redrawing.

The best way to learn graphics programming on the HC is probably to follow the example programs at the end of this chapter and then extend and modify their function. For example one of the example programs illustrates the use of the `wInfoMsg` and `wSetBusyMsg` functions. These powerful graphics functions display an information message and a flashing busy message respectively at the bottom right corner of the screen. They are hardly more difficult to use than simple console functions such as `p_printf` and `p_puts`. Working out how this program and the others work will help to familiarise you with the more commonly used Window Server calls.

The example programs and the discussion in this chapter should provide the would-be HC applications programmer with a sufficiently sound base to enable him/her to make effective use of the *Window Server Reference* manual.

Synchronous or asynchronous processing

There is a class of programs in which all input to a program comes via the keyboard. These programs can be schematised as follows:

```
Initialise();
FOREVER
    {
    ReadKeyFromKeyboard();
    ProcessKey();
    }
```

The program terminates in response to a certain pre-defined key. Whilst waiting for a key from the keyboard, the program "hangs", i.e. it is unresponsive to other sources of input. In this case the hanging of the program does not matter as there are no other sources of input.

The call `ReadKeyFromKeyboard` makes what is known as a *synchronous* read for a key; it is synchronous becomes it does not return until the key it is waiting for has been delivered: the return of the call making the request is automatically *synchronised* with the delivery of the key.

Consider another example of *synchronous* i/o. In this case, a program that is printing data might be structured (at least in part) as follows:

```
Initialise();
FOREVER
    {
    PrepareLineToPrint();
    SendLineToPrinter();
    }
```

This program loop terminates when there is no more data to print. Now the process of sending a line of data to the printer might take some time. The printer buffer could be full in which case the program would have to wait for the buffer to empty a bit before being able to prepare the next line for printing. Thus the call `SendLineToPrinter` could be synchronous (this is the way beginner programmers would tend to write the code), with the program "hanging" in the call until the printer has removed the data passed to it by the program. In this state, the program is, again, unresponsive to other sources of input.

In either of the above examples, a simple extension of the code would require the *synchronous* call to become *asynchronous*. The printing program could and should be extended to allow the user to terminate the printing while in progress by simply pressing a predefined key. The key-processing program could be extended so as to respond to a timer expiring (for example a signal to commence a backup procedure).

Many programmers approach this kind of generalisation in an ad hoc manner resulting in spaghetti like code that is hard to debug, hard to maintain and hard to extend.

Such code will usually force the user to wait while it is waiting for one or more events. The user can thus be shut out for significant periods of time.

The software on the HC has been explicitly designed to address these issues. For all but the simplest of programs the concept of *asynchronous* events is central to successful programming on the HC: would-be applications writers are strongly urged to face up to this issue squarely, from the beginning.

This may sound daunting (and it probably *would* be daunting, on alternative software platforms), but for two reasons, it is not:

- the HC operating system software has carefully isolated the various components involved in asynchronous i/o: signals, semaphores, "status words", and "active words" (amongst others)
- example programs in the *Fundamental Programming Guidelines* chapter of the *General Programming Manual* survey these components in a thorough yet straightforward manner.

Example programs

There are example programs scattered throughout the length and breadth of the SDK. It is recommended that, whenever possible, would-be HC applications developers should take the time to try out these examples, and to modify them. As in all fields, practice makes perfect - and it is always possible to get an idea from the detail of one of these programs, which will prove helpful in a quite different coding situation.

The three programs to be discussed in this chapter have particular relevance to the HC. They demonstrate its graphics potential, and show how to create line editors to allow convenient data entry by end users of the HC (whereas Series3 and Series3a programmers can use the Hwif library to obtain easy access to line editors and other related user interface objects, there is at the time of writing no corresponding library for the HC - so programmers have to take care of the user interface by themselves).

These examples build on those discussed in the *General Programming Manual*, and it is suggested that any readers who have not yet worked through that manual carefully should do so now, before proceeding any further.

In contrast with the examples in the *General Programming Manual*, which only use console i/o, the example programs in this chapter all interact more directly with the Window Server.

The source code for all these examples is located in *\sibosdk\demo*. Incidentally, these programs can also be made to run, with minor modifications, on Series3 and Series 3a machines.

A graphics version of Hello World

The first example is a short program stored as *w_hello.c*:

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

GLDEF_C INT main(VOID)
    {
    WS_EV event;

    wStartup();
    gBorder(W_BORD_CORNER_4);
    wSetBusyMsg("Hello world",W_CORNER_BOTTOM_LEFT);
    do
        {
        wGetEventWait(&event);
        } while (event.type!=WM_KEY || event.p.key.keycode!=W_KEY_ESCAPE);
    return(0);
    }
```

The call `wStartup` takes care of routine preparation to interact with the Window Server (see the *Window Server Reference* manual for more details of all of these calls).

The call `gBorder` draws a pleasant curved border around the edge of the screen. Vary the flags passed to `gBorder` for different types of curves.

The call `wSetBusyMsg` displays the specified message flashing, at the nominated corner of the screen. In general, the message will continue to flash, without any assistance from the application, until such time as a call such as `wCancelBusyMsg` is made.

The call `wGetEventWait` is a *synchronous* request to receive an event from the Window Server. These events include notification of coming into foreground or background, as well as keypresses and requests to redraw portions of the screen (these latter events are used by applications that explicitly handle window redraws - such applications do not use the wStartup function and instead use the lower level function).

As `wGetEventWait` is synchronous, it does not return until there is an event for the application to process. In this example, the application is uninterested in any events other than keypresses, and even then, only the ESC keypress is of interest.

In order to build *w_hello*, simply type `make w_hello` when in the appropriate source directory (*\sibosdk\demo*).

The Gauge application

The *Gauge* application is altogether more sophisticated than *w_hello*:

- the screen display contains text in various font styles.
- the screen also contains a "growing scrollbar" or "petrol gauge" display item, whose content grows regularly, as a timer beats.
- the speed at which the timer beats can be adjusted by keypresses from the user.
- the user can also reset the gauge display at will.
- the range of options open to the user is displayed on a range of "buttons", which momentarily highlight whenever they are selected.
- in programming terms, a timer channel is created as a second event source.
- the synchronous `wGetEventWait` call is replaced by the asynchronous version `wGetEvent`.

The schematic form of `main` in *gauge.c* is as follows:

```
GLDEF_C VOID main(VOID)
    {
    WS_EV event;
    WORD wactive;

    wStartup();
    INITIALISE();
    QueueTimer();
    wactive=FALSE;
    FOREVER
        {
        if (wactive)
            wFlush();
        else
            {
            wGetEvent(&event);
            wactive=TRUE;
            }
        p_iowait();
        if (event.type==E_FILE_PENDING)
            {
            PROCESS_TIMER_EVENT();
            QueueTimer();
            continue;
            }
        wactive=FALSE;
        if (event.type==WM_KEY)
            {
            switch (event.p.key.keycode)
                {
                ....
                }
            }
        }
    }
```

The use of a little imagination will make it clear that this is the same basic architecture (albeit rearranged) as in the *Events* programs discussed in the *General Programming Manual*:

- the variable `wactive` is the active word for the Window Server event source
- the status word for the Window Server event source is built into the `WS_EV` struct passed to the call `wGetEvent`: it is the `event.type` field
- there is no test on the timer status word, `timstat`, since if the call to `p_iowait` has returned and `event.type` is still equal to `E_FILE_PENDING`, it can only be the timer which has an event to deliver (given that there are only two event sources in the application).

The need to flush the Window Server buffer

Note the special test on `wactive` at the top of the event loop in `main`. If `wactive` is still `TRUE`, it means there is no need to call `wGetEvent` again (and in fact the application would be panicked if it did so). However, it is necessary, in this case, to call `wFlush`, to ensure that the Window Server function buffer is flushed out. Otherwise drawing calls could remain in this buffer all the time that the application is suspended, inside `p_iowait`.

The point here is that, for efficiency (minimising IPC - InterProcess Communication - traffic between the application and the Window Server), many Window Server functions are not implemented immediately: rather, they are stored in a buffer which is only "flushed" every so often. See the *Window Server Reference* manual for full details.

Another instance in the *Gauge* application where `wFlush` is called is in the routine `Flash`, in which a highlight is momentarily displayed over a "button" containing the choice the user has just selected:

```
{
P_EXTENT ext;

...
gInvObloid(&ext);
wFlush();
p_sleep(2);
gInvObloid(&ext);
}
```

Other graphics calls in Gauge

The contents of *gauge.c* can usefully be studied (eg use the SIBO Debugger while the program is running) for examples of the following graphics function calls:

`gPrintBoxText`	useful for "flicker free" drawing of text.
`gSetGC`	allows a change in the font or font style (and more besides) used to draw text.
`gClrRect`	clears or highlights a given rectangle.
`gFillPattern`	applies a pattern (here, a "grey" pattern) to an area.
`gTextWidth`	calculates the width of a string of text.
`gInvObloid`	allows special "rounded" or "obloid-shaped" inverse videoing.
`gBorderRect`	draws any of a variety of curves around the edge of a specified rectangle.

A suite of line editor functions

The application *LinEd* demonstrates the use of a suite of line editor functions: three line editors are created on the screen, each with text that the user can edit. The user chooses which entry to edit at any one time by using the UP and DOWN cursor keys. Other editing keys have the expected effects on the editors:

- typing printable characters enters these characters into the current string (with any existing highlighted selection in the string being deleted).
- the editor beeps if it has already grown to its maximum size.
- the editor scrolls horizontally if there are more characters to display than can fit in the width allocated to it on the screen.

- the DEL key deletes the character to the left of the cursor, whereas SHIFT+DEL deletes the character to the right of the cursor, PSION+DEL deletes to the end of the line.
- PSION+LEFT and PSION+RIGHT "home" and "end" the cursor, respectively, LEFT and RIGHT just move the cursor one position.

The suite of "lined" (line editor) functions should be independently useful, either in their present form, or modified for particular purposes (the lined functions are as they stand fairly general). From a broader perspective, the lined functions demonstrate the creation of a user interface for applications on the HC.

The code in *lined.c* divides into two parts: the implementation of the lined functions, and the testing of these functions. The `main` routine of the test program is worth considering in full:

```
GLDEF_C VOID main(VOID)
    {
    LINED *ed[3];
    INT which;
    WS_EV event;
    INT keycode;

    wStartup();
    gBorder(W_BORD_CORNER_4);
    ed[0]=CreateLined(10,"One",TRUE);
    ed[1]=CreateLined(30,"Two",FALSE);
    ed[2]=CreateLined(50,"Three",FALSE);
    which=0;
    FOREVER
        {
        do
            {
            wGetEventWait(&event);
            } while (event.type!=WM_KEY);
        keycode=event.p.key.keycode&(~W_SPECIAL_KEY);
        switch (keycode)
            {
        case W_KEY_ESCAPE:
            if (event.p.key.modifiers==W_PSION_MODIFIER)
                p_exit(0);
        case W_KEY_UP:
            if (which)
                {
                le_emphasise(ed[which--],FALSE);
                le_emphasise(ed[which],TRUE);
                }
            break;
        case W_KEY_DOWN:
            if (which<2)
                {
                le_emphasise(ed[which++],FALSE);
                le_emphasise(ed[which],TRUE);
                }
            break;
        default:
            le_key(ed[which],keycode,event.p.key.modifiers);
            }
        }
    }
```

The array of three pointers `ed[3]` is used to hold the "handles" of the three lined objects created. This creation is done inside the call `CreateLined` (further discussed below). At any one time, only one of these three editors is "active" - displaying a flashing cursor and receiving editing keys from the user. The application uses the variable `which` to keep track of the current active editor.

On receipt of an UP or DOWN key, the application changes its record of which editor is active. At the same time, the editors themselves have to be informed of this change - so that they can adjust their appearance. This is the role of the calls to `le_emphasise`.

All other keys (apart from PSION+ESC, which exits the application) are passed straight through to the current editor, using the call `le_key`.

Full specification of the lined functions

The routine `le_init` creates and initialises a lined object, according to the data in an `IN_LINED` struct passed. This creation involves two separate allocator calls - one for the control block of the editor itself, and one for the buffer to hold the string of text to be edited. Note that either of these calls can fail - in which case the failure is reported back to the caller. The test application in *lined.c* ignores this possibility, under the rationale that the minimum heap of the application guarantees that these calls, made during program initialisation, will always succeed.

The call either returns `NULL`, in the case of an alloc failure, or the handle to be used to identify this particular editor in all subsequent `le_xxx` calls.

The meanings of the fields in the interface struct `IN_LINED` (defined in *lined.h*) are as follows:

`maxchars`	the maximum length of text that can be edited.
`winid`	the id of the window in which the editor is to appear.
`xoff`	the x-offset from the origin of the window to the top left of the editor (in pixels).
`yoff`	the y-offset from the origin of the window to the top left of the editor (in pixels).
`width`	the width of the editor (in pixels).
`height`	the height of the editor (in pixels).
`asc`	the distance (in pixels) between the top of the editor and the base line of the text edited.
`font`	the identifier of the font used to display the text.
`style`	the style of the font used to display the text.
`autoselect`	`TRUE` to automatically select the entirety of any text set into the editor by the calling program, `FALSE` to leave such text un-selected.

Note how these fields are set up in the routine `CreateLined`:

```
LOCAL_C LINED *CreateLined(INT yoff,TEXT *msg,INT emph)
    {
    IN_LINED init;
    LINED *ed;

    init.maxchars=20;
    init.winid=wMainWid;
    init.xoff=10;
    init.yoff=yoff;
    init.width=80;
    init.height=10;
    init.asc=8;
    init.font=WS_FONT_BASE+4;
    init.style=0;
    init.autoselect=TRUE;
    ed=le_init(&init);
    le_set_text(ed,p_slen(msg),msg);
    le_emphasise(ed,emph);
    le_visible(ed,TRUE);
    return(ed);
    }
```

The static `wMainWid` is one that is set up by the call `wStartup`. See the *Window Server Reference* manual.

The initial text of the editor is set in by a call `le_set_text` made after the call to `le_init`, but before the call to `le_visible` which causes the editor to actually be drawn. Also in between the `le_init` and `le_visible` calls is a call to `le_emphasise` to specify whether the editor should be displaying a flashing cursor (and also whether any selected region should be visibly highlighted).

Another call that could be made between `le_init` and `le_visible` is `le_set_cwidth`, to change the width of the flashing cursor from its default (which is two pixels wide).

As noted above, the way the application sets text into a lined object is with the call `le_set_text`. In this implementation, the application is required to specify the length of the string as a parameter to `le_set_text` - ie there is no requirement to pass the string in zero-terminated form.

On the other hand, the editor itself maintains the string, as it is edited, in zero terminated form - which may be convenient for the application.

The way the application can "sense" the contents of the string, as edited by the user, is simply to read this string out from the data maintained by the lined object. For this purpose, the form of the `LINED` struct needs to be known. This struct is defined in *lined.h*. Needless to say, most parts of the data in this struct are strictly read-only. If an application writes directly into this data, random problems can ensue later.

If a lined object is no longer needed, all the memory it uses can be freed by calling `le_destroy`. Be sure to have an independent copy of the string edited, before making this call.

Finally, the function `le_visible`, as well as initially making the editor visible, can also be used at some later stage to "hide" the editor again, if desired.

General comments

Device drivers for the HC

Note that the *i/o Devices Reference* manual gives details of how to program many of the peripherals that can be attached to an HC:

a parallel port.

a serial port (including xmodem and ymodem file transfer).

a magnetic card reader.

a bar code reader.

a modem.

The chapter *The HC in the Cradle*, later in this manual, gives details of the operation of the HC when located in a cradle.

Writing a customised shell process

The *System startup* section of the *Introduction* chapter of the *Window Server Reference* manual gives two examples of possible small alternative shell programs. The source for one of these, *lkshell.c*, may be found in *\sibosdk\demo*. As well as presenting the source, this section of the SDK raises various issues to do with replacing the built-in shell program with a customised one.

In case it is desired to create a shell process with functionality intermediate between *lkshell* and *corpshll* (which is the Command Shell), see the documentation, later in this manual, of each keyword supported by the Command Shell, for a reference to the C functions used to implement that keyword.

Developing applications on restricted-keyboard HCs

Developers writing for HCs with restricted keyboards lacking a full set of alphabetic keys face the problem that many commands that might ordinarily be typed into an HC during the course of program development - for example, file or SSD management commands in the HC Command Shell - simply cannot be typed into the HC, on account of the required alphabetic keys not being present on the keyboard.

In practice, preliminary development would probably be done using a different HC, with a fuller complement of keys. The program being developed would only be transferred to the restricted-keyboard HC at a later stage of development. However, the problem recurs at this later stage.

The comprehensive solution to this problem involves one of the fundamental principles of the HC - its interconnectability with other computers. Briefly, rather than the HC being controlled from its own keyboard, it can be controlled *from a remote keyboard*, say that of a PC. The commands are transmitted to the HC via one or other form of serial connection.

See the chapter *HC Command Shell* for more details of this mechanism.

CHAPTER 3

HC COMMAND SHELL

Overview

The HC Command Shell provides a MS-DOS like utility for functions that can be executed from a command line. The range of functionality covered includes file and SSD management, program management, information requests, and HC configuration.

Commands can be entered by typing at the HC command line in response to a `$` prompt. Alternatively, commands can be entered *remotely*, by typing at the terminal of a PC connected to the HC.

The HC will run batch files consisting of a sequence of commands. Batch files can be run in either of the above modes.

Batch file processing

Epoc batch files are plain text files consisting of a series of commands. Each command has a line to itself. By default batch files have extension *.btf*.

To invoke a batch file with name *backup.btf*, type `@backup` at the Command Shell `$` prompt. If necessary specify the full path of the batch file. Thus

```
@loc::b:\batch\backup
```

or

```
@rem::c:\hc\devp\restore.bat
```

would both invoke batch files. In the first case the file is assumed to have a *.btf* extension. In the second case the file extension is specified to be *.bat*.

Batch files can also call other batch files, and so on, up to eight levels deep.

Batch files are executed *synchronously*, i.e. no additional commands can be typed into a Command Shell until any batch files it is executing have completed.

Whilst batch files significantly enhance the utility of the HC Command Shell they do have some notable limitations:

- they cannot have parameters passed to them.
- they cannot contain conditional statements, such as `if ... goto ...`

These limitations can be got round by replacing the batch file with a program written in Opl, or in another high level language such as C.

Launching programs

The Command Shell can be used to launch both batch files and programs (either Epoc executables or OPL programs).

Epoc executables and OPL programs are run by simply typing their name without any additional prefix (except possibly for an `&` - see below).

When the following line is entered at the Command Shell

```
dojob
```

the HC will attempt to locate the corresponding command or file. The HC will execute this command or file when and if it is found. The search is carried out as follows:

- the HC checks that there is no internal command with the name `dojob`
- the HC looks for a file *dojob.opo* in the current directory
- the HC looks for a file *dojob.opo* on *a:*, *b:*, and *m:* (in the order given)
- the HC looks for a file *dojob.img*, first in the current directory, then (as above) on drives *a:*, *b:*, and *m:*, and then in *rom::*
- the HC looks along the same search path for a file *dojob.app*.

The search terminates once the command or file is found. Note that a file *dojob.opo* will be found in preference to a file *dojob.img*.

To ensure that a file *dojob.img* is run, enter the extension explicitly:

```
dojob.img
```

Programs are assumed to be Epoc executables unless they have the extension *.opo*, in which case they are assumed to be translated Opl programs.

Additional parameters can be passed to these programs. For example,

```
dojob b:
```

Synchronous programs and asynchronous programs

In contrast to batch files, which are always run synchronously (see above), programs can be run either synchronously or asynchronously thus exploiting the multi-tasking capabilities of the HC.

By default, programs are launched asynchronously. This means that while the program is executing, the user can task back to the Command Shell and continue to issue other commands.

When the program is started, it will by default (assuming it has a user interface) take over the foreground screen. To access the Command Shell, or indeed any other tasks that may be running on the HC at the time, press TASK as many times as is required. Every time TASK is pressed, a different program cycles into foreground.

Note that there is no need to quit the foreground program in order to start another - start a new program by pressing the TASK key until you get into the Command Shell, then type the name of the program at the command line.

However, users should avoid starting up new programs unnecessarily - since each additional program reduces the memory available for the programs already running.

To run a program synchronously, prefix the program name with an `&`. Note however that the command `offenable 0` should be issued before synchronously executing any lengthy program - otherwise it will be impossible for the user to switch the HC off until the program has completed.

Terminating programs

Many programs include a facility that allows user termination. For example many programs contain an Exit menu command.

When required the user can kill a program from the Command Shell, using either the `terminate` or `kill` commands. As explained in the alphabetical listing (see below), `terminate` should be used in preference to `kill` whenever possible.

A program run synchronously can not be terminated by tasking to the Command Shell that launched it - since that Command Shell is inaccessible until the program terminates. In extreme circumstances it may be necessary to reset the HC.

When a program launched from a Command Shell terminates, either normally or abnormally, the Shell reports this fact to the user.

The command line editor

Up to eight previous commands can be reviewed by means of the UP and DOWN cursor keys at the command line. Any previous command displayed in this way can be edited before being issued again.

To clear the command line at any time, press ESC.

As might be expected, each individual command is entered to the HC by pressing ENTER after typing its name. In most cases, the name can be abbreviated, as indicated in the alphabetical listing below.

Pausing the screen display

Some commands (such as `lproc` and `lseg`) automatically pause when a screenful of information has been displayed. Other commands (such as `dir`) must be entered with a `/p` flag to obtain the same effect. In either case, pressing any key will resume the display (though the ESC key sometimes terminates the command listing).

At all times, the display of the Command Shell can be paused, independently, by means of the PSION+LEFT key combination (or by SHIFT+LEFT on restricted keyboards (this feature is shared by all console programs). Again pressing any key will resume the display.

Additional copies of the Command Shell

The Command Shell can be run from the command line just like any other program. The first and subsequent copies of the Command Shell differ only in that, by default, subsequent copies terminate as soon as they have processed the command lines passed to them.

For example typing `sys$shll ver` runs a copy of the Command Shell with the command line argument `ver`. The effect is the same as simply typing `ver` on its own except that the new copy of the Shell terminates after the ver command completes. The display then reverts to that of the previous Command Shell.

To force a copy of the Command Shell to pause before terminating, type `/p` immediately after `sys$shll`. Thus

```
    sys$shll /p ver
```

causes the display to pause waiting for any keypress, after completing listing the version information.

Exceptionally, if there is little available memory on the HC (for example, if there are many files on *m:*) additional copies of the Command Shell may fail to perform fully as expected.

Sending commands from a remote PC

The utility of running second copies of the Command Shell is most apparent when used in conjunction with MCLink. MCLink allows programs on the remote computer (in this case, the HC) to be invoked with the MCLink `run` command.

For example, typing

```
    run sys$shll /p del *.bak
```

at the MCLink command line is essentially equivalent to typing

```
    del *.bak
```

at the command line of the HC.

Operators may find typing at the PC to be more convenient than typing on the naturally more restricted keyboard of the HC. In cases where the HC has only a numeric keyboard, commands must be entered using a mechanism such as MCLink running on some remote computer.

The following alias may prove especially useful: typing

```
    ! <text>
```

at the MCLink command line is shorthand for typing

```
    run sys$shll <text>
```

Thus typing

```
    ! /p del *.bak
```

at the MCLink command line may be a yet more convenient way of issuing the HC with the command

```
    del *.bak
```

Often even typing these few characters is undesirable (it is impossible in the case of restricted keyboard HCs) and so a batch file is used instead. A batch file *autoexec.btf* is placed in the root directory of an SSD. This batch file is executed whenever the HC is reset - if the file contains the command `link`, the Link software will automatically be started every time the HC is reset. Another (more advanced) possibility is to place an alternative (custom) shell on an SSD, before resetting the HC.

More on running programs remotely

Even if an alternative (custom) shell is running on an HC, the Command Shell can in many cases still be invoked by means of typing (eg)

```
! /p ver
```

at the command line of MCLink. This mechanism may be found useful in cases where it is briefly required to access the functionality of the Command Shell, even though, ordinarily, a custom shell is run in place of the Command Shell.

Occasionally this mechanism will fail to work - for reasons explained below - with a second copy of the custom shell being run instead.

The MCLink `run` command proceeds as follows:

- first, an extension *.img* is added to the program name supplied and if no extension was explicitly supplied
- the Link software starts looking, on the remote computer, for a program with this name; if at any stage a program with this name is found, an attempt is made to execute it; if this attempt fails, the search continues
- the first place searched is *the current path of the Link software on the remote computer* (see below for an explanation of the concept of current path)
- the search continues, if required, in the ROM of the remote computer
- finally, if required, the search continues on all the root directories of the remote computer, in alphabetical order.

Accordingly, if the current path of the Link software on the HC contains a custom copy of *sys$shll.img*, this copy will be launched by an MCLink `!` command; otherwise, it will be the Command Shell (from the HC ROM).

In practice, the only way for the current path of Link software on an HC to differ from *m:\* is for a `set` command to be issued *before the Link software is started*.

There is one further complication when attempting to simultaneously run two different programs with the same name. Ordinarily, Epoc will refuse to allow the second program to run and will generate a "File already exists" error message. The only exception is if the second program is in ROM. This explains why the Command Shell can be started when a custom shell is already running, whereas a custom shell cannot be started with the Command Shell still running (try it and see).

Auto-terminating and non-auto-terminating Command Shells

A Command Shell will only auto-terminate if invoked with a command line. Whether an instance of the Command Shell is the first or an additional copy is irrelevant. To run an additional copy of the Command Shell that does not auto-terminate after processing its command line (either straightaway, or after pausing to receive a keypress), just type `sys$shll` by itself, without any additional parameters (any "Capture failed - File already exists" error message can in this case be safely ignored).

Files and directories

File In Use error messages

On the HC an open file can only be accessed by the application that opened it. An attempt by another application to modify the file will lead to a File In Use error message.

As a consequence file commands issued from the Command Shell will fail when asked to access an already open file. For example an attempt to copy an open file will generate a File In Use message.

Default path and current directory

In Epoc there is one current path for each running process (in Epoc it is preferable to refer to the *current path* rather than the *current directory* as this also includes the drive and the filing system). In contrast MS-DOS logs a current directory for each drive and thus has as many current directories as drivers.

The current path of a Command Shell can be altered using the `cd` command. For example `cd b:\play` would change the current path to `b:\play`. Note that this could have a quite different effect in MS-DOS. In MS-DOS change the current directory to *a:\work\* then type *cd b:\play\* followed by a `dir` command. The `dir` command will list the files in *a:\work\* and not those in *b:\play\*.

Changing the current path for one application does not alter the current path for any other application. This is in contrast to and an improvement on MS-DOS. In this case the currently logged directories in the command shell can be annoyingly altered by running (synchronously!) another process. By the time the second process has terminated the MS-DOS command shell may have been logged to a different drive and a different directory.

The HC Command Shell also supports the command `set` to alter the so-called *default path* which affects *all* applications subsequently launched. The `set` command has no effect on the current path of the present application but provides the *initial current path* for all future launched applications (regardless of where these applications are launched from). Thus the sequence of commands

```
cd m:
set b:
sys$shll
dir
cd a:
dir
exit
dir
```

will bring about the following sequence of directory listings:

- first the second copy of the Command Shell lists the contents of *b:\* since its current path was set (on initialisation) to *b:* with the `set` command.(in the parent Command Shell)
- next, after changing the current path of the second Shell to *a:\*, the second `dir` command lists the contents of *a:\*
- finally, once the second copy of the Shell has been exited, the last `dir` command lists the contents of *m:\* - since the current path of the parent Shell has been affected by neither the `set` command (as that does not alter the current path) nor the `cd` command (as that altered the current path of a different process).

Note that any changes to the path of a Command Shell inside a batch file will continue to have effect after the termination of the batch file, since no new process is run up, just by virtue of a batch file being executed.

Specifying file names as command parameters

It is not always necessary to supply the full specification for a filename when passing it as an argument to a Shell command. The missing parts (if any) are filled in from the current path.

Thus if the current path is *m:\img\*, the command `att job.img -r` operates on the file with full path name *m:\img\job.img* and the command `att b:\backup\job.img` operates, naturally enough, on the file *b:\backup\job.img* - regardless of whether the current path is on *m:*, *b:*, or whatever.

Beware that `att b:job.img` is equivalent to `att b:\img\job.img` *and not* `att b:\job.img` (the first and last forms differ only in the presence or not of a back-slash immediately after the colon). The reason why these two forms are interpreted differently is that the filename *b:job.img* is interpreted as having three parts:

- a drive (*b:*)
- a basic name (*job*)
- an extension (*.img*).

As the path is not explicitly specified, the path specified in the current path of the application will be assumed. If the (incorrectly specified) file is not found the command will fail with the error message "Directory does not exist".

To specify a file *job.img* on the root directory of *b:*, type `b:\job.img`, including the crucial `\` character.

More details on filename specifications

Filenames are assumed to have *five* parts:

- a *filing system* (eg *loc::* or *rem::*)
- a *drive* (eg *b:*)
- a *path* (eg \ or *\accounts\jan\*)
- a *basic name* (eg *job*)
- an *extension* (eg *.img*).

The current path of an application contains the first three components. It does not contain the last two.

The filing system will always be assumed to be that specified in the current path unless an alternative is explicitly supplied.

Note that various Shell commands may unexpectedly fail to work if the filing system specified in the current path is set to *rem::*, and the remote link connection is subsequently broken.

Specifying paths as command parameters

When using a command such as `cd` it is often more convenient to omit one or more components of the path as any missing components will be filled in from the current path. Thus with a current path of *m:\img\* the command `cd files` would be equivalent to `cd m:\img\files` and the command `md tools` would be equivalent to `md m:\img\tools`.

Note that it can sometimes be an error to supply a trailing back-slash. Whilst `md m:\img\tools\` is acceptable, `md tools\` is not. The reason is that the trailing back-slash indicates that a path component follows - the `md` command does not expect a path component to follow.

The requirements of generality

The user might consider the syntax of HC Command Shell commands to be more limiting than the MS-DOS equivalent - the syntax of commands such as `md` and `rd` on the HC is not the same as that of the `md` and `rd` commands in MS-DOS although there are considerable similarities.

The extra limitations stem from a central design feature of the Epoc operating system - under Epoc an application can directly access files that are stored on a remote computer whose filing system may or may not be MS-DOS. Alternative remote filing systems that need to be borne in mind include Unix, Vax VMS, and the Apple Macintosh operating system.

Thus if the HC is connected to an Apple Macintosh computer, the following could be entered at the command line:

```
cd rem::hd40:hcdevp:stock
```

Accordingly, the HC Command Shell does not simply approach filenames and path specifications in terms of questions of back-slashes (were the Shell to insert a back-slash at the end of the above command, "on behalf of the user", this would, most decidedly, *not* be what the user intended). Instead, the approach is much more general, in terms of the five part breakdown of filename specifications discussed two sections previously.

Similarly, the HC provides no support for the syntax of "double dot" (for the parent directory) and "single dot" (for the current directory).

Although this extra discipline has its occasional drawbacks, the advantages that it brings with it are an important part of the vital *inter-connectable* feature of the HC.

Alphabetical listing

Notation

This list of commands uses the following syntax:

```
COM[MAND] supplied-parameter [optional-parameter]
```

Items shown in square brackets ([]) are optional. To include optional information, type only the information within the brackets. Do not type the square brackets themselves.

Legal shortened versions of commands may be inferred from the syntax given. Thus in the above (generalised) example, `COM` would be an acceptable shortened form of `COMMAND`. Any intermediate form between `com` and `command` would also be acceptable - eg `comm` (but not `comd`, needless to say).

Commands can be typed in any combination of lower and upper case. For example, except where clearly stated to the contrary below, pairs of command such as `wnot on` and `wnot ON` are completely equivalent.

Commands must be separated from their options by inserting a space character.

Default values may be assumed if some options are not supplied. Default values of particular commands are given in the individual command descriptions which follow.

Note: the following list actually contains two entries that are not really commands of the Shell, in the strict sense, but are just the names of programs in the rom: `link` and `batchk`. However, this distinction may seem irrelevant to the user, and so, for convenience, these commands are listed too.

How commands are implemented

In many cases, the description of a command below gives the name of some of the key C functions involved in the implementation of that command. This is provided partly for interest, partly as an additional reference source (so that the corresponding section of the *Plib Reference* or *Window Server Reference* manuals can be consulted), and partly as a guide for people wishing to write an alternative shell (or shell-like) program.

Command ATTRIBUTE Set or clear file attributes (ATTRIBUTE)

```
ATT[RIBUTE] filename [(+/-)h] [(+/-)s] [(+/-)m] [(+/-)r]
```

Sets or resets the hidden (`h`), system (`s`), modified (`m`) and/or read-only (`r`) attributes of a file.

For example, `att list.dat -m +s` clears the modified attribute and sets the system attribute of *list.dat,* without altering its hidden or read-only attributes.

For each of `h`, `s`, `m`, and `r`, a prefix of `-` clears the corresponding attribute, and a prefix of `+` sets it. The four attributes can be specified in any order, and any combination of the four bits can be set or cleared at once. Omitting all four is pointless: nothing will happen.

The `attribute` command does not accept a wild card specification.

This command is implemented via the C function `p_sfstat`.

Note that the `dir` command includes the attributes of files as part of its display.

Command AUTO Set time to auto-switch-off (AUTO)

```
AUT[O] seconds
```

Sets the time for auto-switch-off.

The auto-switch-off time is set to `seconds`. If `seconds` is `-1`, auto-switch-off is disabled. The maximum value for `seconds` is 32767, and the minimum non-zero value is 15.

This command is implemented via the C function `p_setauto`.

Command BACKLIGHT Set backlight time-out (BACKLIGHT)

```
BACK[LIGHT] [time]
```

Sets the backlight auto-time-out to `time`, or if `time` is omitted, displays the current setting (in hexadecimal).

The value of `time` is in ticks, ie 1/32 of a second.

If `time` is zero, the backlight will remain on for as long as the HC is switched on.

Passing `time` as negative has the effect of *disabling* the BACKLIGHT key. In that case, the backlight can only be switched on under software control.

This command is implemented via the C functions `p_backlight`, `p_getbacklight`, and `p_setbacklight`.

Command BATCHK Start battery check program (BATCHK)

```
BATCHK interval
```

Starts the program *rom::batchk* (if found), which monitors the voltages of the main and backup batteries.

The value of `interval` gives the time period, in tenths of a second, between the time when checks are made.

If either battery is found to be low when a check is made, a Notifier is displayed.

The *batchk* program also captures the INFO key so that, whenever this key is pressed, the user is presented with information on the current voltages of the batteries. At the same time, a Notifier is displayed if either battery is low.

Passing `interval` as `0` has the effect that a check on the battery voltages is performed only when INFO is pressed; no timer operates in this case.

If `interval` is omitted, it defaults to `3000` (5 minutes). Any value of `interval` less than `100` has the same effect as passing `0`.

A copy of *batchk* is automatically run when the Command Shell starts. The Command Shell starts *batchk* with a value of `interval` equal to zero.

Attempting to run a second copy of *batchk* without terminating the first will result in an error message and then a notification of abnormal program termination (the second copy of *batchk*). In order to change the value of `interval` that is in operation, to ten minutes (say), the following has to be entered:

```
    term batchk
    batchk 6000
```

See the `lowbat` command for an independent method of checking the battery voltages.

The core functionality of the *batchk* program is provided by the C calls `p_supply` and `p_wsupply`. Applications in which it is critical that battery power does not drop too low during some activity should make their own calls to these functions when needed.

Command BATTERY Specify battery type (BATTERY)

```
BAT[TERY] type
```

Specifies which type of main battery is installed. This information may be used by other software on the HC, affecting (eg) when low battery warnings are issued.

Allowed values of `type` include:

1	alkaline batteries
2	600 mAh Nickel Cadmium batteries
3	1000 mAh Nickel Cadmium batteries
4	500 mAh Nickel Cadmium batteries.

This command is implemented via the C function `p_setbat`.

Command CD — Change directory (CD)

```
CD [path]
```

Changes to a different path, or (if `path` is omitted) displays the current path.

For example, to change the current directory from *\work\product\* to *\work\admin\*, type

```
    cd \work\admin\
```

To move to a directory below the current one, only the path from the current directory needs to be entered. So to change from *\work\admin\* to *\work\admin\forms\*, the following command could be used:

```
    cd forms\
```

The trailing back-slash in the above commands can be omitted. Thus `cd forms` instead of `cd forms\`.

There is no support for a command such as `cd ..` (to move to the parent directory).

Type `cd b:` (or `cd b:\`) to change to the root directory of *b:*.

This command is implemented via the C function `p_setpth` (amongst others).

Command CONFIG — Set language file (CONFIG)

```
CON[FIG] filename
```

Changes the language data file to that specified. If `filename` is omitted, the effect is to revert to the file *sys$ctry.cfo*.

If the extension is omitted, it is assumed to be *.cfo*.

The file given must be in the ROM of the HC. Unless a specially customised version of the ROM has been made, this in practice limits the use of this command to

```
    config custom$.dat
```

where the file *custom$.dat* has been specially prepared by means of the tool *romwrite*.

The effect of specifying a file that has an unsuitable form is drastic: almost certainly, the HC will require a hard reset to recover.

This command is implemented via the C function `p_setconfig` (amongst others).

Command COPY — Copy file(s) (COPY)

```
COP[Y] source_filespec target_filespec
```

Copies one or more files, possibly changing their names in the process.

Any part of the target filename that is not specified (for example, the extension) and which cannot be filled in from corresponding parts in the current path is taken from the corresponding part of the first pathname.

The wildcards `*` and `?` can be used to copy multiple files.

For example, `copy fred.* a:\jim.*` copies all files such as *fred.btf* from the current directory into the root of *a:\*, renaming them (eg to *jim.btf*) in the process.

As a possibly surprising example, if the current path is *m:\*, the command `copy a:\file.lis file.old` has the effect of copying the named file to *m:\file.old*.

As files are copied, the names of the files created are listed on the screen.

A file cannot be copied onto itself. If an attempt is made to do this, the `copy` command quits, and an error message such as the following is displayed:

```
    Copy failed - file or device in use
```

This command is implemented via object-oriented techniques using the `fman` object in *Olib.dyl*.

Command D — Brief directory listing (D)

```
D [/p] [filespec]
```

Lists specified filenames in a directory, without any additional information except for the total size and the total number of bytes free on the current device.

Typing `d` by itself lists all filenames in the current drive and directory. Typing `d` and a path, such as *a:\*, lists all entries in the specified directory. If a filename without an extension is included (*invoices*, for example), all files named *invoices* in the specified directory will be listed, whatever their extension.

The wildcards `*` and `?` can be used in the file specification.

The `/p` flag causes the display to pause at the end of each screen. When the display is paused, it can be resumed by pressing any key. However, if `ESC` is pressed, the directory listing is terminated.

This command is implemented via the C functions `p_open(P_FDIR)`, `p_dinfo`, and `p_iow(P_FREAD)`.

Use the `dir` command for a fuller listing of the details of files.

Command DATE — Display date and time (DATE)

```
DAT[E]
```

Displays the current date and time.

This command is implemented via the C function `p_date`.

Use the `setdat` command to change the date and/or time.

Command DELETE — Delete file(s) (DELETE)

```
DEL[ETE] filespec
```

Deletes the specified file or files.

To delete more than one file at a time, the wildcards `*` and/or `?` can be used. Alternatively, the following deletes all files in the directory *\temp*:

```
    del \temp\
```

As files are deleted, the names of the files deleted are listed on the screen.

This command is implemented via object-oriented techniques using the `fman` object in *Olib.dyl*, which result, in the end, in calls to the C function `p_delete`.

See also the command `rd`, which, in contrast to `del`, can delete directories.

Command DEVICE — List devices (DEVICE)

```
DEV[ICE] [filespec]
```

Lists all devices ("drives") in the filing system specified by `filespec`. The only relevant part of `filespec` is the filing system (*loc::*, *rem::*, or whatever).

For example, `dev rem::` lists the devices in *rem::* - assuming a remote connection is established.

Typically, the command `dev` just results in the following listing:

```
    List of file devices for LOC::
    A: - OK
    B: - OK
    M: - OK
```

In practice, the only time a device will be reported as other than `"OK"` will be if the connection to a remote computer is broken midway through listing the devices of *rem::*.

This command is implemented via the C functions `p_open(P_FDEVICE)` and `p_iow(P_FREAD)`.

Command DIR — Full directory listing (DIR)

```
DIR [/p] [filespec]
```

Lists all the specified files in a directory, together with their sizes, the time and date of their last modification, and their attributes.

The wildcards `*` and `?` can be used in the file specification.

The `/p` flag causes the display to pause at the end of each screen. When the display is paused, it can be resumed by pressing any key. However, if ESC is pressed, the directory listing is terminated.

This command is implemented via the C functions `p_open(P_FDIR)`, `p_dinfo`, `p_iow(P_FREAD)`, and `p_finfo`.

Use the `d` command for a briefer listing of the details of files.

Command ENV — Display or set environment variable (ENV)

```
ENV [var[=[value]]]
```

Displays or sets the value of environment variables.

With no parameters, the values of all current environment variables are displayed. If `var` is given but without any trailing equals sign (`=`), the values of all environment variables matching the specification in `var` are listed. If the equals sign (`=`) is given too, the environment variable `var` is set to `value.` But if the equals sign is given whilst `value` is omitted, the environment variable `var` is deleted.

For example:

`env $WS*`	displays the values of all environment variables whose names start with `$WS`
`env last=34`	sets the value of `last` to the string `34`
`env last=`	deletes the environment variable `last`.

Values are displayed inside square brackets. The list pauses when the screen is full.

Note that environment names and values are both case dependent. Thus the environment variables `group` and `GROUP` would be distinct.

Indeed, environment names and (more likely) environment values can even be binary. Non-printable byte values are displayed as (eg) `<04>` or `<01>`. There is no mechanism for *setting* binary values from the Command Shell.

This command is implemented via the C functions `p_findenviron`, `p_delenv`, and `p_setenv`.

Command EXIT — Exit level (EXIT)

```
EXI[T]
```

Exits the Command Shell. May be used to terminate second copies of the Command Shell that are no longer required.

If the `exit` command is typed into the first copy of the Command Shell, the HC will automatically re-launch a shell process, as explained in the chapter *Introduction to the HC*.

If the `exit` command is found in a batch file, all that happens is that the batch file is terminated, and control passes back to the previous level of batch file (or to the command line).

The command is implemented (when not in a batch file) by the C function `p_exit`.

Command FORMAT — Format device (FORMAT)

```
FOR[MAT] [device:][volname]
```

Formats Ram and Flash SSDs (or the internal disc *m:*).

The command detects the type of SSD and places the appropriate format information onto the disk. This information differs for Flash and Ram SSDs.

The volume name `volname` is optional.

For example, `format a:new` will format the *a:* device, giving the volume name *new*.

If `device:` is omitted, the internal memory is formatted.

Note carefully that no warning is given before the formatting takes place. So accidentally typing (eg) `for b` could be disastrous:

- since no colon is typed, `b` is interpreted as the volume name
- since no device name is specified, formatting defaults to *m:*
- accordingly, all data on *m:* is lost in a trice (with the volume name of *m:* being set to *b*).

The mere fact that there are read-only files on an SSD will not prevent it from being formatted. However, if an SSD has the write-protection switch set, it will not be possible to format it.

Another reason for `format` being disallowed for a disk would be if there are any *open files* on it. In this case, the `format` request will fail with the error message "File or device in use".

This command is implemented via the C functions `p_open(P_FFORMAT)` and `p_read`.

Command FREE — Display free memory (FREE)

```
FRE[E]
```

Displays the amount of free RAM in Kbytes.

Note that this in general exceeds the amount of bytes free in *m:*, as reported by a `dir` or `d` command. The discrepancy is because some parts of internal memory are reserved for code and data segments; not all of it can be allocated to the contents of *m:*.

This command is implemented via the C function `p_sgfree`.

Command KILL — Kill a process (KILL)

```
KIL[L] procname
```

Kills the first process found matching the specification in `procname`.

To kill a specified instance of a number of running tasks, all with the same name, the exact process name must be found out and used. Eg `kill job.$09` or `kill job.$14`.

Use the `lproc` command to give the full process names of all current processes.

Note that `kill` should only be used as a last resort, as it does not allow the process to tidy up before exiting - this is a problem with the Link application which starts a number of sub-processes. To shut down a process, `terminate` should normally be used in preference to `kill`.

This command is implemented via the C function `p_pkill`.

Command LDEV — List device drivers (LDEV)

```
LDE[V] [device_spec]
```

Lists all specified device drivers. The list includes all ROM-resident device drivers, as well as external ones that are currently loaded.

If `device_spec` is omitted, it defaults to `*.*`.

For each device driver listed, the label *ldd* or *pdd* is given - the former for logical device drivers (which are hardware-independent), the latter for physical device drivers (which are hardware dependent).

For example, entering `ldev con` displays

```
    List of devices:con
    ----------------
    LDD - CON (units=-1)
```

The value given for `units` is the number of channels a logical device driver can support. A value of `-1` means that an unlimited number of channels can be opened.

As another example, entering `ldev fsy` displays

```
    List of devices:fsy
    ---------------
    PDD - FSY.REM
    PDD - FSY.LOC
    PDD - FSY.ROM
```

listing the three ROM-resident filing system device (*fsy*) drivers **-** for *rem::*, *loc::*, and *rom::*.

This command is implemented via the C functions `p_devfnd` and `p_devqu`.

Command LINK Start Link program (LINK)

```
LINK [-b<baud>] [-p<port>] [filename]
```

Starts the Link communication software on the HC.

If `filename` is specified, it is assumed to specify a *.trm* file, and in that case, there should be no other parameters on the command line. The extension *.trm* is supplied for `filename` if required.

If the command line is empty, the Link software searches as follows for a file *mclink.trm* to configure it:

- first, in the current path of the Link software
- next, in the HC ROM (where it will indeed find a file *mclink.trm*).

The format and creation of *.trm* files is discussed in the *Additional System Information* manual.

Possible values of `baud` range from `19200` and `9600` all the way down to `110`, `75`, and `50`, with all common baud rates in between being supported. In the absence of a command line and if no external *mclink.trm* file is found, `baud` defaults to `9600`. If the `port` or `serial_device` is specified but not `baud`, `baud` defaults to 19200.

The only time it is necessary to specify `port` is if there are serial expansion devices in both the top and the bottom of the HC. In this case, the parameter `-p1` means to use the top port, and `-p2` means to use the bottom port. Otherwise, the Link software simply uses whichever port is available.

(Other parameters are also possible but are omitted from the present description. See the chapter *Mclink, Mcprint, and Slink* in the *Additional System Information* manual.)

Just typing `link` should suffice in the majority of cases.

To terminate the Link software at some later date, type `term link`.

To discover whether or not Link software is running, type `lproc link`.

If the `link` command is issued while Link is already running, a second copy of Link will be launched briefly, but will quickly exit with the error number -32 (or 224), meaning that a process *link.** already exists. No harm will ensue as a result.

See the section *Connecting to other computers* in *Introduction to the HC*, for more details.

Command LOWBAT Configure low battery warnings (LOWBAT)

```
LOW[BAT] state
```

If `state` is `ON`, the HC will check, each time the HC is switched on, for either of the batteries being low. On detecting a low battery, the HC will issue a warning in the form of an information message in the bottom right hand corner of the screen.

If `state` is `OFF`, this behaviour will not take place. (This is the default.)

This command is implemented via the C function `wSystem`.

See also `batchk` for an independent method of periodically checking the battery voltages.

Command LPROC List processes (LPROC)

`LPR[OC] [process_spec]`

Lists information about all specified processes. The information listed is:

- the full process name (in the form *batchk.$07*)
- the size, in bytes, of the process data segment (given in hexadecimal)
- the current state of the process.

If `process_spec` is omitted, it defaults to `*.*`.

Possible values of the state of the process are:

`CURRENT`	the process is currently receiving cpu
`READY`	the process has some events ready to process, as soon as cpu is given to the process by the multi-tasking scheduler
`DELTA`	the process is "sleeping" (eg as a result of calling the C function `p_sleep`)
`SUS`	the process has been suspended
`SEM`	the process is waiting for some event to happen.

Additionally, the text `WSusp` will be displayed if the process is *waiting* to be suspended.

For example, entering `lproc sys$shll` may produce the display

```
List of processes:sys$shll
-----------------
SYS$SHLL.$05 3DA0 SEM
SYS$SHLL.$11 3DA0 CURRENT
```

One common use of the `lproc` command is to check whether Link software is currently running: `lproc link`.

This command is implemented via the C functions `p_pfind`, `p_getosd`, and `p_sgsize`.

Command LSEG List segments (LSEG)

`LSE[G] [process_spec]`

Lists all memory segments currently in use by the specified process(es).

If `process_spec` is omitted, it defaults to `*.*`.

The information listed about each memory segment is:

- its size in paragraphs (one paragraph is sixteen bytes)
- its segment address
- its access count.

Values are displayed in hexadecimal.

At the end the display, the total size in paragraphs of all the free segments is given (this gives the same value, when converted into Kbytes, as `free`).

This command is implemented via the C functions `p_sgfind`, `p_getosd`, and `p_sgfree`.

CommandMASTER Display time/date of mastering (MASTER)

`MAS[TER]`

Displays the time and date when the ROM was mastered.

The command is implemented by the C function `p_finfo`, passing as a parameter a file known to be in the ROM (*rom::sys$shll.img*).

Command MD Make directory (MD)

`MD path`

Makes a directory.

When a directory is created, it will appear in the current directory, unless a different path is explicitly specified.

It is possible to omit the trailing back-slash from the path specification.

The following commands both create a directory named *\work* in the root directory of the current drive:

```
    md \work\
    md \work
```

This command is implemented via the C function `p_mkdir`.

Command NOTIFY Control whether the Notifier appears (NOTIFY)

`NOT[IFY] state`

Controls whether the Notifier ever appears as a result of a file operation carried out by the Command Shell.

If `state` is `ON` (this is the default), and a file operation unexpectedly fails to find an SSD that was present earlier, a Notifier will be presented giving the user the opportunity to replace the SSD, instead of just having the file operation fail.

If `state` is `OFF`, no such Notifier will be displayed.

The `notify` command in the Shell has no effect on whether Notifiers are ever displayed by *other* programs.

This command is implemented via the C functions `p_setnotify` and `p_getnotify`.

Command OFFENABLE Enable off-key handling (OFFENABLE)

`OFFE[NABLE] value`

If `value` is `0`, the Command Shell gives up its capture of the OFF key, thereby allowing other applications to capture this key to do their own processing of it.

If `value` is any non-zero number, the Command Shell attempts to capture the OFF key again.

This command is implemented via the C functions `wCaptureKey` and `wCancelCaptureKey`.

Note that there is no special need to have *any* application capture this key, since by default, the HC simply switches itself off when this keypress is received. The behaviour of the Command Shell in response to the OFF key adds nothing to this.

Indeed, it is recommended that the command `offenable 0` be issued early in any *autoexec.btf* start-up batch file.

Command RD Remove directory (RD)

`RD path`

Deletes a directory, including any files in it (and subdirectories).

Note that, in contrast to MS-DOS, there is no requirement to delete all the files in a directory before removing the directory. Further, no warning is given before the directory is removed.

In another difference from MS-DOS, it is perfectly possible, in the HC Command Shell, to remove the directory where the current path is. All that will happen is that subsequent commands such as `dir` may fail until such time as the current path is changed.

As files and directories are deleted, their names are listed on the screen.

The `rd` command does not accept a wildcard specification.

This command is implemented via object-oriented techniques using the `fman` object in *Olib.dyl*, which result, in the end, in calls to the C function `p_delete`.

Command RENAME Rename file(s) (RENAME)

```
REN[AME] filespec filename
```

Changes the name of a file or files.

The command renames all files matching `filespec` - which can include wildcards.

For example, the command `ren work.* play.*` changes the names of all files called *work* in the current directory (regardless of extension) to *play*, with the extension being preserved across the rename.

As files are renamed, they are listed on the screen.

Because it is not possible to rename files from one directory to another, the command fails if any path specified with `filename` (explicitly or implicitly) differs from that of `filespec`.

It is not possible to rename a file to have the same name as a file that already exists.

This command is implemented via object-oriented techniques using the `fman` object in *Olib.dyl*, which result, in the end, in calls to the C function `p_rename`.

Command RESUME Resume a suspended process (RESUME)

```
RES[UME] procname
```

Resumes the previously suspended process `procname`.

See also `suspend`.

Some care needs to be exercised in the use of this command, to resume an instance of process *job* (say), in any case where there may be more than one instance of *job* running at a time. This is because the command simply attempts to resume the first instance of the process *job* found, regardless of whether or not that particular process is actually suspended.

This command is implemented via the C function `p_presume`.

Command SET Set default path (SET)

```
SET path
```

Sets the default path.

For example, the command `set b:\` has the effect that all subsequently launched tasks start with their current paths set to *b:\*. This may be useful if a program assumes that its current path on start up is where it should read and/or write certain files.

See the earlier section *Files and directories* for further discussion.

This command is implemented via the C function `p_setdefaultpath`.

Command SETDATE Set time and date (SETDATE)

```
SETD[ATE] dd/mm/yy hh:mm:ss
```

Sets the date and time.

For example, `setdate 26/02/92 15:10:00` sets the date to the 26th of February, 1992, and the time to ten minutes past three in the afternoon.

All parameter fields must be present, with a two digits being supplied for each field.

The time should always be specified in 24 hour format.

If `yy` is in the range `70` to `99`, the century is set to `19`. Otherwise it is set to `20`. That is, the range of years that can be set is from `1970` to `2069`.

This command is implemented via the C function `p_sdate`.

Command SUSPEND Suspend a process (SUSPEND)

```
SUS[PEND] procname
```

Suspends the first process found matching the specification in `procname`.

To suspend a specified instance of a number of running tasks, all with the same name, the exact process name must be found out and used. Eg `suspend job.$09` or `suspend job.$14`. If only one instance of *job.img* is running, it suffices to enter `suspend job`.

Use the `lproc` command to give the full process names of all current processes. Use the `resume` command to reverse the effect of a `suspend` command.

This command is implemented via the C function `p_psuspend`.

CommandTERMINATE Terminate a process (TERMINATE)

```
TER[MINATE] procname
```

Terminates the first process found matching the specification in `procname`.

To terminate a specified instance of a number of running tasks, all with the same name, the exact process name must be found out and used. Eg `ter job.$09` or `ter job.$14`.

For most applications, the effect of being terminated is identical to being killed: the application is interrupted immediately, with no chance being provided for data being saved to file or to environment variables. However, an application can make use of an operating system service (in C, `p_onterminate`) to specify behaviour to be invoked whenever the application is to be terminated in this way.

This command is implemented via the C function `p_pterminate`.

Command TYPE Type a text file (TYPE)

```
TY[PE] filename
```

Prints a text file to the screen.

There is no provision for the display to pause itself automatically. However, the user can pause the display at any time, in the usual way, by pressing PSION+LEFT.

This command is implemented via the C functions `p_open(P_FTEXT)` and `p_read`.

Command VER Display software version number (VERSION)

```
[VER]SION
```

Displays the Operating System (Epoc) version number, the HC Rom version number, and the Command Shell version number.

This command is implemented via the C functions `p_version` and `p_romversion`.

Command WAIT Wait for a process to complete (WAIT)

```
WAI[T]
```

Waits until a process completes. The message "Waiting" is displayed and the Shell becomes non-interactive until such time as another process completes.

To break out of this mode, press PSION+ESC.

Commonly, this command will be used inside batch files in the following general pattern:

```
...
<launch program asynchronously>
...
<some processing>
...
wait
...
```

Command WNOTIFY   Configure Notifier appearance (WNOTIFY)

`WNO[TIFY] state`

Configures the appearance of the Notifier.

If `state` is `OFF`, the Notifier will be drawn in the same way as it was for software versions prior to release 1.50 of the HC ROM. (This is the default.)

If `state` is `ON`, the Notifier will be drawn in an arguably more attractive form, and will also be displayed automatically whenever any program terminates abnormally. This form of the Notifier also involves less RAM usage.

The `W` in the name `Wnotify` stands for *Window Server* **-** the part of the operating system which actually produces the more attractive version of the Notifier display.

To see what a Notifier looks like under either of the two methods, first terminate Link (if it is running) and then type (eg)

```
link x
```

This brings about a "File does not exist" Notifier, since the file *x.trm* (presumably) does not exist.

This command is implemented via the C function `wSystem` (amongst others).

Using the command `fre` before and after typing `wnot on` should reveal a memory saving of around 7 Kbytes.

What happens when the Command Shell starts

Exactly what happens when the Command Shell starts depends on whether it has been passed a command line.

When the Shell is started by the Window Server (after a reset, for example), no command line is passed. In most other cases, however, the user would pass a command line to the Shell.

For example, typing

```
run sys$shll /p ver
```

into MCLink has the effect of running a copy of *sys$shll* on a remote HC, passing it the command line `/p ver`.

When no command line is passed

The Command Shell checks to see if a process *sys$ntfy* is already running. If not, it launches one from the HC ROM. (However, if the Window Server has taken over the notifier function, the independent *sys$ntfy.img* process will quickly discover this fact, and terminate itself silently.)

Similarly, the program *batchk* is launched, if it is not already running.

Next, the Command Shell searches for a batch file *autoexec.btf* and executes that, if one is found. The search is on the root directories of *a:*, *b:*, and *m:*, in that order. If no such batch file is found, the user is prompted to insert an SSD containing this file, and to press ENTER to continue. However, the search can be abandoned by pressing PSION+ESC instead.

Then some system information is displayed on the screen: the version numbers of the ROM-resident software, the date and time, the size of the display screen, the battery type and internal power supply type, the reason why the operating system was last restarted, and the size of the RAM and how much of it remains free.

C functions involved in the start-up display (in addition to those mentioned in the above alphabetical listing) include `p_getlcd`, `p_getbat`, `p_getpsu`, and `p_getres`.

The final thing the HC Command Shell does, before starting to process commands from the user, is to attempt to capture the OFF key to itself.

CHAPTER 4

THE HC IN THE CRADLE

Introduction

The Psion Cradle was designed to provide:

- a secure mounting for the HC.
- hands-free operation.
- battery recharge.
- high speed data transfer with a PC.

The cradle automatically engages with the high speed serial port on the HC and can be connected via a high speed cable to a PC. Running special software on the PC enables a high speed serial connection that is significantly faster than the standard serial connections described elsewhere.

See the chapter *Introduction to the HC* for additional background details about the Cradle.

Port C

The Cradle contains an expansion socket that can accept some, but not all, of the HC's standard expansion modules. This expansion socket has name "Port C" as seen by software (the two standard HC ports have names "Port A" and "Port B".

For example, software that opens `TTY:C` will open any serial port in the Cradle expansion slot.

Link software can use a standard serial port fitted into the Cradle. The Link software must be invoked as follows:

```
link -p3 -b9600
```

This allows the Link software to operate at standard rates of data transfer, i.e. up to Baud 9600.

The remainder of this chapter describes various kind of higher speed connections that are possible between a PC and an HC. These require the expansion port of the Cradle to be fitted with a special high speed serial module. Note that this module will not operate if it is connected into either Port A or Port B of an HC - it has to be fitted into Port C.

Hardware connections

The high speed cable plugs into a special socket on an ASIC-2 expansion card fitted in the PC.

The remainder of this chapter assumes that the PC has an ASIC-2 expansion card fitted, and that the high speed cable connects into this card.

Fitting an ASIC-2 expansion card

An ASIC-2 expansion card in a PC contains two sockets:

- the upper one is designed to be connected to a (local) set of SSD drives
- the lower one is designed to be connected to a high speed cable leading to an HC Cradle.

The two possible uses of an ASIC-2 card in a PC are completely independent from each other. Any local SSD drive can be accessed as long as software device drivers such as *devram.sys* have been loaded by the *config.sys* program on the PC (the drivers *ifs.sys*, *fefs.sys*, and *devflash.sys* also need to be loaded to access Flash SSDs in these drives), none of these drivers are required for the high speed socket to work.

The ASIC-2 card occupies eight consecutive memory addresses, and uses one hardware interrupt. A set of jumpers on the card controls these two settings.

The standard ASIC-2 card works with MS-DOS versions 3.2 upwards. It is suitable for all PCs, XTS, ATs, and fully compatible computers. A variant of the card is also available for MCA-based computers, such as most PS/2 models.

Full details of installing and configuring the ASIC-2 card are contained in the documents *Installing the Psion SSD/ fast serial card for PCs* and *Installing and using the Psion SSD software and SSD drive unit for PCs* that accompany the ASIC-2 expansion card.

Software connections

There are two quite separate software mechanisms for connecting an HC in a Cradle to a PC with an ASIC-2 expansion card:

- running suitable Link software on each end of the connection, allowing high speed remote file access between the two computers
- using the `PMX:` device driver on the HC and the *hssram.sys* device driver on the PC, allowing RAM SSDs in the HC to be accessed from the PC as if they were SSD drives directly connected to the PC.

At the time of writing, the remote file access supported by the ASIC-2 card allows data transfer on average about four times faster than that possible using a standard RS232 serial connection between an HC and a PC. The PMX/HSS mechanism allows data transfer that is considerably faster than this. However:

- the PMX/HSS mechanism only allows access to RAM SSDs in the HC, not (at the time of writing) to Flash SSDs, nor to the "internal" drive (*m:*)
- the PMX/HSS mechanism only allows access to the HC SSD drives from the PC: there is no question of access to the PC drives from the HC.

Evidently, the two different mechanisms are both well-suited to different circumstances.

Check with Psion on the availability of a driver *hssflash.sys* allowing access to Flash SSDs in the HC.

High speed remote file access using Link software

In order for Link software on the HC and on the PC to use the high speed connection, the parameter

```
-stty:z
```

needs to be specified.

Thus at the HC end:

- any current Link software should be terminated, using the command `term link`.
- Link software should then be started (or restarted), using the command `link -stty:z`.

At the PC end, the same parameter should be passed on the command line to MCLink.

To check that a connection has successfully been established, simply type `dir rem::` at either end.

In both cases, other parameters on the command line (such as an explicit value for the Baud rate) will generally be ignored.

As is standard for Link and MCLink software, command line parameters can be specified *implicitly* by creating *.trm* files. For example, any parameters in a local file *mclink.trm* (as created by a `set` command inside MCLink) will apply, in the absence of any other contents on the command line. For more details, see the chapter *Mclink, Mcprint, and Slink* in the *Additional System Information* manual.

Version 3.0 or higher of MCLink is required, in order for the `-stty:z` parameter to be recognised.

High speed debugging using Link software

The Sibo Debugger can use a high speed connection to cut down on the time spent in data communication between the PC (where the Debugger runs) and the HC (where the program being debugged runs).

For general information about the Sibo Debugger, see the *Sibo Debugger* manual.

As always when using the Sibo Debugger, Link software has to be running on the HC. In order for the Link software to use the high speed connection, the parameter `-stty:z` has to be specified:

```
link -stty:z
```

The same parameter has to specified on the command line of the Debugger. Thus instead of typing e.g.

```
\sibossdk\sys\sdbg sample
```

to debug the program *sample.img*, the following should be typed:

```
\sibosdk\sys\sdbg -stty:z sample
```

The PMX/HSS mechanism

If the following line (or equivalent) is placed in the *config.sys* start-up program for a PC

```
device=c:\ssd\hhsram.sys
```

and the PC has an ASIC-2 expansion card fitted, the PC will gain four more disc drives.

If the PC ordinarily has floppy drives *a:* and *b:*, and a hard disk *c:*, then drives *d:* through *g:* will be added by this process.

However, typing e.g. `dir d:` at the MS-DOS command line would almost certainly lead to a message such as

```
Not ready reading drive D:
Abort, Retry, Fail
```

This is because `PMX:` software has not yet been enabled at the HC end of the connection.

Incidentally, two definite effects of running the *hssram* driver on the PC can clearly be seen, even in the absence of co-operating `PMX:` software on the HC:

- typing `dir d:` gives an error message of the above sort, whereas typing e.g. `dir h:` leads to the more cursory message `Invalid drive specification`
- if there is a *hardware* connection between the PC and the Cradle, and if there is an HC in the Cradle, the green "Data" light will flash when the `dir d:` command is given.

Configuring hssram.sys

The document *Installing and using the Psion SSD software and SSD drive unit for PCs* that accompanies the ASIC-2 card for PCs describes how the base address of the ASIC-2 card can be altered, by means of adjusting jumpers on the card.

This adjustment may occasionally be necessary, away from the default base address of `0x2a0` and hardware interrupt `7`, in order to avoid conflicts with other expansion cards already fitted in the PC (for example, network cards or internal modems).

In this case, as well as adjusting the jumpers on the ASIC-2 card, you will need to change the configuration of some of the associated software drivers.

When using an SSD drive unit with the ASIC-2 card, the software driver *devram.sys* needs to be reconfigured (if the jumpers are adjusted). This is fully described in the document *Installing and using the Psion SSD software and SSD drive unit for PCs*.

When using the high speed connection to a HC in a Cradle, it is the software driver *hssram.sys* that needs to be reconfigured (if the jumpers are adjusted). The new configuration is established in exactly the same way as for *devram.sys*, except that every reference to *devram.sys* has to be replaced by one to *hssram.sys*.

For example, to change the base address to `0x370` type the following:

```
c:
cd \ssd
config -a0x370 hssram.sys
```

In practice the default settings of the jumpers and of the ASIC-2 card should be suitable for the vast majority of PCs.

In case it is known to what base address the card should be set, but it is unclear how the jumpers should be set to effect this, simply run the *config* program specifying the required base address (and/or hardware interrupt number). The output of the *config* program specifies which of the nine jumpers on the card should be set.

The PMX: device driver

The following very simple program demonstrates the operation of the PMX: device driver:

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

LOCAL_C VOID GetKey(TEXT *mess)
    {
    p_printf("Press a key to %s PMX:",mess);
    p_printf("(PSION-ESC to terminate)");
    p_getch();
    }

GLDEF_C VOID main(VOID)
    {
    VOID *handle;

    FOREVER
        {
        GetKey("open");
        p_open(&handle,"PMX:",-1);
        GetKey("close");
        p_close(handle);
        }
    }
```

Basically, so long as the `PMX:` device is open by an application on the HC, any RAM SSDs in the HC will be inaccessible to the HC filing system. Attempting to read from these SSDs will give a "Not ready" error. Instead, these drives are given over to the control of any high speed serial requests from the PC.

Thus whilst the `PMX:` device is open, typing e.g. `dir d:` at the PC end of the connection will give a directory listing of the contents of any Ram SSD in drive *a:* of the HC. Likewise, typing `dir e:` at the PC will list the contents of any Ram SSD in drive *b:* of the HC (this assumes that *hssram.sys* has been installed on the PC and that there is only one hard disk partition on the PC).

When the `PMX:` device is closed again, the Ram SSDs in the HC come back under the aegis of the HC, and the familiar "Not ready" message will be given in response to an attempt to access these drives directly from the PC.

More details about PMX

The `PMX:` device driver has no interface other than the `p_open` and `p_close` functions used in the above code fragment.

It is possible for the `p_open` to fail, with the following errors:

`E_FILE_ALLOC`	failed to allocate memory for the control block
`E_GEN_INUSE`	the `PMX:` driver is already open (e.g. in another application), or the high speed port is already in use (e.g. by high speed Link)
`E_FILE_LOCKED`	same as the previous case.

The CRD device driver

The `CRD:` device driver, which is built into the ROM of the HC, can be used to report changes of state when an HC is inserted or removed from a Cradle.

The purpose of the `CRD:` device is to allow a program to perform specific operations automatically when the HC is inserted into a Cradle, and to "tidy up" when the HC is removed.

Note that the `CRD:` device does not have to be open for the Cradle expansion port to be used in any way. The operating system will automatically stop and start active devices in the Cradle, regardless of whether `CRD:` is open.

See the *HC Cradle and Holster* chapter of the *I/O Devices Reference* for more details of using the `CRD:` device in HC programs.

In fact, the `CRD:` device has a particularly simple interface (though not quite as simple as that of `PMX:`, described earlier in this chapter). The entirety of the functionality of `CRD:` when used with a Cradle is demonstrated by the following example program:

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

LOCAL_D WORD CradleStatus;
LOCAL_D WORD CradleStat;
LOCAL_D VOID *Cradle;

LOCAL_C VOID ReadCrdStatus(VOID)
    {
    p_ioc4(Cradle,P_FREAD,&CradleStat,&CradleStatus);
    }

GLDEF_C VOID main(VOID)
    {
    if (p_open(&Cradle,"CRD:",-1))
        {
        p_puts("Can't open CRD:");
        p_getch();
        p_exit(0);
        }
    ReadCrdStatus();
    FOREVER
        {
        p_iowait();
        p_puts(CradleStatus? "IN Cradle": "OUT of Cradle");
        ReadCrdStatus();
        }
    }
```

In practice, of course, there would be more than one event source in the application (the only events in the above example application are when the HC is inserted or removed from the Cradle).

CHAPTER 5

CUSTOMISING THE HC ROM

Introduction

HCs are shipped with a standard set of software programs in their rom. Application programs usually reside on SSDs which the user has to insert into the HC. These application programs generally rely on the ROM software in many ways, both direct and indirect.

For some purposes, however, it may be more suitable to alter the set of software programs that is on the ROM of the HC:

- programs run out of ROM have less of a RAM overhead than those run from an SSD
- programs in the ROM are physically more secure than those on an SSD, in the sense that an SSD can be removed by a user but the ROM cannot
- programs in the ROM may be able to take advantage of special software features inaccessible to programs on an SSD **-** for example, the fact that ROM code and data segments always remain at a fixed address
- programs in the ROM are easier to copy-protect.

All the different files comprising an HC ROM need to be assembled on a PC, and then combined into a special *master* file, with extension *.mas*. This process involves the Psion proprietary tool *erom.exe*.

The next step is to copy the master file onto a specially formatted SSD. This requires the use of an SSD drive attached either internally or externally to the PC, and the Psion proprietary tool *emast.exe*. The outcome of this is a so-called *master SSD.*

Finally, the *.mas* file can be transferred from the SSD into the ROM of an HC, by the procedure of *reprogramming* (or *reproing* for short).

Some cautionary remarks

The process of creating customised HC roms is not without its own considerable drawbacks:

- the sheer inconvenience of reproing every relevant HC, each time the customised ROM software is upgraded, has to be weighed against the simpler alternative of just copying new program files onto an external SSD
- reproing "in the field" is an impractical option, given that mains adaptors (which must be present for a repro to proceed) are unlikely to be present or usable in these circumstances
- a customised ROM may fail to be "future proof" in that future upgrades to the standard OS may reduce the free space in the ROM to the extent that additional custom software no longer fits
- again, a customised ROM may fail to be "future proof" against growth *within* the customised part of the ROM **-** bear in mind that program systems almost inevitably develop over time and grow in size as they develop
- a customer who damages an HC will find it is less convenient to have it replaced or repaired if it has been specially customised, than if it is a standard stock item

- if an unsuitable combination of files is combined into a *.mas* file, the outcome of reproing this onto an HC may be a totally useless HC, that has to be returned to Psion and taken apart before being capable of being used again (and note that HCs returned to Psion on account of a repro failure in these circumstances would count as having violated the standard warranty conditions).

Incidentally, in the last of these above cases, it may be possible to rectify the situation by means of putting another Window Server onto an SSD and rebooting the HC. This is because any program *sys$wsrv.img* that is found on the root directory of an SSD is started in preference to that in the rom. At a simpler level, putting an alternative shell (*sys$shll.img*) on an SSD and rebooting may also salvage matters.

Perhaps the largest drawback of all has not been mentioned so far. This is the possible effort required to produce software sufficiently small that it fits on the available space remaining in the HC rom. In practical terms, this may mean "optimising" and compressing code to the extent that it becomes unmaintainable or otherwise flawed. However, it may still be worthwhile putting *part* of a customised software system into the ROM of an HC, instead of *all* of it, so that at least some of the benefits mentioned earlier can be gained.

Creating an HC master file

Invoking erom

The program *erom.exe* is used to create the master file image of the HC rom. As so many parameters must be passed it is usually invoked via a short batch file. The following batch file *mrchv.bat* could be used:

```
erom >sch.mep -c -m -b0xa000 -v0x033e -lsch -oENG epocchp
type sch.mep
```

This batch file records its screen output to the file *sch.mep*, before printing the contents of this file onto the screen.

The meanings of the other parts of this batch file are as follows:

`-c`	the ROM is to be marked as suitable for reproing onto HC computers (as opposed to others in the Sibo range).
`-m`	the ROM image should be written to a *.mas* file.
`-b0xa000`	the ROM is to have base address `0xa000` in the address map.
`-v0x033e`	the version number of the ROM is to be `0.33`e.
`-lsch`	the files listed in the text file *sch.rom* are to be assembled into the rom.
`-oENG`	the notional language of the ROM is English.
`epocchp`	the ROM is to be based around the version of Epoc that is in the file *epocchp.exe*.

The master file produced by this batch file, if successful, would have name *v033eeng.mas*. This name is made up as follows:

- the first letter is always *v*.
- the next four letters are the version number (in this case *033e*).
- the final three letters are the notional language identifier.

Allowed values of language identifier include:

`ENG`	"English"
`FRN`	"French"
`GRM`	"German"
`SPA`	"Spanish"
`ITA`	"Italian"
`SWE`	"Swedish"
`DAN`	"Danish"
`DUT`	"Dutch"

In fact, *erom.exe* will fail if an unrecognised language identifier is specified. Note that the language thereby identified has a purely notional role, being announced only during the process of reproing, when the user is given a last chance to cancel before the ROM contents are changed (the actual value of language, as determined by software calling `p_getlanguage`, is set by the contents of one of the files in the rom).

Valid version numbers

See the documentation of `p_version` and `p_romversion` in the *Plib Reference* manual for some background details on valid version numbers.

Note that whereas roms produced by Psion are generally released with a version number ending in `f`, those produced by *erom.exe* are automatically constrained to a final letter in the range `a` to `e`. This is to help guard against any confusion between customised roms and those produced by Psion.

One other feature of the roms produced by *erom.exe* is that they always contain a zero-length file with the name *non$std.rom* (in *addition* to the files specified in the *sch.rom* file).

The files comprising the rom

In order to produce a standard HC rom, the contents of the list file *sch.rom* referred to by the batch file *mrchv.bat* would have to be as follows:

```
cheng.cfo,sys$ctry.cfo
wsrvhch1.img,sys$wsrv.img
corpshll.img,sys$shll.img
corpntfy.img,sys$ntfy.img
sys$env.ini
sys$rfsv.img
sys$ncp.img
link.img
mclinkpa.trm, mclink.trm
olib.dyl
big.fon
small.fon
mon_5x8.fon
mono.fon
sys$norm.fon
sys$bold.fon
exopl.img
oplch.dyl,opl.dyl
batchk.img
ttest.img
pprint.img
custom$.dat
```

The form of any line in this file is as follows:

```
<full path of the original name>[,<name by which the file should be called inside the
rom>]
```

These files have the following functions (see elsewhere in the *HC Programming Guide* for more details):

`cheng.cfo`	The standard English language "config file" for the HC (non-backlit variant)
`wsrvhch1.img`	The Window Server program for the HC
`corpshll.img`	The Command Shell
`corpntfy.img`	The original (non-Window Server) Notifier program for the HC
`sys$env.ini`	Initialisation data for the Window Server
`sys$rfsv.img`	The Remote File Server program
`sys$ncp.img`	The Networking Control Protocol program used by Remote Link
`link.img`	The Link program
`mclinkpa.trm`	Standard customisation data for Remote Link on the HC
`olib.dyl`	A dynamic library of object-oriented classes and methods

`*.fon`	Six different font files
`exopl.img`	A program facilitating the execution of Opl programs
`oplch.dyl`	The Opl dyl for the HC (implementing Opl/g)
`batchk.img`	Displays and monitors information about battery voltage levels
`ttest.img`	A utility program to test the status of the serial port
`pprint.img`	A utility program to print a specified file via a nominated peripheral
`custom$.dat`	An initially blank file that can be written to after reproing has finished, to allow additional once-only customisation.

Of these files, the only one that it is absolutely mandatory for the ROM to contain is a version of the config file, *sys$ctry.cfo*. All others can in principle be dispensed with, though some can be replaced more easily than others - as is discussed below.

Size considerations

The following extract from a standard *.mep* file (the "list" output of running *erom.exe*) may give some idea as to the current amount of free space in the HC rom:

```
CHENG.CFO              - B=B832 L=010FD(Hex),004349(Dec)
WSRVHCH1.IMG           - B=B942 L=085C0(Hex),034240(Dec)
CORPSHLL.IMG           - B=C19E L=03D30(Hex),015664(Dec)
CORPNTFY.IMG           - B=C571 L=007A0(Hex),001952(Dec)
SYS$ENV.INI            - B=C5EB L=00060(Hex),000096(Dec)
SYS$RFSV.IMG           - B=C5F1 L=004F0(Hex),001264(Dec)
SYS$NCP.IMG            - B=C640 L=02330(Hex),009008(Dec)
LINK.IMG               - B=C873 L=00AD0(Hex),002768(Dec)
MCLINKPA.TRM           - B=C920 L=000E2(Hex),000226(Dec)
OLIB.DYL               - B=C92F L=04828(Hex),018472(Dec)
BIG.FON                - B=CDB2 L=00BCE(Hex),003022(Dec)
SMALL.FON              - B=CE6F L=0093E(Hex),002366(Dec)
MON_5X8.FON            - B=CF03 L=0093E(Hex),002366(Dec)
MONO.FON               - B=CF97 L=00918(Hex),002328(Dec)
SYS$NORM.FON           - B=D029 L=0093E(Hex),002366(Dec)
SYS$BOLD.FON           - B=D0BD L=0093E(Hex),002366(Dec)
EXOPL.IMG              - B=D151 L=002C0(Hex),000704(Dec)
OPLCH.DYL              - B=D17D L=054C6(Hex),021702(Dec)
BATCHK.IMG             - B=D6CA L=00750(Hex),001872(Dec)
TTEST.IMG              - B=D73F L=00F40(Hex),003904(Dec)
PPRINT.IMG             - B=D833 L=00850(Hex),002128(Dec)
CUSTOM$.DAT            - B=D8B8 L=01800(Hex),006144(Dec)

Rom Base Segment is 0A000(Hex)
   Rom code size is 18150(Hex) 098640(Dec)
   Rom disk size is 22230(Hex) 139824(Dec)
   Free rom size is 05C60(Hex) 023648(Dec)
```

Note that the length of each file listed is given by the `"L"` value - first in hex, then in decimal - and the corresponding notional base address is given by the `"B"` value.

As can be seen, the amount of free space in the standard HC ROM is about 23k. However, this figure can be increased by omitting some of the files normally included.

Some possibilities for customisation

An alternative shell

Rather than using *corpshll.img*, a customised version of the shell program may be substituted.

This alternative shell can have any suitable name (eg *datashll.img*), so long as the extension is *.img* and the *.rom* file renames the shell to *sys$shll.img*.

See elsewhere in the *HC Programming Guide* for further details of writing a customised shell.

Variant config files

Replacing *cheng.cfo* with *chengel.cfo* changes from the non-backlit variant to the backlit variant of the keyboard table (in fact altering the value of the keycode returned to software when the middle of the three salmon-coloured keys on the top row is pressed).

Similarly:

chfrn.cfo	produces a ROM suited to the continental version of the keyboard, supporting some accented characters such as é
chswe.cfo	produces a ROM suited to the Scandinavian version of the keyboard, supporting characters such as æ
chnzl.cfo	produces a ROM suited to the numeric version of the keyboard.

There are also files *chfrnel.cfo, chsweel.cfo,* and *chnzlel.cfo,* which are backlit variants of *chfrn.cfo, chswe.cfo,* and *chnzl.cfo.*

Config files in *.cfo* format are produced from text format *.fig* files using the Psion proprietary tool *econfig.exe,* details of which are available upon request.

Additional files that might be added

As many additional program (or data) files can be added as will fit in the rom. To make room for these files, other files in the standard ROM may have to be omitted - see below.

Files that might be omitted

It is possible to omit any reference to *sys$ntfy.img* from the ROM file list provided that an alternative shell is used. This shell must make a suitable call on start-up to `wSystem` so that the Windows server version of the notifier is enabled. Alternatively suitable values must be written into *sys$env.ini* (see below).

The files *ttest.img* and *pprint.img* can each be omitted without any undue loss.

The file *custom$.dat* can be omitted if there is no intention to further customise individual HC roms afterwards. Alternatively, a shorter form of this file can be substituted (every single byte in this file must have the value `0xff`).

The file *batchk.img* can be omitted or replaced with alternative battery checking software. In this case, it is best not to use *corpshll.img,* as this emits an error message on start-up if it cannot locate and run a copy of *batchk.img.*

The files *exopl.img* and *oplch.dyl* can be omitted if there is no need to run Opl programs on the HC.

Some of the *.fon* files can be omitted, provided due care is paid to provide a suitably adjusted *sys$env.ini* (see below). For example, the MC-derived fonts *big.fon, small.fon,* and *mono.fon* could be omitted, leaving only *mon_5x8.fon* and the Series3-derived fonts *sys$norm.fon* and *sys$bold.fon.* Note that the order of listing *.fon* files in the *.rom* file defines which fonts correspond to which Window Server font ids - `WS_FONT_BASE` specifying the first *.fon* file in the *.rom* listing, and so on. Note also that the HC console (as used by C programs containing statements like `puts` or `p_printf`, and also by `print` statements in Opl) presupposes the use of the third *.fon* file listed, so that this should always be mono-spaced and of size 5 by 8.

Customising the Window Server

Whenever the system restarts, the Window Server reads the contents of *sys$env.ini,* and sets environment variables according to the data therein.

Dumping the contents of the standard *sys$env.ini* will confirm that a particularly simple format is used in this file:

```
[<byte count><name><byte count><value>]
```

with this pattern being repeated as many times as there are environment variables to initialise. For each such environment variable, the byte count preceding the environment variable name gives the length of the name, and the byte count preceding the environment variable value gives the length of the value.

The only environment variables that you *need* to *consider* defining in *sys$env.ini* are the following:

`$WS_SF`	gives (in two bytes) the index number of the *.fon* file to use as the "system" font, which is the default font for all drawing via any GCs (graphics contexts). This will usually be left at value `0`, and as such can be omitted from *sys$env.ini*.
`$WS_IF`	gives (in two bytes) the index number of the *.fon* file to use as the "internal" font, which is what the Window Server uses when drawing alerts, busy messages, and information messages. This has the value `4` in the standard *sys$env.ini*, thereby specifying *sys$norm.fon* as the internal font. In case an alternative *.rom* file omits *big.fon* and moves *sys$norm.fon* into this slot, the value of `$WS_IF` should be adjusted to `0`.
`$WS_FL`	gives (in two bytes) the initial value of the Window Server flags. This is zero in the standard *sys$env.ini*.

See the *System start-up* section in the *Window Server Reference* manual for more details of the possible Window Server flags. In many cases, the initial value `0x0a` will be appropriate, setting the flags `_NO_NOTIFIER_REBOOT` and `_HOOK_NOTIFIER`.

Creating and using a master SSD

Once a suitable *.mas* file has been created, the next step is to transfer it, together with the repro software, onto a master SSD. This requires an SSD drive to be attached to the PC, although no special software drivers (such as *fefs.sys* and *devflash.sys*) need to be installed.

Anyone ordering an SSD drive for their PC should note that, for it to function, not only is the drive itself required, but also an ASIC-2 expansion card, and (in the case of an external SSD drive) a suitable connecting cable.

The master SSD is usually created under the control of a batch file, of which the following (*makemast.bat*) is an example:

```
@echo off
emast -u1 v033eeng.mas
echo  Transferring other software...
xcopy \hcmast\*.* f:\*.* /s
```

This potentially copies a whole directory tree onto the master SSD, i.e. the contents of *\hcmast* including subdirectories. In this case, the contents of *\hcmast* would include *repro.app*.

The batch file assumes that the PC sees the first SSD slot as drive `f:`, and would need to be altered if this is not the case (changing `f:\` to eg `e:\`). The batch file also assumes that all relevant software drivers for the external SSD drives have been loaded.

The `-u1` parameter to *emast* specifies that the top left SSD slot in the attached SSD drive is to be used. The filename passed to *emast* obviously has to match that of the master file created earlier by *erom*.

Once the batch file has finished, the SSD can be used to repro HCs in the normal way.

More details on master SSDs

A master SSD must be a 512k Flash device and has a very special format: part of it is devoted to a "file" that is outside the filing system proper, and the remainder (just under 256k) is presented to the outside world as if it were the entirety of the SSD.

That is, normal file operations do not see the *.mas* "file" that is on the SSD. This is why the *.mas* file has to be copied onto the SSD using a special tool, i.e. *emast.exe*. This tool not only copies on the *.mas* file but also specially formats the remainder of the SSD.

To repro numeric keyboard HCs

To prepare a master SSD that can be used to repro an HC with a numeric keyboard (and which therefore lacks keys such as `R`, `E`, `P`, and `O`), a copy of *repro.app* should be renamed to *y0n0.img* before being copied onto the master SSD.

Files required

This section lists the files from the Optional Disk of the SDK that are needed, in order to be able to produce a customised ROM for the HC:

source files:	*mrchv.bat*, *sch.rom*, *epochhp.exe*, *repro.app*, *makemast.bat*, and the 22 files listed above as the standard contents of a *.rom* file, i.e. *cheng.cfo* through *custom$.dat*, together with the other 7 standard *.cfo* files
tools:	*erom.exe* and *emast.exe*.

APPENDIX A

TECHNICAL SPECIFICATIONS

Psion's continuing product development and improvement programs mean that specifications and features are subject to change at any time and without notice.

Psion Solid State Disks Technical Specification

Dimensions

Size:	63mm (length) x 52mm (width) x 6mm (height)
Weight:	≈25g

Capacities

RAM:	128KB, 512KB, 1MB, 2MB
Flash:	128KB, 256KB, 512KB, 1MB, 2MB, 4MB, 8MB
PSRAM:	512KB, 1MB, 2MB
Solo Flash:	128KB, 256KB, 512KB

Filing System

Flash, RAM and PSRAM:	MS-DOS
RAM and PSRAM:	FAT directory/file system

Interface

Physical:	6 pin serial
Electrical:	Clock, 0V, V_{backup}, V_{pp}, V_{cc}, Data

Data Transfer

SSD interface:	320Kbytes/sec

File Access

Note: the quoted rates for the Series 3c also apply to the Siena with external SSD drive

All SSD types read:	30-40Kbytes/sec depending on SSD/directory structure (HC and Series 3c)
Flash write:	≈6Kbytes/sec (HC), ≈8Kbytes/sec (Series 3c)
RAM and PSRAM write:	30-40Kbytes/sec depending on SSD/directory structure

Formatting

Flash:	≈30 secs per 128K (HC), ≈20 secs per 128K (Series 3c); 9,999 times minimum
Format/write voltage:	12V DC
Programming voltage:	15.0V to 18.0V DC on the Vh pin @ 40mA max. (type II flash)
RAM:	≈7.5 secs per 128K (HC), ≈6 secs per 128K (Series 3c)

Power

Flash, standby:	500µA
RAM, standby:	10µA
Flash and RAM, reading:	1mA
Flash, writing:	20-30mA
RAM, writing:	≈1mA

PSRAM versus SRAM SSDs

Important: Pseudo-Static RAM (PSRAM) SSDs are **not** recommended for use in consumer machines, (Series 3a, Series 3c and Siena). This section explains why.

PSRAM SSD's are *not suitable* for use with some machines:

Machine	**Compatible**
HC	Yes (with an upgrade - see below)
HC-DOS	Yes (with an upgrade - see below)
Work*about*	Yes
MC (all variants)	No
Series 3 (all variants)	No
Series 3a 256K, 512K	No
Series 3a 1M, 2M (S3m)	Yes (see below)

The fact that PSRAM SSDs can only be used with the 1MB and 2MB Series 3a is effectively a *No* to the Series 3 range, because of the problem of having to differentiate the various Series 3 models at point of sale. PSRAM SSDs are therefore only recommended for use in Psion Industrial’s handhelds.

An upgrade to both HC and HC-DOS machines (involving a component change on the main PCB only) is in progress. HC and HCDOS machines supporting PSRAM SSDs will be identifiable by their serial number. The performance of the upgraded machines is not affected in any other way, including power consumption and use with any peripherals and other SSD types.

All SSD's have a common serial interface which sets the data transfer rate. This means that for reading and writing data, all types of RAM SSDs work at the same speed.

The power consumption of a PSRAM chip is generally higher than for the equivalent Static RAM (SRAM) chip when read/writing data. However, both types of memory only transfer data during a small part of the SIBO cycle so the power consumption of an active SSD is not much higher for a PSRAM as opposed to an SRAM.

However, a PSRAM SSD consumes a significant amount of current when the host machine is turned on, even though no data is being transferred. This is because an oscillator is required to refresh the memory. This can increase the overall current consumption of the machine by up to a third. Also, when the host machine is off, the backup current of the PSRAM SSD is much higher than the equivalent SRAM SSD. A lithium cell in a 2MB PSRAM will have a life of about 17 days (400 hours) outside a SIBO machine. PSRAM SSDs are best considered as memory expansion rather than removable media and are most suited to applications where they remain inside a machine; then they only rely on their backup batteries when the machine's main battery is changed.

PSRAM SSDs should therefore only be used in applications where the above disadvantages have little affect. An example is a Work*about* which is mostly powered/recharged through insertion into a docking station and whose SSDs are never removed.

Psion HC Technical Specification

Models

Psion HC100:	128K CMOS static RAM.
Psion HC110:	256K CMOS static RAM.
Psion HC120:	512K CMOS static RAM.

All models have 256K internal Flash ROM.

Processor

Type:	80C86-compatible 16-bit processor.
Clock:	3.84MHz.

Dimensions

Size:	200mm (length) x 80mm (width) x 35mm (height).
Weight:	395g (540g with batteries but no SSDs).

Environmental

Temperature:	Operating 0C to +50C, storage -20C to +70C.
Humidity:	Operating 90% max non-condensing.
Weatherproofing:	IP54. Splashproof (depending on variant).
Drop resistance:	1 metre onto concrete.
EMC:	FCC Class B; CE marked, E-marked.
Safety:	EN60950.

Software

Operating system:	Psion EPOC multitasking OS.
Command shell:	MS-DOS like command interpreter
Communications:	Psion LINK, 50-9600 baud, asynchronous, compatible with MCLINK, RCOM & PSIWIN software on remote PC.
Printing:	Parallel and serial and via remote PC.

Solid State Disks

Built-in drives:	Two SSD drives.
SSD capacities:	Flash: 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, 8MB RAM: 128KB, 512KB, 1MB, 2MB.
Filing system:	MS-DOS compatible.

File Access

Note: the quoted rates apply only to the HC.

Flash read:	30-40Kbytes/sec depending on SSD/directory structure.
RAM read:	30-40Kbytes/sec depending on SSD/directory structure.
Flash write:	≈6Kbytes/sec.
RAM write:	30-40Kbytes/sec depending on SSD/directory structure.

Formatting

Flash:	≈30 secs per 128K.
Format/write voltage:	12V DC.
RAM:	≈7.5 secs per 128K.

Screen

Type:	Black and white retardation film LCD, optional back lighting.
Resolution:	160 X 80 pixels, 26 characters x 9 lines (default font).
Dimensions:	60mm (width) x 50mm (height).

Keyboard

Alphanumeric:	53 key UK/US, European and Scandinavian versions.
Numeric:	31 key with function keys.
Custom:	Can be provided.

Sound

Built-in:	Piezo buzzer (single tone) and loudspeaker.

Power

Main battery:	NiCad 500mAH or 600mAH rechargeable pack.
Back-up:	CR1620 3V lithium cell.
External:	12V DC via Psion adaptor.
Battery life:	Typically up to 50 hours, depending on use and configuration.

Expansion

Capabilities:	Two expansion module interfaces.
Modules:	RS232/Parallel; Quad Modem; MCR/RS232/TTL-RS232; Bar Code Reader; RS232/TTL-RS232; RS232/Bar Code Reader; Printer; LIF-PFS/Barcode; LIF-PFS/TTL-RS232; 16550 RS232/TTL-RS232; Vehicle/TTL-RS232; Integral Laser Scanner.

Psion HC RS232/Parallel (printer) module, version 1 Technical Specification

Important Notice - Compatibility

This module can continue to be used in both top and bottom ports of the HC. Applications requiring serial or parallel comms from the HC should use this module.

This module can be used in the Work*about* Docking Station for serial or parallel communications.

This module must not be used in an HC Docking Station sold in countries requiring the CE Mark. The version 2 Psion HC RS232/Parallel (printer) module must be used instead.

Physical

Part number:	1502-0001
Module:	Integrated removable module.
HC compatibility	Yes. Fits into either of the HC's expansion ports.
HC-DOS compatibility	Yes
Docking station compatibility	Not CE marked configuration - version 2 model required
EMC:	FCC Class B, CE-mark and E-mark**not** CE marked for use with HC Docking Station
Safety	EN60950

Connectors

RS232:	9 way miniDIN female.
Parallel:	Standard Centronics 25 way D female

Serial Interface

Baud:	50, 75, 11, 134, 150, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200, 9600.
Data bits:	5, 6, 7, 8 (ASCII).
Stop bits:	1, 2.
Parity:	Odd, even, none.
Handshaking:	XON/XOFF, RTS/CTS, DSR/DTR, DCD.
Remote switch-on	Via DSR line (optional).
Protocols:	Psion proprietary MCLINK protocol. Xmodem protocol.

RS232 interface

Provides standard RS232 level signals and is similar to the RS232 interface on an IBM AT. The difference between this socket and an IBM AT is that the RI (ringing indicator) pin is not implemented. This pin on the HC can be optionally connected to the HC VSUP input/output supply via an on board link.

RS232 , (9 way male D-type), pinout

Pin 1:	DCD input.
Pin 2:	RX input.
Pin 3:	TX output.
Pin 4:	DTR output.
Pin 5:	Ground (0V).
Pin 6:	DSR input.
Pin 7:	RTS output.
Pin 8:	CTS input.
Pin 9:	Optional VSUP input/output, (7-10V DC). DC input must not exceed 10V.

Parallel Interface

Parallel Interface pinout

Pin 1	Strobe output
Pin 2	Data 0 output
Pin 3	Data 1 output
Pin 4	Data 2 output
Pin 5	Data 3 output
Pin 6	Data 4 output
Pin 7	Data 5 output
Pin 8	Data 6 output
Pin 9	Data 7 output
Pin 10	ACK input
Pin 11	BUSY input
Pin 12	PE input
Pin 13	NC
Pin 14	AUTO FD XT output
Pin 15	ERROR input
Pin 16	INIT output
Pin 17	SLCT IN output
Pins 18-25	Ground 0V

Psion HC RS232/Parallel (printer) module, version 2 Technical Specification

Important Notice - Compatibility

This module must be used in the HC Docking Station for countries requiring the CE Mark.

This module is compatible with all configurations of Psion HC, HC Docking Station and Work*about* Docking Station and is recommended for all new installations.

Physical

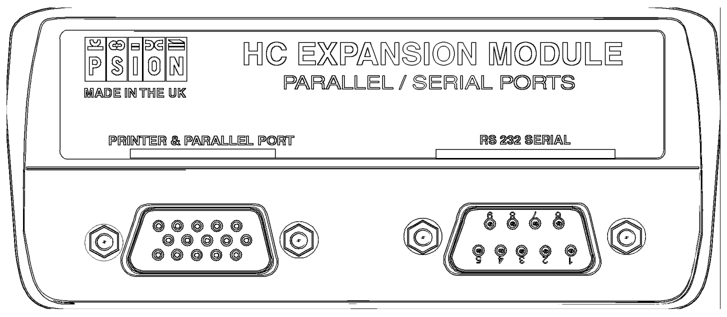
Part number:	1502 0052 10
Module:	Integrated removable module.
HC compatibility	Yes. Fits into either of the HC's expansion ports.
HC-DOS compatibility	Yes
Docking station compatibility	Yes
EMC:	FCC Class B, CE-mark and E-mark
Safety:	EN60950

Connectors

RS232:	9 way male D-type (RS232, PC AT type).
Parallel:	15 way High Density D male. A 15 Way to 25 Way converter cable is required for connection to a standard Centronics port.

Serial Interface

Baud:	50, 75, 11, 134, 150, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200,
Data bits:	9600.
Stop bits:	5, 6, 7, 8 (ASCII).
Parity:	1, 2.
Handshaking:	Odd, even, none.
Remote switch-on	XON/XOFF, RTS/CTS, DSR/DTR, DCD.
Protocols:	Via DSR line (optional).
	Psion proprietary MCLINK protocol.
	Xmodem protocol.



RS232 interface

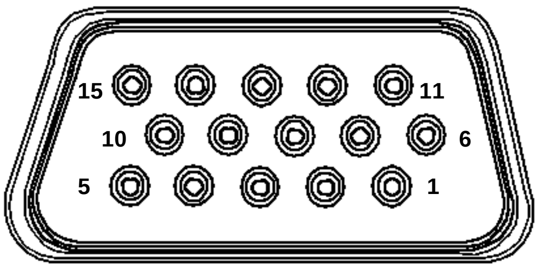
Provides standard RS232 level signals and is similar to the RS232 interface on an IBM AT. The difference between this socket and an IBM AT is that the RI (ringing indicator) pin is not implemented. This pin on the HC can be optionally connected to the HC VSUP input/output supply via an on board link.

RS232 , (9 way male D-type), pinout

Pin 1:	DCD input.
Pin 2:	RX input.
Pin 3:	TX output.
Pin 4:	DTR output.
Pin 5:	Ground (0V).
Pin 6:	DSR input.
Pin 7:	RTS output.
Pin 8:	CTS input.
Pin 9:	Optional VSUP Input/output, (7-10V DC). DC input must not exceed 10V.

15 way High Density Parallel Interface

15 way High Density Parallel socket (male)



15 way High Density Parallel Interface pinout

Pin 1	Strobe output
Pin 2	Data 0 output
Pin 3	Data 1 output
Pin 4	Data 2 output
Pin 5	Data 3 output
Pin 6	Data 4 output
Pin 7	Data 5 output
Pin 8	Data 6 output
Pin 9	BUSY input
Pin 10	ERROR input
Pin 11	INIT output
Pin 12	SLCT IN output
Pin 13	Data 7 output
Pin 14	PE input
Pin 15	Ground 0V

Psion 15 Way to 25 Way converter cable Technical Specification

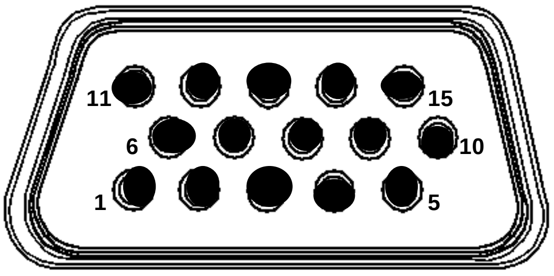
Physical

Part number:	2403 0026 01
Length:	30cm.
EMC:	FCC Class B, CE-mark and E-mark
Safety:	EN60950

Connectors

15-way parallel:	female; for connection to the 15-way high-density parallel (printer) socket on the Psion HC RS232/Parallel (printer) module, version 2
25-way parallel:	for connection to a standard Centronics port on a printer.

15 way High Density Parallel plug (female)



15 way High Density Parallel Interface connector pinout

Pin 1	Strobe output
Pin 2	Data 0 output
Pin 3	Data 1 output
Pin 4	Data 2 output
Pin 5	Data 3 output
Pin 6	Data 4 output
Pin 7	Data 5 output
Pin 8	Data 6 output
Pin 9	BUSY input
Pin 10	ERROR input
Pin 11	INIT output
Pin 12	SLCT IN output
Pin 13	Data 7 output
Pin 14	PE input
Pin 15	Ground 0V

25-way connector pinout

Pin 1	Strobe output
Pin 2	Data 0 output
Pin 3	Data 1 output
Pin 4	Data 2 output
Pin 5	Data 3 output
Pin 6	Data 4 output
Pin 7	Data 5 output
Pin 8	Data 6 output
Pin 9	Data 7 output
Pin 10	NC
Pin 11	BUSY input
Pin 12	PE input
Pin 13	NC
Pin 14	NC
Pin 15	ERROR input
Pin 16	INIT output
Pin 17	SLCT IN output
Pins 18-25	Ground 0V

Psion HC MCR /RS232 /TTL RS232 module, (Version 2), Technical Specification

Physical

Part number:	1502-0003
Module:	Integrated removable module.
HC compatibility	Yes. Fits into either of the HC's expansion ports.
HC-DOS compatibility	No
Docking station compatibility	Yes
Certification:	FCC Class B VDE Class B

MCR Interface

Connections

Socket:	7 way locking miniDIN female.
Readers supported:	Single or simultaneous two track reader.
MCR unit power supply:	5V DC output available to power MCR unit. This is software switchable.

Pinout

Plug required:	Hosiden type TCP6170-1100 or equivalent.
Pin 1:	5V output (100mA max).
Pin 2:	CLD. Card load input (low when card in reader). 100k pullup to 5V.
Pin 3:	DATA1. Data input for track 1 reader. Data is read in on falling edge of the clock line. 100k pullup to 5V.
Pin 4:	CLOCK1. Clock input for track 1 reader. 100k pullup to 5V.
Pin 5:	DATA2. Data input for track 2 reader. Data is read in on falling edge of the clock line. 100k pullup to 5V.
Pin 6:	CLOCK2. Clock input for track 2 reader. 100k pullup to 5V.
Pin 7:	Ground. (0V).

RS232 / RS232 TTL Interface

Connections

The single RS232 interface can be software switched between 2 sockets.

Sockets:	8 way locking miniDIN socket	(RS232 TTL).
	9 way miniDIN socket	(standard RS232).

Interface

Baud:	50, 75, 11, 134, 150, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200,
Data bits:	9600.
Stop bits:	5, 6, 7, 8 (ASCII).
Parity:	1, 2.
Handshaking:	Odd, even, none.
Remote switch-on	XON/XOFF, RTS/CTS, DSR/DTR, CDC.
Protocols:	via DSR line (optional).
	Psion proprietary MCLINK protocol.
	Xmodem protocol.

RS232 TTL socket

TTL levels:	0-5V.
TTL signals:	TX, RX, RTS, CTS, DSR., plus software switchable unregulated 7-10V DC and regulated 5V DC.
TTL polarity:	Programmable in software.
Readers supported:	This interface is intended for use with peripherals such as low power laser and CCD bar code scanners which support a TTL level RS232 interface.
Scanner power supply:	Unregulated 7-10V DC and regulated 5V DC to power the scanner. These are software switchable.

RS232 TTL socket pinout

Plug required:	Hosiden type TCP6180-1100 or equivalent.
Pin 1:	5V DC regulated output. (250mA max*).
Pin 2:	TX output. TTL transmit.
Pin 3:	RTS output. TTL handshaking.
Pin 4:	VSUP output. Unregulated 7-10V DC output (250mA max*).
Pin 5:	RX input. TTL receive.
Pin 6:	CTS input. TTL handshaking.
Pin 7:	DSR input. TTL handshaking.
Pin 8:	Ground. (0V).

*The maximum **combined** current must not exceed 250mA

Standard RS232 interface

Provides standard RS232 level signals and is similar to the RS232 interface on an IBM AT. The difference between this socket and an IBM AT is that the RI (ringing indicator) pin is not implemented. This pin on the HC can be optionally connected to the HC VSUP input/output supply via an on board link.

RS232 , (9 way male D-type), pinout

Pin 1:	DCD input.
Pin 2:	RX input.
Pin 3:	TX output.
Pin 4:	DTR output.
Pin 5:	Ground (0V).
Pin 6:	DSR input.
Pin 7:	RTS output.
Pin 8:	CTS input.
Pin 9:	Optional VSUP Input/output, (7-10V DC). DC input must not exceed 10V.

Psion HC RS232 /TTL RS232 module, Technical Specification

Note: a 16550 RS232/TTL-RS232 module is also available and is described later in this Appendix.

Physical

Part number (IP64):	1502-0039
Part number (non-IP64):	1502-0040
Module:	Integrated removable module.
HC compatibility	Yes. Fits into either of the HC's expansion ports.
HC-DOS compatibility	Yes
Docking station compatibility	Yes

EMC:	FCC Class B, CE-mark and E-mark
Safety	EN60950
Weatherproofing:	IP64 (depending on model, see part number above)

Connections

The single RS232 interface can be software switched between 2 sockets.

Sockets:	9 way female D-type	(RS232 TTL).
	9 way male D-type	(RS232, PC AT type).
Peripherals supported:	The TTL RS232 interface is intended for use with peripherals such as low power laser and CCD bar code scanners which support a TTL level interface.	

RS232 interface

Baud:	50, 75, 11, 134, 150, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200,
Data bits:	9600.
Stop bits:	5, 6, 7, 8 (ASCII).
Parity:	1, 2.
Handshaking:	Odd, even, none.
Remote switch-on	XON/XOFF, RTS/CTS, DSR/DTR, CDC.
Protocols:	via DSR line (optional).
	Psion proprietary MCLINK protocol.
	Xmodem protocol.

RS232 TTL interface

TTL levels:	0-5V.
TTL signals:	TX, RX, RTS, CTS, DSR., plus software switchable.
TTL polarity:	Programmable in software.
Power supply outputs:	Software switchable unregulated 7-10V DC, unswitched unregulated 7-10V DC and regulated 5V DC.
Power supply inputs:	The unswitched 7-10V pin is directly connected to the HC main power rail and can therefore be used to power the HC. To do this the supply coming into the HC must be diode isolated (so as not to take power from the HC) and in the range 7-10V (10V maximum). **NOTE:** powering the HC from this pin will **not** charge the HC internal battery.

RS232 interface

Provides standard RS232 level signals and is similar to the RS232 interface on an IBM AT. The difference between this socket and an IBM AT is that the RI (ringing indicator) pin is not implemented. This pin on the HC can be optionally connected to the HC VSUP input/output supply via an on board link. This is described above.

RS232 , (9 way male D-type), pinout

Pin 1:	DCD input.
Pin 2:	RX input.
Pin 3:	TX output.
Pin 4:	DTR output.
Pin 5:	Ground (0V).
Pin 6:	DSR input.
Pin 7:	RTS output.
Pin 8:	CTS input.
Pin 9:	Optional VSUP Input/output, (7-10V DC). DC input must not exceed 10V.

RS232 TTL interface

RS232 TTL socket pinout

Pin 1:	VSUP switched output. Unregulated 7-10V DC output (250mA max*).
Pin 2:	RX input. TTL receive.
Pin 3:	TX output. TTL transmit.
Pin 4:	5V output. (250mA max*).
Pin 5:	Ground (0V).
Pin 6:	DSR input. TTL handshaking.
Pin 7:	RTS output. TTL handshaking.
Pin 8:	CTS input. TTL handshaking.
Pin 9:	VSUP Input/output. 7-10V DC input/output. DC input must not exceed 10V.

*The maximum **combined** current must not exceed 250mA

Psion HC 16550 RS232 /TTL-RS232 module, Technical Specification

Note: an ASIC5 based RS232/TTL-RS232 module is also available, and is described above.

This module may be used with any standard HC (or HCDOS machine) for faster data transfer rates than the standard module. The essential difference between this module and the standard "RS232 / TTL-RS232" module is that a 16550 UART is used rather than the ASIC5 UART used in the standard module. Also the RI (Ringing Indicator) function is provided to make it a true IBM PC-AT RS232 interface (note that the HC software does not make any use of RI).

Physical

Part number	1502-0045
Module:	Integrated removable module. Fits into either of the HC's expansion ports.
Operating temperature	-20 to +60 °C
Storage temperature	-20 to +60 °C
Weight	60g
HC compatibility	Yes, loadable PDD required
HCDOS compatibility	Yes
Docking Station compatibility	No
Emissions	FCC class A

Connections

The single RS232 interface can be software switched between 2 sockets.

Sockets:	9 way female D-type	(RS232 TTL).
	9 way male D-type	(RS232, PC AT type; full EIA-232 signal levels).

Peripherals supported: The TTL RS232 interface is intended for use with peripherals such as low power laser and CCD bar code scanners which support a TTL level RS232 interface. The polarity of this interface can also be programmed to be standard (non-inverting) or inverting.

RS232 interface

Baud:	50, 75, 11, 134, 150, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200, 9600, 19200; also 38400 using error corrected transmission.
Data bits:	5, 6, 7, 8 (ASCII).
Stop bits:	1, 2.
Parity:	Odd, even, none.
Handshaking:	Xon/Xoff, RTS/CTS, DSR/DTR, CDC.
Remote switch-on	via DSR line (optional).
Protocols:	Psion proprietary MCLINK protocol. Xmodem protocol.

RS232 TTL interface

TTL levels:	0-5V.
TTL signals:	TX, RX, RTS, CTS, DSR., plus software switchable.
TTL polarity:	Programmable in software.
Power supply outputs:	Software switchable unregulated 6-10V DC, unswitched unregulated 6-10V DC and switched regulated 5V DC.
Power supply inputs:	The unswitched 7-10V pin is directly connected to the HC main power rail and can therefore be used to power the HC. To do this the supply coming into the HC must be diode isolated (so as not to take power from the HC) and in the range 7-10V (10V maximum). **NOTE:** powering the HC from this pin will **not** charge the HC internal battery.

HC usage

A loadable software Physical Device Driver (PDD) is available to allow this module to be used on any standard HC. Once this driver is loaded the module is accessed exactly the same as the current ASIC5 based RS232/TTL-RS232 module.

When fitted it allows the HC to reliably communicate at 19200 baud (the standard module is only reliable up to 9600 baud). However when using error corrected protocols such as LINK, communicating at 38400 baud is feasible. In tests file transfer rates in excess of 2Kbytes/second have been achieved using LINK in conjunction with MCLINK on a 486 PC.

Docking station usage

The HC/HCDOS communicates with this module via the parallel expansion bus, therefore it cannot be plugged into a Docking Station.

HCDOS usage

Baud rates of up to 115,200 are achievable compared with a maximum of 19,200 for the standard module.

Connector selection and TTL polarity selection will normally be under application control but the HCSETUP utility can be used. The port appears as COM1 if the module is plugged into the top HCDOS expansion slot or COM2 in the bottom slot.

RS232 interface

Provides standard RS232 level signals and is similar to the RS232 interface on an IBM AT. The difference between this socket and an IBM AT is that the RI (ringing indicator) pin can be optionally connected to the HC VSUP input/output supply via an on board switch. This is described below.

RS232 , (9 way male D-type), pinout

Pin 1:	DCD input.
Pin 2:	RX input.
Pin 3:	TX output.
Pin 4:	DTR output.
Pin 5:	Ground (0V).
Pin 6:	DSR input.

Pin 7:	RTS output.
Pin 8:	CTS input.
Pin 9:	RI input ***or***
	VSUP Input. 7-10V DC input. DC input must not exceed 10V. ***or***
	VSUP Output. 6-10V DC unregulated. 250mA maximum current, or 200mA maximum current if another expansion module is also being powered.
	See below for full details.

Pin 9 RI / VSUP switch

If you hold the module with the component side of the PCB facing you and the D-type connectors at the top, this switch is located below the right hand D-type connector.

Important: the normal position for this switch is 'RI' (in the left hand position), it should only be switched to the 'VSUP' position for special applications as described below otherwise damage could occur to either the HC or the device connected at the other end.

The 'VSUP' connection should be selected **only** if one of the following is required:

1. The HC is to be powered externally. The supply applied to the HC must be diode isolated (to prevent power drain from the HC) and in the range **7-10V** (10V maximum).
   NOTE: powering the HC from this pin will **not** charge the HC internal battery.

2. The HC is required to supply power to another device, for example certain true RS232 I/F laser scanners require power to be supplied from the RS232 connector. Note that VSUP is not software switchable and is present even when the HC is switched off, so the device would need its own ON-OFF switch to prevent the HC battery being drained when the device is not in use.
   VSUP is an unregulated supply in the range 6-10V. The device powered from pin 9 should not draw more than **250mA** from VSUP if no other expansion modules are powered up simultaneously (if another expansion module is fitted and powered up, the current drawn should not exceed **200mA**).

DSR auto-wakeup switch

If you hold the module with the component side of the PCB facing you and the D-type connectors at the top, this switch is located to the bottom left hand corner of the PCB. If this switch is ON (in the right hand position) the HC is automatically turned on when DSR is asserted by the device connected to the RS232 port.

Power consumption

When the RS232 port is open it typically draws **8mA** plus the current drawn by the device connected at the other end, this will vary depending on the device, for example connected to a PC the total current drawn will increase to typically 18mA (this however will vary from one PC to another).

TTL interface

RS232 TTL socket pinout, (9 way female D-type)

Pin 1:	VSUP switched output. Unregulated 6-10V DC output (250mA max*).
Pin 2:	RX input. TTL receive.
Pin 3:	TX output. TTL transmit.
Pin 4:	Switched 5V output. (250mA max*).
Pin 5:	Ground (0V).
Pin 6:	DSR input. TTL handshaking.
Pin 7:	RTS output. TTL handshaking.
Pin 8:	CTS input. TTL handshaking.
Pin 9:	VSUP Input. 7-10V DC input. DC input must not exceed 10V. ***or***
	VSUP Output. 6-10V DC unregulated. 250mA maximum current*, or 200mA maximum current* if another expansion module is also being powered.

*The maximum **combined** current must not exceed 250mA

Switched VSUP (pin 1) and 5V (pin 4) outputs

Both these switched power supply outputs are provided to power peripherals plugged into the TTL connector and both are enabled only when the port is open and TTL connector is selected.

VSUP is an unregulated supply in the range 6-10V.

The 5V regulated output has a tolerance of +/- 5%.

The device powered from either supply should not draw more than **250mA** if no other expansion modules are powered up simultaneously (if another expansion module is fitted and powered up, the current drawn should not exceed **200mA**). If current is drawn from both rails then the combined current should not exceed 250mA (or 200mA if another module is present).

VSUP direct connection (pin 9)

A direct VSUP connection is also provided for the same purposes as the VSUP option on pin 9 of the RS232 connector, see description above.

Power consumption

When the port is open and the TTL interface is selected, the interface typically draws **5mA.** In most cases however the current drawn by the peripheral device dominates.

Psion HC Bar Code Reader module, (Version 2), Technical Specification

Physical

Part number (HP wand):	1502-0020
Part no. (Welch Allen wand):	1502-0021
Module:	Integrated removable module.
HC compatibility	Yes. Fits into either of the HC's expansion ports.
HC-DOS compatibility	No
Docking station compatibility	No
Certification:	FCC Class A VDE Class B

Connection

Socket:	6 way locking miniDIN socket.
Peripherals supported:	Will support most standard bar code wands and also scanners with wand emulation output. It is supplied with one of either: HP wand HPBCS-A207 and plug **or** Welch Allen wand and plug
Remote switch-on:	Facility for remotely switching machine on with bar code wand or scanner.
Power supply output:	5V DC available to power a wand or scanner. This is software switchable.

Pinout

Plug required:	Hosiden type TCP6160-1100 or equivalent.
Pin 1:	EXON. Turns HC on when pulled low. 100k pullup to 5V present at all times (even if machine is switched off)
Pin 2:	Enable output. Optional output to barcode device. Under software control (application dependant).
Pin 3	Switch input. Optional input for barcode switch.
Pin 4:	Data input. Input from barcode wand. 2k2 pullup to 5V when reading from wand.
Pin 5:	5V output. (Max 250mA)
Pin 6:	Ground. (0V)

Psion HC RS232 / Bar Code Reader module, Technical Specification

This module combines a standard RS232 interface via a 9 way male D-type (PC-AT type) connector with a bar code interface via a 9 way male D-type click-lock connector.

Physical

Part number	1502-0044
Module:	Integrated removable module.
Operating temperature:	-20 to +60 °C
Storage temperature:	-20 to +60 °C
Weight:	62g
HC compatibility	Yes. Fits into either of the HC's expansion ports.
HC-DOS compatibility	Yes
Docking station compatibility	Yes
EMC:	FCC Class B, CE-mark and E-mark
Safety	EN60950
Weatherproofing:	No

RS232 interface

The RS232 interface is accessed by opening TTY:A (top slot) or TTY:B (bottom slot).

It provides standard RS232 level signals and is similar to the RS232 interface on an IBM AT. The difference between this socket and an IBM AT is that the RI (ringing indicator) pin (pin9) is not implemented. This pin can be optionally connected to the HC VSUP main power supply rail by fitting a jumper to the 2-pin header on the PCB (described below).

Connection

Socket:	9 way male D-type (PC-AT type)
Peripherals supported:	Will support most standard bar code wands and also scanners with wand emulation output.
Remote switch-on:	Facility for remotely switching machine on. Selectable by switch on PCB; see below.
Power supply input:	Optional VSUP connection, (7-10V DC unregulated), to power the HC; see below.
Power supply output:	Optional VSUP connection, (6-10V DC unregulated), available to power a wand or scanner; see below.

Pinout

Pin 1:	DCD input
Pin 2:	RX input.
Pin 3:	TX output.
Pin 4:	DTR output.
Pin 5:	Ground (0v)
Pin 6:	DSR input.
Pin 7:	RTS output.
Pin 8:	CTS input.
Pin 9:	Optional VSUP connection. See below.

Pin 9 VSUP connection

The jumper should be fitted to make this connection **only** if **one** of the following is required:

- The HC is to be powered externally. The supply applied to the HC must be diode isolated (to prevent power drain from the HC) and in the range **7-10V** (10V maximum).
  NOTE: powering the HC from this pin will **not** charge the HC internal battery.

- The HC is required to supply power to another device, for example certain true RS232 I/F laser scanners require power to be supplied from the RS232 connector. Note that VSUP is not software switchable and is present even when the HC is switched off, so the device would need its own ON-OFF switch to prevent the HC battery being drained when the device is not in use.
  VSUP is an unregulated supply in the range 6-10V. The device powered from pin 9 should not draw more than **250mA** from VSUP if no other expansion modules are powered up simultaneously. If another expansion module is fitted and powered up, the current drawn should not exceed **200mA**.

Warning: the jumper should not be fitted in any other circumstance, to prevent damage to the device connected or to the HC.

DSR auto-wakeup switch

If you hold the module with the component side of the PCB facing you and the D-type connectors at the top, the switch is located to the bottom left hand corner of the PCB. If this switch is ON (in the left hand position) the HC is automatically turned on when DSR is asserted by the device connected to the RS232 port.

Power consumption

The RS232 port is only powered up when the appropriate channel is open. Typically the interface draws **10mA** plus the current drawn by the device connected at the other end, this will vary depending on the device, for example connected to a PC the total current drawn will increase to about 20mA (this however will vary from one PC to another).

Bar code interface

The interface to the Bar code decoder is via RS232 serial signals. It is accessed by opening TTY:D(top slot) or TTY:E (bottom slot). The port is powered up and down by opening and closing the appropriate channel.

Decoder

Decoder IC:	Hewlett Packard HBCR-1612
Input speed:	9600 Baud via serial
Input data:	8 Data bits, 1 Stop Bit
Discrimination:	Automatic
Supported symbologies:	Code 39 (standard or extended)
	Interleaved 2 of 5
	UPC A, E0, E1 (with supplemental digits)
	EAN/JAN 8,13 (with supplemental digits)
	Codabar
	Code 128
Maximum scan speed:	30 ips (76 cm/s)
Output data:	By default when a successful bar code is read the number is transmitted to the HC in ASCII format followed by a carriage return.
Peripherals supported:	Will support most standard bar code wands and also scanners with wand emulation output.
Unsupported scanners:	'Undecoded Laser Scanner' (also known as HHLC, hand held laser compatibility).
Programming:	The HBCR-1612 is programmable via escape sequences. For more detailed programming information refer to the *I/O Devices Reference* manual.

Connection

Socket:	9 way male D-type click-lock
Power supply outputs:	VSUP connection, (6-10V DC unregulated), and 5V DC regulated, available to power a wand or scanner. See below.

Pinout

Pin 1:	DCD input
Pin 2:	Bar Data input
Pin 3:	No connect
Pin 4:	Switched VSUP (6-10V DC) output*.
Pin 5:	DSR input
Pin 6:	DTR output.
Pin 7:	Ground (0V)
Pin 8:	Ground (0V)
Pin 9:	Switched 5V regulated output*.

VSUP and 5V regulated outputs

Both these switched power supply outputs are provided to power devices plugged into the bar code port and both are enabled only when the appropriate channel is open.

VSUP is an unregulated supply in the range 6-10V.

The 5V regulated output has a tolerance of +/- 5%.

*The device powered from either supply should not draw more than **250mA** if no other expansion modules are powered up simultaneously. If another expansion module is fitted and powered up, the current drawn should not exceed **200mA**. If current is drawn from both rails then the combined current should not exceed 250mA (or 200mA if another module is present).

Power consumption

The Bar code port is only powered up when the appropriate channel is open. Typically the interface draws **10mA** idle plus any current drawn by the wand. During a scan the interface typically draws **24mA** plus the current drawn by the wand. Current drain varies greatly from one wand to another and choice of wand can have a considerable effect on battery life.

Note

HC bar code readers may be converted for use with the Psion Work*about*. See *Appendix A - Technical Specifications* in the *Workabout Programming Guide* manual.

Psion HC Modem UK module, Technical Specification

Physical

Part number:	2400-0090-01
Module:	Integrated removable module.
HC compatibility	Yes. Fits into either of the HC's expansion ports.
HC-DOS compatibility	No
Docking station compatibility	Yes
Certification:	BABT approved in UK (approval number NS/1397/3/T/605141) BS6301 (safety)
Power:	70mA maximum

Environment

Operating temperature:	0 - 50C
Operating humidity:	0 - 96% non-condensing

Communication modes

V standards:	V21, V22, V22bis, V23.
Operational modes:	V22bis 2400 bps full duplex.
	V22 1200 bps full duplex.
	V23 1200/75 bps full duplex.
	V23 75/1200 bps full duplex.
	V21 300 bps full duplex.
Data transfer rate:	Up to 2400 bps with V22bis.

Network connection

Line connection:	BT 600 series jack for 2 wire PSTN,
	3 wire bell tinkle suppression supported
Signal level:	-9dBm.
Equalisation:	Transmit - fixed compromise, receive - automatic adaptive.
Interface:	600Ω
REN:	1

Autodial/autoanswer

Dial method:	Pulse and tone dialling.
Call progress:	Internal loudspeaker with volume control, extended results codes.
Call control:	Extended Hayes AT command set.
Auto answer:	To ITU-T (CCITT) V25 recommendation, with echo suppression.
Mode selection:	Automatic configuration to V23/V22bis/V22/V21 on receive.
Call disconnection:	Loss of carrier, DTR or by command.

Data interface

DTE interface:	Psion high speed serial
	Compliant with V24/V28 TX, RX, RTS, CTS, DSR, DCD, DTR, RI
Command buffer:	40 characters
Protocol:	Async command and data mode.
DTE speed:	300, 600, 1200 and 2400bps.
Error correction:	V42 including LAPM and MNP Class 4.

Diagnostics

Test modes:	V54 digital and analogue loops.

Psion HC Vehicle Interface Box Technical Specification

This unit is designed to be mounted in a vehicle and provide the following functions:

- DC power regulation and protection.
- Wiring interfacing.
- Direct connection to the RS232 interface.

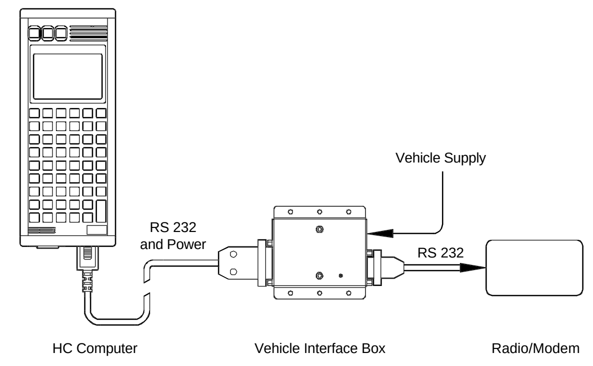
The unit and cables have E-Mark certification.

The wiring connections, (as shown in the system block diagram below), are:

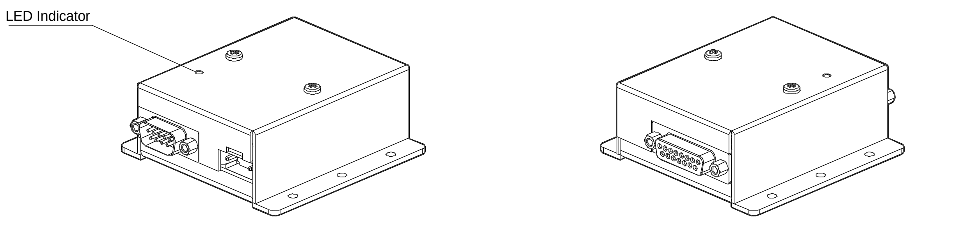
- Unregulated 10-18 volts input from the vehicle source, ('Vehicle Supply').
- RS232 serial interface to a radio or telephone modem, ('RS232').
- RS232 to/from the HC, power, trickle charge for the battery, ('RS232 and Power').

Note that the HC Vehicle Interface Box does not support the Psion Work*about* range; see *Appendix A* in the *Workabout Programming Guide* manual for a description of the Work*about* VIC (Vehicle Interface Cradle).

The HC needs to be fitted with a special LIF-PFS/TTL-RS232 expansion module.



System block diagram



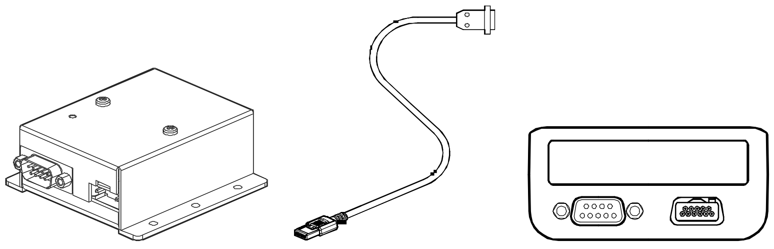
9 Way D type (Male) connector and 2 way Power input

15 Way D type (Female) connector

End views of the HC Vehicle Interface Box showing the connectors

The Vehicle Interface kit includes:

- Vehicle Interface Box, Part Number 2400-0079.
- A LIF - RS232 cable 1.5m length, terminated at one end with a LIF connector (Polarisation Type A) and a 15 way D type connector at the other.
  LIF - RS232 Cable : Part Number 2403-0011
- The appropriate HC Expansion Module, TTL/LIF-RS232. Part Number 2400-0068



Vehicle Interface Box Installation Kit

Psion HC Cradle Technical Specification

Note: this HC accessory has been superseded by the Psion HC Docking Station.

Dimensions

Size:	190mm (length) x 150mm (width) x 850mm (height)
Weight:	400g
HC compatibility	Yes. Models prior to revision 4 (serial number below 200,000).
HC-DOS compatibility	No

Interfaces

HC serial interface:	High speed - 190kBytes/sec
Expansion module slot:	Fits RS232/Parallel and Modem modules

Battery recharge

Trickle charge:	14-16 hour recharge of HC battery in situ
NiCad recharge slot:	Allows charging of stand alone spare battery

Features

Security lock:	Ensures HC held in position
Insertion/removal:	Trigger loaded spring release and hand recess
Control panel:	LEDs indicating mains power, fast charge, spare battery charge, active comms

Mounting options

Flat surface:	e.g. point-of-sale counter
Wall mounting:	e.g. industrial environments
In-vehicle:	e.g. fleet vehicles

Psion's continuing product development and improvement programs mean that specifications and features are subject to change at any time and without notice.

Psion HC Docking Station Technical Specification

Introduction

The HC Docking Station is designed to provide a multi-function mounting point for the Psion HC and the Psion HC-DOS corporate hand held computers, (referred to in this technical specification as *the computer*).

The Docking Station supersedes the HC Cradle, and has the following features:

- Battery management, including fast charge of batteries (fast model)
- Small footprint
- Reliable connection between the Psion and the docking station using the new LIF connector
- Option for in-vehicle use
- FCC, static and safety approval

Compatibility with Psion HC and RWAN machines

Compatibility with the Psion HC

To allow connection of the HC to the Docking Station using the LIF connector, the HC main circuit board was revised so that the connections for the fast serial and charging interfaces were available at the bottom expansion slot (version 4 onwards). As a result of this, Psion HC computers with pre revision 4 boards (serial numbers below 200,000) are not compatible with the Docking Station or expansion modules which have a LIF interface. The HC must also be reproed to version 1.70F or above of the EPOC operating system.

A spares kit is available consisting of an HC main board (latest revision), plus the side and bottom boards. This allows a field update so that early versions can be made compatible with the Docking Station. Contact your Psion distributor for more information.

Only HC battery packs marked "Fast Rechargeable" and with the letters "FC" (for Fast Charge) in the top right hand corner of the label are suitable for fast charging with the HC docking station.

Compatibility with RWAN/PDT220

The Docking Station does not support RWAN/PDT220 machines.

Variants

A total of four build options available for the HC:

1. HC fast charge

2. HC trickle charge,

Note: Both the above variants are also available with vehicle support circuitry on board.

This gives a total of 4 possible build variants.

Identification

PCB number and revision marked on PCB is common for all variants.

The main visual differences that distinguish an HC fast charger from a Work*about* fast charger are:

HC fast charger:	4 pin bulky power supply socket fitted
Work*about* fast charger:	2 pin 1.3mm DC jack fitted

Docking Station Unit

Main features

The Docking Station Unit has the following features:

- Fast charging of the computer internal battery pack, (fast model).
- Spare battery pack fast charging.
- Stable desktop mounting.
- Accepts some of the HC expansion modules which communicate via the Psion Fast Serial (PFS) protocol. These are accessible by the HC and HC-DOS computers. See the table *Psion HC build variant and accessories matrix* at the end of this Appendix for details.
- Data transfer from the computer, (with the appropriate expansion module and software driver).
- Simultaneous battery charging and data transfer, (if the Psion is not monitoring battery status).
- Wall mounting and bulk head fitting designed in.
- The battery compartment is factory configured to accept as standard the HC rechargeable battery pack.

Status indicators

There are several LEDs on the front of the charger unit to indicate the following:

- Communications/data transfer
- Yellow during data transfer
- Power status-On/Off
- Green when charger is connected to mains power
- Main computer battery charging status (Fast model only, see *Battery Status LED conditions*)
- Spare battery charging status (Fast model only, see *Battery Status LED conditions*)

Battery charging

The Docking Station has two charging modes:

- Normal
- Software controlled. See the *Cradle and Docking Station* chapter in the *I/O Devices Reference* manual.

If both the computer and the spare battery are fitted when the docking station is connected to mains power, charging priority will go to the spare battery. If the docking station is already connected to mains power , charging priority will go to whichever battery was plugged in first.

The computer main battery can be discharged before charging commences. This feature is controlled from the computer.

Note that the Slow Charge variant of the Docking station does not have a Battery Status LED. This is because it has only one status - charging.

Battery Status LED conditions

LED indication	Battery status
Flashing red	Preparation for fast charging (two seconds) **or** Battery condition outside specified range - trickle charging **or** For the battery pack inside the Psion: discharging under software control **or** Error
Steady red	Charging
Steady green	Charged
Flashing red/green	Waiting **or** For the battery pack inside the Psion: discharging under software control, while the spare is fast charging

Charging both battery packs

If a spare battery is inserted into the Docking Station whilst a battery pack inside the Psion is being charged, charging of the spare pack will begin after the internal battery pack has been charged.

If a Psion computer is inserted into the Docking Station whilst a spare battery pack is being charged, charging of the battery inside the Psion will begin after the spare pack has been charged.

If both the Psion and the spare battery pack are inserted into the Docking Station at the same time, (or both are in the Docking Station prior to it being connected to the mains), the packs will not be charged simultaneously. In the case of the Fast Charge variant of the Docking Station their respective LEDs will flash red for about two seconds, until the charger decides which to battery pack to charge. The LED for the one charging then comes on red, and the other one's LED starts flashing red/green as it is waiting to be charged. For both Docking station variants the spare battery pack will normally be charged first.

Battery Fast Charging conditions

The Fast Charge variant of the Docking Station can Fast Charge in the following conditions:

Within the temperature range:	5 to 45 °C
Voltage of the battery pack:	4.5 to 11.3V DC for the HC and HC-DOS

If the battery pack temperature or voltage is outside the specified range, the charger trickle charges until the condition is within the allowable range, after which it will fast charge. A new or fully discharged battery pack (that has been left on for a long time) may have a voltage below the minimum for Fast Charging.

If the battery temperature is within the allowable range and the battery status LED continues to flash red it is likely that the battery pack is faulty.

Discharging prior to charging & capacity measurement

The Psion's internal battery pack may be discharged, under software control, prior to charging. This is not possible with the spare battery pack.

It is possible to charge the spare battery pack whilst discharging the main battery pack in the Psion.

Software controlled discharging of the battery pack leaves the voltage above the allowable minimum for subsequent Fast Charging.

Charging will automatically commence after the battery is discharged.

Under software control it is also possible to measure the actual capacity or the remaining capacity of the battery pack inside the Psion computer. the discharging current for the HC is 300mA ± 5%.

Fast Charging times

A fully discharged battery pack takes approximately one hour to Fast Charge to 90-95% of its maximum capacity. If left in the Docking Station after this time it will be "topped-up" to its maximum capacity after a further two hours.

Slow Charging times

A fully discharged battery pack takes approximately 14 to 16 hours to Slow Charge to 100% of its maximum capacity.

Charging limitations

The Fast Charge and Slow Charge facilities only support the main Computer battery, not the battery of any attached peripheral. The HC Printer however, contains its own Quick Charge circuitry and may charge simultaneously under software control.

LIF Mounting Kit

The LIF mounting kit allows a LIF connector on the end of a cable to be fitted to a holster.

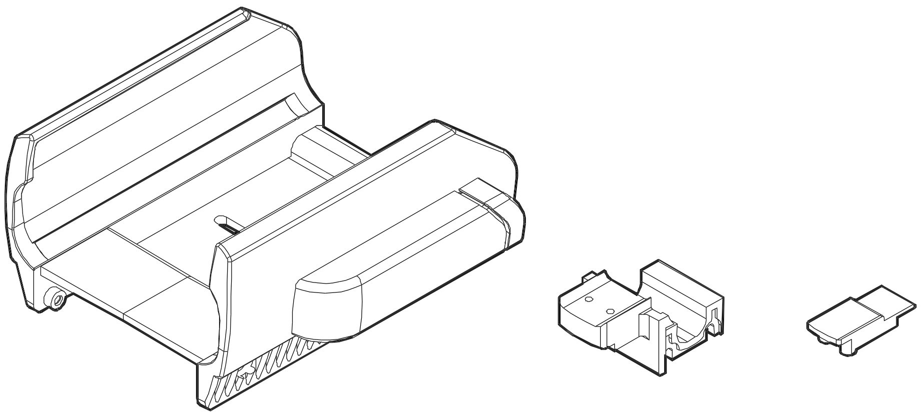
The holster itself is a plastic moulding into which the computer can be inserted. This incorporates a positive latching mechanism which holds the computer securely in place. The holster does not include any electronics.

The Clip cover and the 2 short screws that are fitted as standard to the LIF connector will need to be replaced with the blank cover and the 2 long screws supplied with the kit.

HC/HC-DOS Holster with Socket Housing

The kit for the HC Computer consists of:-

- HC holster
- LIF connector rear housing
- LIF connector blank front cover
- 2 screws **-** type K2.2 x 12 mm CSK (not shown)

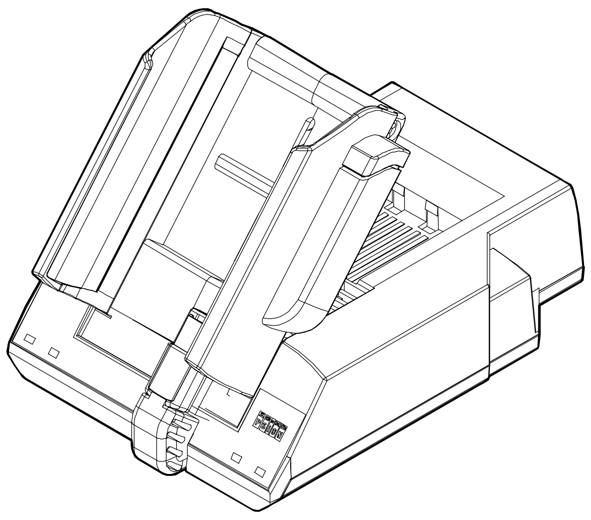


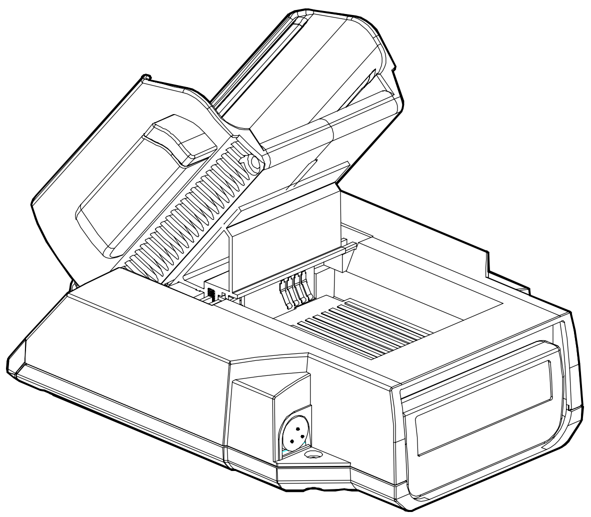
HC Docking Station

This is a Battery Charger with serial data communication capabilities supplied with a factory fitted HC holster, also known as an HC Docking Station. It comes in two variants, Fast Charge and Slow Charge. The cable from the hardware board to the LIF connector is protected by an over-moulded rubber grommet. The HC Docking Station is also compatible with the HC-DOS computer.

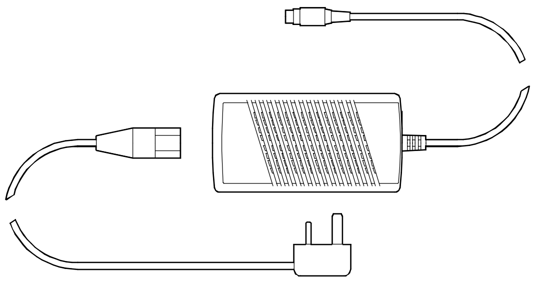
A 12v 2 amp unregulated power supply is available separately.

HC Docking Station: Part Numbers 1503-0017-01 (Fast Charge)
1503-0018-01 (Slow Charge)





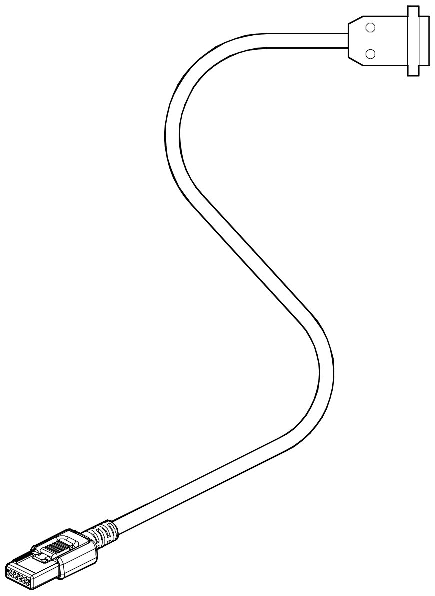
12V 2 amp unregulated Power Supply



Note: Euro part number 2300-0212-01, US part number 2300-0213-01; a universal switch mode adaptor is also available, part number 2402-0003-01 (contact your Psion distributor for details).

Psion LIF - RS232 Cable Technical Specification

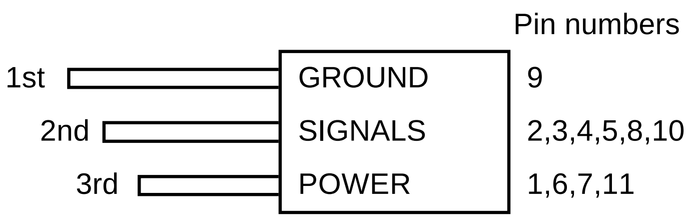
A cable 1.5 m length terminated at one end with a LIF connector (Polarisation Type A) and a 15 way D type (Male) plug at the other. LIF - RS232 Cable .



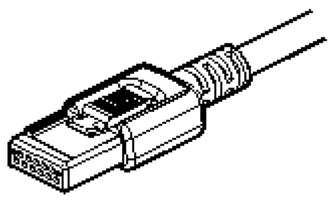
Psion LIF Connector Technical Specification

The Low Insertion Force (LIF) connector has been designed for connecting the computer to the Docking Station, as well as to other Psion accessories.

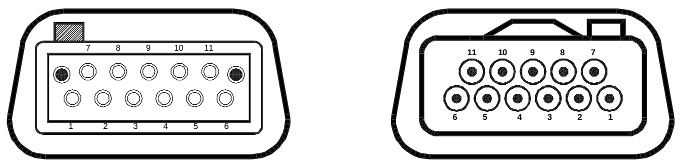
The LIF connector cover is moulded with a polarising pin in one of two positions.



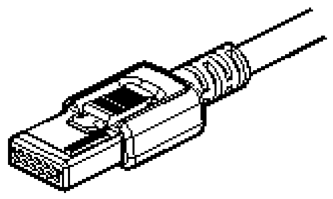
The step arrangement of the LIF Connector pins



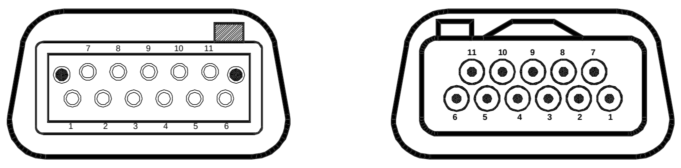
Polarising Pin A



Cable mounted LIF (Female plug)   Computer mounted LIF (Male socket)



Polarising Pin B



Cable mounted LIF (Female plug)   Computer mounted LIF (Male socket)

The Type A and Type B polarisation of the LIF Connector

Pin Definition for LIF - PFS Connector

LIF Connector Polarisation Type B

Pin No	Pin Name	Wire Gauge	Colour	Contact	Direction (Docking Station's perspective)	Standard Function	Docking Station usage
1	LCA	7/0.1	Brown	Third	Input	Local[1] Computer Active. High when the computer is on. (The Work*about* can source 100mA from this pin and the HC/HC-DOS 5mA to power remote[2] circuitry)	Used as an enable for the Docking Station resident expansion module 5V supply.
2	EXON	7/0.1	Blue	Second	Output	EXternal switch ON, active high (+5V). Asserted by a remote[2] device to switch on the computer.	May be asserted by a Docking Station resident expansion module.
3	INT	7/0.1	Orange	Second	Output	INTerrupt to computer, active high (+5V).	May be asserted by a Docking Station resident expansion module.
4	THM	7/0.1	Yellow	Second	Input	Battery thermistor terminal. Allows remote[2] sensing of the battery temperature.	Standard function
5	DLA	7/0.1	Green	Second	Output	Disconnect Local[3] ASIC, active high (+5V). (does not apply to Work*about*). When this signal is asserted the serial channel is disconnected from the local[3] ASIC4/5 in the HC resident expansion module (if present) and instead connected to a remote ASIC4/5 (if present).	Asserted by the Docking Station ASIC, connects the Docking Station resident expansion module to the serial channel.
6	BAT	28 SWG	Red	Third	Output	+ve battery terminal (1 amp)	Standard function
7	Vin	28 SWG	Black	Third	Output	Power supply to computer (+10V)	Standard function
8	SCLK	7/0.1	Grey	Second	Input	Serial channel CLocK.	Standard function
9	GND	28 SWG	White	First	-	Power, signal ground and -ve battery terminal (1 amp)	Standard function
10	SDATA	7/0.1	Violet	Second	Bi-directional	Serial channel DATA.	Standard function
11	STATUS	7/0.1	Pink	Third	Output	STATUS. Connected to a pull-up resistor to allow connection to an open-collector/drain driver. Normal usage is: low indicates the presence of a remote[2] device.	Driven low by an open collector driver when LCA is high and the Docking Station is powered-up to allow the computer to sense whether or not the Docking Station is connected.

Pin Definition for LIF - RS232 Connector

LIF Connector Polarisation Type A

Pin No	Pin Name	Wire Gauge	Colour	Contact	Direction (Computer's perspective)	Function
1	DCD	7/0.1	Brown	Third	Input	RS232 signal
2	RX	7/0.1	Blue	Second	Input	RS232 signal
3	TX	7/0.1	Orange	Second	Output	RS232 signal
4	THERM	7/0.1	Yellow	Second	-	Battery thermistor terminal
5	DTR	7/0.1	Green	Second	Output	RS232 signal
6	VBAT	28 SWG	Red	Third	-	+ve battery terminal
7	VIN	28 SWG	Black	Third	Input	Power supply to computer
8	DSR	7/0.1	Grey	Second	Input	RS232 signal
9	GND	28 SWG	White	First	-	Power, signal ground and -ve battery terminal
10	RTS	7/0.1	Violet	Second	Output	RS232 signal
11	CTS	7/0.1	Pink	Third	Input	RS232 signal

Definitions

Computer	HC, HC-DOS or Work*about*
Docking Station resident expansion module	Expansion module fitted to the Docking Station, may or may not be present.
HC resident expansion module	Expansion module fitted to the HC which contains the Docking Station interface and possibly another peripheral.
HC peripheral	A peripheral located in the HC resident expansion module which is connected to the same serial channel as the Docking Station.
Docking Station ASIC	An ASIC5 located on the main Docking Station PCB which remains connected to the serial channel irrespective of the state of DLA.

Notes

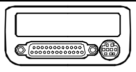
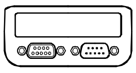
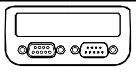
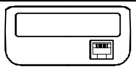
1. The term "local computer" implies the computer local to the LIF connector, i.e. the HC, HC-DOS or Work*about*, as opposed to a "remote" computer which might be connected via a Docking Station resident expansion module, for example.
2. The term "remote" implies something on the other side of the LIF connector to the computer.
3. The term "local" implies something on the computer side of the LIF connector including devices on an HC resident expansion module.

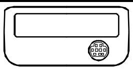
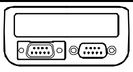
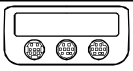
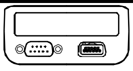
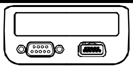
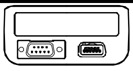
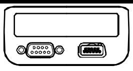
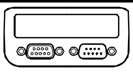
Psion HC build variant and accessories matrix

KEY : ● Compatible X Not compatible / not available

Build variants		**HC 100**	**HC 110**	**HC 120**	**HCR 400/800 900**	**HC Docking Station**	**Work*about* Docking Station**
Screen	**With EL backlighting**	●	●	●	●		
	Without EL backlighting	●	●	●	X		
Use	**Industrial**	X	●	●	●		
	Non-industrial	●	●	●	X		
Keypad	53 Key A/N UK 2401-0026	●	●	●	X		
	A/N European 2401-0051	●	●	●	●		
	A/N Scandinavian 2401-0050	●	●	●	X		
	Numeric only UK 2401-0046	●	●	●	X		
	53 Key A/N USA 2400-0026	X	X	X	●		

HC Expansion modules

	**RS232 / Parallel (printer) version 1** 1502-0001, 25 way D type (F) + 9 way Mini DIN FCC Class B, CE-mark, E-mark, EN60950	●	●	●	●	**not CE**	●
	**RS232 / Parallel (printer) version 2** 1502-0052, 15 way High Density + 9 way D type (M) FCC Class B, CE-mark, E-mark, EN60950	●	●	●	●	●	●
	**RS232 / TTL-RS232** 1502-0039 (IP64), 1502-0040 (NON IP64) 9 way D type (F) + 9 way D type (M) FCC Class B, CE-mark, E-mark, EN60950	●	●	●	●	●	●
	**UK Modem (ASIC 8) 1502-0010** RJ 11 connector BABT Approved in UK, BS6301 (Safety)	●	●	●	●	●	●

Build variants		**HC 100**	**HC 110**	**HC 120**	**HCR 400/800 900**	**HC Docking Station**	**Work*about* Docking Station**
	**Barcode only** HP Wand HBCS-A207 + Plug + EXMOD 1502-0020 Wand Welch Allen + Plug + EXMOD 1502-0021 FCC Class A / VDE Class B	●	●	●	●	X	X
	**RS232 / Barcode 1502-0044** 9 way D type Quick Loc(F) + 9 way D type (M) FCC Class B, CE-mark, E-mark, EN60950	●	●	●	●	●	●
	**MCR / Scanner / RS232 1502-0003** MiniDIN connectors: Scanner NipDenso + Plug (1502-0022) Scanner DigVision + Plug (1502-0023) Magnetic Card Reader + Plug (1502-0024) FCC Class B / VDE Class B	●	●	●	●	●	●
	**LIF-PFS / RS232 (available on request)** 9 way D type (M) + 9 way LIF- PFS (M) FCC Class B, CE-mark, E-mark, EN60950	●	●	●	X	X	X
	**LIF-PFS / TTL-RS232 (due 1995)** 9 way D type (F) + 9 way LIF- PFS (M) FCC Class B, CE-mark, E-mark, EN60950	●	●	●	X	X	X
	**LIF-PFS / Barcode 1502-0043** 9 way D type Quick Loc(F) + 9 way LIF- PFS (M) FCC Class B, CE-mark, E-mark, EN60950	●	●	●	●	X	X
	**TTL-RS232 / LIF-RS232 (Vehicle)** 9 way D type (F) + 9 way LIF- RS232 (M)	●	●	●	●	X	X
	**16550 RS232 / TTL-RS232 (1502-0045)** 9 way D type (F) + 9 way D type (M) FCC Class A	●	●	●	●	X	X
	Printer (high resolution) (1502-0037)	●	●	●	X	●	●
	Laser scanner (1503-0012)	●	●	●	X		

Fast	Docking Station (Fast Charger with Holster)	●	●	●	X
Charger	Fast Charger without Holster (not yet available)	●	●	●	X
Trickle	Docking Station (Trickle Charger with Holster)	●	●	●	X

Build variants		HC 100	HC 110	HC 120	HCR 400/800 900	HC Docking Station	Work*about* Docking Station
Charger	Trickle Charger without Holster (not yet available)	●	●	●	X		

Additional accessories	Nicad battery pack 600 mA (1503-0005)	●	●	●	X
	15 way high density to 25 way Centronics convertor cable (2403-0026)	●	●	●	●

APPENDIX B

SAFETY AND EMISSIONS APPROVALS

Safety and emissions technical terms explained

CE	From 1 January 1996 all electrical and electronic equipment, that fall within the scope of 89/336/EEC (‘The EMC Directive’), sold in the EU must have a CE Mark.
EN60950	The *E*uropean *N*orm (i.e. a specification recognised throughout the EU) for Safety of Information Technology Equipment.
EN55022	The *E*uropean *N*orm for Emissions from Information Technology Equipment. It is known as a 'Product Specific Standard'.
FCC	Stands for *Federal Communications Commission* which is the body in the USA for providing equipment authorisation. Class B are the emission limits the FCC have set for residential equipment. Class A are the emission limits for commercial equipment. Psion equipment for sale in the USA needs to meet the appropriate requirement.
IEC	Stands for the *International Electrotechnical Commission* which is a standards body recognised by most western countries. IEC801 is known as a 'basic standard' and is divided into various parts, one of which covers static. The 801 series cover susceptibility, or immunity.
IP	Stands for *International Protection*. It gives a measure of how weatherproof a product is.
GS	Stands for *Geprüfte Sicherheit* ("Proof of safety"), which is used in Germany to indicate safety.

INDEX