SERIES 3/3A PROGRAMMING GUIDE

Version 2.30

March 1, 1999

Contents

CHAPTER 1

SERIES 3 PROGRAMMING OVERVIEW

Throughout the *Series 3 Programming Guide*, reference to the Series 3 machine should be taken to include the Series 3a and the Work*about* unless stated otherwise. See also the *Workabout Programming Guide* for further information specific to that machine.

Programming possibilities

In general terms, there are six different levels at which programs can be written for the Series 3:

- using OPL, creating *.opo* or *.opa* program files
- using CLIB, with user interface restricted to console i/o of the `getchar` and `puts` variety
- using PLIB, again with user interface restricted to console i/o
- using WLIB, accessing the graphics and windowing capabilities of the Window Server
- using Hwif, with its support for menus and dialogs similar to those available in OPL/w
- using HWIM, FORM, OLIB and (for the Series 3a) XADD, the object-oriented DYLs built into the ROM.

All but the first of these uses the C programming language.

For programming in OPL see the Series 3, Series 3a and Work*about* Programming manuals, or the *SIBO OPL Software Development Kit*.

For more detailed information about PLIB library routines and the window server see the *PLIB Reference* and *Window Server Reference* manuals respectively.

Programming using Hwif is also covered in its own separate manual, *Programming in Hwif*. This manual is particularly recommended on account of its relevance to the Series 3.

For details of object-oriented programming see the object oriented sections of the C SDK, starting with the *Object Oriented Programming Guide*.

The present manual focuses on features of the Series 3 and Series 3a that are by-and-large unique to these particular computers (as opposed to other computers in the SIBO range, that is, the HC and MC computers). These features include:

- the difference between *.app* and *.img* forms of program file
- communications between applications and the System Screen
- the specialised form of the command line for applications
- some device drivers unique to the Series 3 and/or Series 3a
- development tools particularly suited to the Series 3 and Series 3a computers.
- writing applications that run on all machines.

Other parts of the SDK that may be found of particular relevance to programming for the Series 3 include:

- the descriptions of file formats used by various applications, in the Additional System Information manual
- a description of the special form of printer driver files used on the Series 3, also in the Additional System Information manual.

Before proceeding further with this chapter, the reader is recommended to be familiar with the basic contents of the *General Programming* manual.

Differences between .app files and .img files

Strictly speaking, there is no real difference between files with extension *.img* and those with extension *.app*. For example, although the System Screen usually expects to install *.app* files, it will also, if requested, install suitable *.img* files.

However, by convention a *.app* file contains one or more extra so-called *add-files* embedded within it, in addition to the core *.img* file itself. These files may include:

- a *.pic* file providing the *icon* for the application
- a *.rsc* or *.rzc* file providing the *resource file* for the application
- a *.shd* file providing the *shell data* for the application.

Any such add-files can be added into the *.img* file automatically, via the operation of *emake.exe*, at the time the *.img* file is itself created. What controls the set of add-files used (if any) is the presence or absence of a suitably named *add-file list* (*.afl*) file.

Add-file lists

An add-file list (*.afl*) file is a text file containing from one to four filenames. For example, the contents of a file *tele.afl* could be:

```
tele.pic
tele.rsc
tele.shd
```

When any *.pr* project file is invoked that leads to the building of *tele.img*, the existence of a file *tele.afl* is checked for. If such a file is found, the files listed therein are combined with the core *.img* file to form a larger *.img* file as output. By convention, *.img* files that contain embedded add-files are renamed to *.app* files.

See the chapter *Building an Application* in the *General Programming* manual, for more details.

Pre-defined add-file slots

The `RSCFILE` class in the OLIB library can find resource files embedded in a *.app* file, but only if they are placed in the *second* of the four slots.

The System Screen expects to find a *.shd* file in the *third* add-file slot of a program being installed, and indeed will refuse to install it if no such data is found there.

The Window Server, however, will find a *.pic* file (or a *.fon* file) at any of the four add-file positions within a *.app*. In view of the requirements for resource files to occupy the second slot, and for *.shd* files to occupy the third slot, *.pic* icon files are usually placed in the first add-file slot - as in the above example.

At the time of writing, there is no mechanism to specify a file for (say) the third add-file slot, while omitting the second add-file altogether. For this reason, it may occasionally be necessary to create and maintain a zero-length file for the second slot.

Running programs via RunImg

An alternative to installing a program in the System Screen with its own file list is to run it from the RunImg icon:

- the user presses TAB while the highlight is within the RunImg file list
- the user navigates the file selector until it selects the program file
- the user presses ENTER to start this program.

This procedure can be simplified if the program is placed in a top-level *\img* directory, and given the extension *.img*. In this case, the program will automatically appear as an entry in the RunImg file list.

However, a program run in this way will receive no special command line from the System Screen. Nor will it receive any *Shutdown* or *Switchfile* messages from the System Screen. (These messages are discussed in the following chapter).

Program files with and without icon files

An application can simply not have an icon, although in this case the system screen will simply refuse to install it. The application can, however, still be run from RunImg, in which case, an empty icon boundary will be displayed in any status window used.

Icon files, such as *tele.pic*, may be produced by a variety of means:

- using the *Iconed* demo application that can be built using the Hwif part of the SDK or, for the Series 3a and Work*about*, the *Iconeda* application that is installed in a *\sibosdk\s3atool* directory
- using the Window Server tool *wspcx.exe* on the *.pcx* output of a PC program such as *Windows PaintBrush*.

The *.pic* file can contain:

- a 24x24 bitmap for a Series 3 icon.
- two 48x48 bitmaps for a Series 3a icon - the first and second bitmaps specify the black and grey planes respectively.
- a 24x24 bitmap for a Series 3 icon followed by two 48x48 bitmaps for a Series 3a icon.

The format of *.pic* files is given in the *Bitmaps* section in the *Window Server Reference* manual.

Resource files and shell data files

See the *Resource Files* chapter of the *Additional System Information* manual for a description of the format and uses of resource files.

Shell data files are discussed in the following chapter.

Customised add-files

In most cases, at least one add-file slot will be free for use by the application itself.

One important reason to build an extra file into the *.app* file, rather than leaving such a file separate, is that it diminishes the chance of a user copying the *.app* file from one SSD to another but neglecting to copy a vital associated file at the same time.

Finding add-files within a .app file

Ordinarily, applications have no need to find add-files within themselves, since system code takes care of this on their behalf.

One possible exception is when an application needs to access a *customised* add-file. In this case, knowledge of the format of the header of a *.img* (or a *.app*) file is required. In fact, this header just contains the same information as is returned by invoking *edump.exe*.

The header of a *.img* file is described by the `ImgHeader` struct, defined in *epoc.h*. See the chapter *Processes and Inter-process Messaging* in the *Plib Reference* manual for a discussion on this struct.

An example of code that attempts to locate a resource file in the second add-file slot of a *.app* file is as follows:

```
GLDEF INT rscfile_rs_init(PR_RSCFILE *self,TEXT *name)
    {
    ImgHeader head;

    /* self->rscfile.offset is zero by default */
    f_open(&self->rscfile.pcb,name,P_FRANDOM|P_FSTREAM|P_FSHARE);
    if (p_read(self->rscfile.pcb,&head,sizeof(head))==sizeof(head))
        {
        if (!p_scmp(&head.Signature[0],"ImageFileType**"))
            { /* we have a .img file */
            if (!(self->rscfile.offset=head.Add[1].offset))
                p_leave(E_FILE_INVALID);
            }
        ...
        }
    ....
    }
```

Multi-lingual applications

The topic of multi-lingual applications is discussed in general terms in the course of the *Resource Files* chapter of the *Additional System Information* manual.

There are some issues about the set of possible command hot-keys ("menu accelerators"), however, that are particular to the Series 3.

The set of possible accelerators varies from language to language on account of the keyboard changing.

All languages must, however, support the 26 accelerators 'a' through 'z', together with four more.

These additional accelerators are '+', '-', '*', and '/' in most languages. The only exceptions so far are French and Spanish (and Belgian, which uses the French keyboard):

- French replaces '/' with '?'
- Spanish replaces '*' with '>' and '/' with 'ñ'.

Applications which fail to take account of these changes when they are translated into another language will find they end up carrying a "lame" accelerator: the accelerator is displayed on the menu, but there is no way for the user to press the required key combination.

Environment variables on the Series 3

Environment variables can be a powerful programming resource whilst being, at the same time, potentially *anti-social*.

There are two aspects to this:

- environment variables consume space in a special RAM segment devoted to them - the more environment variables are created (and the larger these are), the greater the chance becomes of other applications failing to work properly - on account of not being able to create *their* environment variables.
- *name clashes* are possible - data stored in an environment variable by one application may get obliterated by another application storing different data to an identically named variable.

With regard to the first problem, all that can be said is that due caution should be observed. Otherwise, your application may earn itself a bad name.

With regard to the second problem, what is evidently required is some kind of *naming convention*.

For a full discussion of environment variables see the paragraphs preceding `p_getenv` in the *Plib Reference* manual.

Avoid $ signs

The '$' sign is used in names of environment variables created and manipulated by Psion system software.

All external applications should completely avoid using environment variables with '$' signs in them - unless they first secure the agreement of Psion.

The plan is to extend the use of '$' to mean, not just "used by Psion", but rather "licensed by Psion". Interested software developers who contact Psion will be given a short identifier - for example, "17". A company which receives this identifier could then create environment variables with names such as "17table" or "17md", secure in the knowledge that no other responsible developer will also use these names.

In conclusion, '$' signs should be avoided in all cases; even where approved by Psion, environment variable names should include '$' signs *only in their identifier region*. Thus a name of "A17$" would not be allowed. Environment variables can of course continue to have "simple" names, such as "table" and "md", but in this case, the chance of a name clash remains.

Series 3 family compatibility

Series 3/Series 3a/Work*about* compatibility

All Series 3 applications are fully compatible with the Series 3a and Work*about*. These two machines automatically recognise such applications and run them in compatibility mode - both the icon, as displayed on the system screen, and the display are expanded linearly by a factor of two in each dimension.

In fact an application that wishes to use the full screen capabilities of the Series 3a or Work*about* must explicitly turn off the compatibility mode by calling the wCompatibilityMode function - see the *Window Server Reference* manual for further details.

An application can identify which machine it is running on - using a call to p_getlcd - see the *Plib Reference* manual for details, and *Compatibility* in the *General Programming Manual* for an example.

Thus it is quite feasible to write an application that runs on the Series 3, Series 3a and the Work*about*, using the screen of each machine to the full. However, care should be taken to ensure that the application does not use any Series 3a or Work*about* specific features when running on the Series 3 - the grey scale or, on the Series 3a, the improved sound facilities for example. The Work*about* also has different keyboard scan codes from the other members of the Series 3 family, (see *Hardware Management*, in the *EPOC O/S System Services* manual).

Compatibility with Series 3c and Siena

Programs written for the Series 3

All Series 3 programs can be expected to run without modification on both the Siena and the Series 3c. The programs will run in compatibility mode, as they do on the Series 3a.

Programs written for the Series 3a

Most types of program will run without modification on the Series 3c.

It is likely that most Series 3a programs will need some modification, to take account of the smaller screen size, before they will run on the Siena. Menus and dialogs in Series 3a programs will, in general, be too wide and will need to be reorganised and/or reworded. Applications that use a sophisticated layout in their display are likely to need extensive modification before they can be used on the Siena.

Series 3a applications written using the Hwif library, and which use grey lines in their menus, will need to be relinked with a suitably modified Hwif library before they will run on either the Siena or the Series 3c. This incompatibility is associated with the introduction of small fonts for dialogs on the Work*about* and Siena.

Hwif programs that are compiled and linked with a modified Hwif library (see the *Programming in Hwif* chapter of this upgrade document) that is supplied with the upgrade software can run on the Series 3, Series 3a, Series 3c, Work*about* and Siena - provided, of course, that their displays are tailored to the various screen sizes and graphics capabilities of these machines.

CHAPTER 2

COMMUNICATING WITH THE SYSTEM SCREEN

Introduction

An important aspect of the Series 3 is the way all the built-in applications communicate with the System Screen application (also known as the *Shell* application):

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

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

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

Creating .shd files

The format of .ms files

The shell data of an application (see above) is expressed in source form in a file that typically has extension *.ms*. For example, the contents of a file *tele.ms* could be:

```
Tele.TEL
\TEL\
3
```

Running the tool *makeshd* as follows

```
makeshd tele
```

would produce the file *tele.shd* from *tele.ms* (barring syntax errors in the *.ms* file).

The *.shd* file can then be combined into the final *.app* form of an application as discussed in the previous chapter.

The first line in a *.ms* file has general form

```
<Name>[<.EXT>]
```

with the extension `.EXT` only being present for file-based applications. In that case, `.EXT` defines the *default extension* for the application. In all cases, `Name` is the so-called *public name* of the application. This must be a valid file name, that is, it must start with a letter and not exceed eight characters.

The second line in a *.ms* file gives the *default directory* for an application. This can be left blank for non file-based applications. For example, the *.ms* file for the built-in *Time* application could be

```
Time

8000
```

with the second line left blank.

The third line in a *.ms* file gives the *type* of the application (sometimes called the *type number* of the application).

A *.ms* file can have fourth, fifth, sixth ... lines, but only if `2000` has been added to the application type, so that the application has *multi-lingual* shell data (see below).

At present, there is no scope for the inclusion of comments in a *.ms* file.

Note that the entire contents of a *.ms* file is case sensitive. Eg an application with first line

```
RunGame
```

in its *.ms* file will have public name *RunGame*, whereas an application with first line

```
Rungame
```

in its *.ms* file will have the distinct public name *Rungame*. Further, the default extension and default directory should always be given in upper case (otherwise the file list may fail to display *any* files).

Default extension

The significance of the default extension for a file-based application is as follows:

- Files will be shown in the file list for the application in the System Screen only if their extension matches the default (with the exception that files are *always* shown - in bold - if they are currently the open file of an application)
- The System Screen will pass the specified default extension to the application as part of its command line, when it is started
- Typically, applications will use filename selectors in dialogs which hide extensions matching the default (eg showing *Tele* rather than *Tele.tel*), and may even omit other files from the initial list presented.

Public name

The significance of the public name of an application is as follows:

- This is the name by which the System Screen refers to the application, eg when confirming that the application be "Removed", or when allowing an application button to be assigned to the application
- This name will be displayed in the file list for the application, in the System Screen, in any case when the list would otherwise be empty.

The contents of `Name[.EXT]` must in all cases be a valid filename, ie `Name` cannot exceed eight characters in length, `EXT` cannot exceed three characters in length, and there must be no embedded spaces (etc).

Default directory

Files will be shown in the file list for the application in the System Screen only if they are located in a directory whose name exactly matches the default (with the exception that files are *always* shown - in bold - if they are currently the open file of an application).

Default directories for applications are usually top level, as in *\TEL*. However, they can equally well be subdirectories, as in *\TEL\PRIVATE*, with the limitation that the total length cannot exceed 20 characters (this count including a terminating zero at the end of the directory path name).

Application type numbers

The significance of an application's type number is as follows:

- A type of `0` means that the application a) will have no file list when installed in the system screen - instead there will be only one entry, giving the public name of the application b) will receive no *Switchfile* commands from the System Screen and c) no filename will be specified in the command line. Type `0` applications can have only one copy running at any one time.

- A type of `2` is used to restrict a file based application such that only one copy can run at any one time - this is the case for the built-in *World* application. In contrast see types `0` and `3`.
- A type of `3` means that the application a) will have a file list when installed in the System Screen, and b) can receive *Switchfile* instructions. More than one copy of the application can be run at any one time.
- A type of `4` means that the application will have a file list in the System Screen, but will never be sent *Switchfile* instructions: however, it should read the command line on its start up (an example is the built-in *RunOpl* application)
- Finally, a type of `5` means that the application is a *pure file list application*, ie not a real application at all, but just an icon to group together various applications or utility programs (like the built-in *RunImg* icon) - see below for more details.

The application's behaviour may be further modified by adding one or more of the following values to the type number:

`8000`	prevents *Switchfile* messages of the Create sort being sent to the application (this makes sense for an application such as a file dumper, which can dump the contents of existing files, but cannot meaningfully create the file it is going to dump)
`4000`	prevents the application being sent *Shutdown* messages - this will also prevent the application being Killed from the System Screen (but not, for example, from the *Spy* application released as part of the SDK)
`2000`	indicates that the application's *.ms* file contains public names *for more than one different language version* (see below)
`1000`	indicates that the *.pic* file contains a 48 by 48 Series 3a icon (preceded by a 24 by 24 icon if the application is to run on the Series 3 as well as the Series 3a). The Series 3 does not recognise this flag and will simply read the 24 by 24 icon if present.
`100`	the application should not be sent an exit message so that, for example, pressing DELETE acts as Kill application. The Series 3 does not recognise this flag.
`80`	on selecting "Create new list" from the System Screen, the resulting dialog box will contain an extra line, for specifying Text editor or Word processor type. The Series 3 does not recognise this flag.

Multi-lingual forms of .ms files

Suppose the *Data* application were to be translated into French, German, and Italian, and that its public names in these languages were to be *Fiche*, *Daten*, and *Archivi*, respectively. In that case, an appropriate *.ms* file would be as follows:

```
Data.DBF
\DAT\
2003
01Data
02Fiche
03Daten
05Archivi
```

Here, the `2000` in the application type means that the remaining lines in the *.ms* file are each made up as follows:

```
<language number><public name in that language>
```

where language numbers are as documented in the `p_getlanguage` part of the *Plib Reference* manual.

The same rules apply to the public names for other languages as to the default public name defined in the first line of the *.ms* file.

It is not possible to change the default extension or the default directory from one language to another.

Applications which are intended to be capable of being run on either mono- or multi-lingual Series 3s should adopt the multi-lingual form of *.ms* file, and simply accept that the public name will be incorrect on mono-lingual machines.

For multi-lingual machines, the System Screen uses the following rules to decide what the public name of the application should be:

- the current language number is obtained by a call to `p_getlanguage`
- if this matches any language for which a public name is explicitly defined in the shell data, that public name is used
- otherwise, the default public name is used (as given on the first line of the *.ms* file).

For the sake of minimising the size of a multi-lingual *.shd* file as much as possible, it is evidently possible to omit any lines such as

```
01Data
```

which merely define the public name for some language to be what it would have been in any case, were this line omitted (in view of the contents of the *first* line of the *.ms* file).

Pure file list applications

If an application (*Utils*, say) has type `5`, and the user presses ENTER when the highlight is over some file *Xyz.abc* in the file list for that application, the System Screen makes no attempt to run the application *Utils*. Rather, it assumes that *Xyz.abc* is itself a program, and attempts to run that. In other words, instead of executing *Utils* and passing the filename *Xyz.abc* as part of the command line, it executes *Xyz.abc* (without any command line being passed).

For example, the *.ms* file for the built-in "application" *RunImg* is as follows:

```
RunImg.IMG
\IMG\
8005
```

This means that the file list for *RunImg* lists *.img* files from *\IMG* top-level directories - each of which are program files.

Note that unless the utility programs cooperate in some limited way, when run they will be listed (in bold) under the *RunImg* icon, rather than under any other pure file list icon. This is explained in the section below on the Epoc reserved static `DatProcessNamePtr`.

In practice, the simplest way to create another pure file list application is probably to use the technique of aliasing, as discussed immediately below, to alias *RunImg*.

Aliasing applications

Some file-based applications may end up with large file lists. It may be desirable to separate a file list into two or more separate lists, for example (for the *Word* application) all correspondence going in one file list, all poetry in another, and so on. These file lists could be distinguished, on the System Screen, by having distinct icons, and different application buttons could be used to cycle round running instances of these tasks.

Going further, it may be desirable for the *behaviour* of the application to alter, depending on which type of file is open. For example, the behaviour of the built-in text editor is different for *.wrd* files (when the application is seen as *Word*) from *.opl* files (when the application is seen as *Prog*).

The concept of *aliasing* an application is designed to meet these requirements. For each required new file list, an *alias* file (*.als* file) should be installed in the System Screen. In practice the user can do this on the Series 3a using the "Create new list" item from the "Special" menu (try it out ...).

Broadly speaking, the contents of a *.als* file match those of a *.shd* file: the public name, default extension, and default directory are all defined, as well as the application type number. However, the *.als* file goes beyond the *.shd* file in that it also specifies:

- the name of the application that is being aliased
- (optionally) some *alias info* that the System Screen should pass to the application when it is run, via the command line, to configure its behaviour in some special way.

Creating .als files

An *.als* file is produced from a *.ma* file and a *.pic* file by running the tool *makeals*. The three files all have the same root name (ie disregarding the extensions). For example, the command

```
makeals letter
```

produces the file *letter.als* from *letter.ma* and *letter.pic*.

The *.pic* file is the icon to use. The process of creating *.pic* files is discussed in the previous chapter.

The *.ma* file is a source file similar in format to a *.ms* file. For example, the contents of a file *letter.ma* could be:

```
Letter.let
\WRD\LET\
3
Word
```

in which there is a fifth line which is blank (*makeals* will give an error if the fifth line is omitted altogether).

Just as there are multi-lingual forms of *.ms* files, there are also multi-lingual forms of *.ma* files. However, in practice these are of limited use, for technical reasons. This is discussed in its own section (which the majority of readers can skip) at the end of this chapter.

The first three lines of a *.ma* file correspond *exactly* to those of a *.ms* file. The fourth file is the *public name of the application to alias*. The fifth line gives the alias info, which is a zero-terminated string of up to eight characters.

In most cases, the application type will be the same for the alias file as for the application being aliased. However, the public name, the default extension, and the default directory are all commonly varied.

Note that the public name of an alias must differ from that of the application it is aliasing. Otherwise, seeking to install the alias in the System Screen will have no effect (it is not possible to have two different file lists, each with the same public name).

Incidentally, no check is made, at the time of installing an alias file, that the application it aliases is itself currently installed. This check is only made when an instance of the alias is to be started.

Active aliasing and passive aliasing

In theory, all applications are capable of being aliased, without them needing to make any conscious provision for this possibility. This is known as *passive* aliasing.

Other applications pay explicit attention to any alias info that may be passed to them on their command lines, and adjust their behaviour according to the contents of this info. This is known as *active* aliasing. An example of active aliasing is that of the built-in text editor, as described in the following section. This is the only one of the applications built into the Series 3 and Series 3a that supports active aliasing.

Any other program that supports active aliasing is free to interpret alias info passed to it in any way that it wishes. There is no obligation to mimic the detailed rules obeyed by the text editor.

Active aliasing in the built-in text editor

If the alias info is a null string, the text editor enters *Word* mode, with multi-level outline facilities, styles and emphases, and so on.

Note that it is not unreasonable for an alias to define null alias info. This allows the creation of aliases of the text editor that behave in exactly the same way as the built-in Word application, but differ from each other in terms of their default extensions, default directories, and/or public names.

If there is any non-null alias info, the text editor enters one of a number of other modes, with the mode depending on the first character of the alias info. Some of these modes are not available on Series 3 machines. At the time of writing, the allowed first characters and the corresponding modes are:

`O`	OPL program editor
`S`	Comms script program editor
`$`	Plain text editor (not Series 3)
`/`	Word processor with custom template (not Series 3)

The program editor mode is available on all machines. In this mode there is no access to the style and emphasis subsystems, the corresponding menu commands being replaced by options to "translate", "run", "show error" and set "indentation".

In this mode, the first letter of the alias info denotes the nature of the program that is being edited. It actually identifies the program to invoke to effect any "translate" and (possibly) "run" commands from the user. The generic name of this program is *sys$prg?.img*, with the question mark being filled in from the first letter of the alias info. Thus the *Prog* alias has `'O'` for the first letter of its alias info, and so the OPL translate/run program *sys$prgo.img* is used. In contrast, the *Script* editor from the communications ROM has `'S'` for the first letter of the alias info, so that the program *sys$prgs.img* is used.

In program editor mode the second letter of the alias info should be `'R'` if the program is of a type that understands "run" instructions in addition to "translate" ones. Any other second character disables the "run" command option. The following three letters (e.g. `'OPO'` or `'SCO'`) denote both the expected file extension and the expected top-level directory where any translated output will by default be placed. (This information is used by the editor when offering the user a suitable filename to "run").

On the Series 3a a final `'*'` character may be added to the alias info. This has the effect of adding an "S3 Translate" menu option.

The remaining modes are not available on Series 3 machines.

Alias info that consists of a single `'$'` character selects a plain text editing mode. In this case the program-related menu options are suppressed, with only an "indentation" option being offered.

A variant on the *Word* mode is set by alias info that consists of a single `'/'` character. This behaves in a similar way to the *Word* application, with the exception that a specific template file is loaded whenever a new file is created. The template must have the same name as the aliased application and must be located on the current drive at the time the new file is created. Thus, an alias created from the following *.ma* file:

```
Letter.LET
\LET\
1083
Word
/
```

would, on creation of a new file, automatically load the template file *\wdr\letter.wrt*, provided it exists on the current drive. Note that, in this mode, the value `80` *must* be added into the application type number. If it is not, the automatic loading of the template is disabled.

How aliasing works

Part of the mechanism of aliasing is handled by the System Screen:

- creating a new file list
- listing the appropriate files in the new file list
- allowing the user to assign a new application button to the new file list
- creating a suitable command line to pass to the relevant application, when the user chooses to start an instance of the alias (by pressing ENTER on an entry in the file list).

However, other parts of the mechanism of aliasing rely on the application paying suitable attention to the details of the command line passed to it. Failure to do this will diminish the effect.

Thus even passive aliasing relies on some cooperation from the application being aliased. For example, an application that is determined that it knows what its public name is (say *Word*) and which writes this to `DatProcessNamePtr` (see below) in all cases, despite any different public name being passed to it on the command line, will frustrate the intent of any aliasing application:

- any application button assigned to the alias by the user will be ineffective
- running instances of the alias will appear (in bold) in the wrong file list in the System Screen.

This is just one reason why all serious applications should analyse the command line passed to them, as part of their initialisation procedures.

There are routines in both the Hwif library and the Hwim dyl to assist in analysing the command line.

Epoc reserved statics

The values of the Epoc reserved statics `DatProcessNamePtr`, `DatLocked`, `DatStatusNamePtr`, and `DatUsedPathNamePtr` all have special significance for Series 3 applications. The values of these variables for different applications are read at various times by the System Screen and also by the Window Server. An application which fails to write suitable data to these statics may find that:

- an incorrect name is displayed in any status window shown in the application
- instances of the application are shown in the wrong file list in the System Screen
- *Shutdown* or *Switchfiles* messages arrive at inopportune moments from the System Screen (see below for more on these messages)
- assigning an application button to the application in the System Screen has no effect.

In general, applications should write to these reserved statics:

- on initialisation (after having analysed the contents of their command line)
- whenever a new file is opened
- whenever the application is about to go "busy" over an extended period of time.

There are routines in both the Hwif library and the Hwim dyl that assist with keeping these reserved statics up to date.

While debugging using the Sibo Debugger, the values of reserved statics can be determined by using the "Magic Statics" menu command.

DatProcessNamePtr (0x22)

This static is read by the System Screen when deciding which file lists bolded running applications should be placed into. It is also read by the Window Server when deciding which action to take when an application button is pressed. Finally, it is read by the System Screen in response to any "Quit application" menu commands, to determine how to implement this request (ie how much cooperation the System Screen might expect from the application).

The way the file lists are built in the System Screen is as follows:

- for each list, the set of all eligible *files* is compiled; these will all be displayed non-bolded
- then for each running application, it is decided which file list the application belongs to
- this involves reading the value of `DatProcessNamePtr` for the application
- further, for each running application, the *name* of the file currently open (if any) is decided
- this involves reading the value of `DatUsedPathNamePtr` for the application (and possibly also the value of `DatProcessNamePtr`)
- if this name matches any entry in the file list, that entry is removed (so that it is no longer displayed non-bolded)
- the name of the open file is added to the list, in bold.

Clearly, the lists will be misleading if the running application is assigned to the wrong list.

The rules for assigning a running application to a particular file list are straightforward:

- the preferred public name of the application is read from `DatProcessNamePtr`
- if this matches the public name of any existing file list, the application is assigned to that list
- otherwise, the application is assigned to the *RunImg* list.

More on the file lists in the System Screen

Incidentally, any entry starting with *Sys$* is never displayed in any file list. Further, the name *Link* is never displayed in the *RunImg* list. These rules prevent the display of private system processes within the System Screen file lists.

Additionally, files with the "hidden" attribute set are never displayed in a file list in the System Screen - unless the file is open within an application (in which case it will be displayed in bold).

In order to check for the existence of hidden files or file starting with *Sys$* in a directory, the user should press TAB to enter "directory" mode of the System Screen.

It is also possible to task to an application whose open file starts with *Sys$* by repeatedly pressing the SHIFT+SYSTEM key combination, which tasks round *all* running applications (that are clients of the Window Server).

Assigning application buttons

Suppose that the user has installed the application *Tele*, and has assigned the application button CONTROL+WORD to it. The following is what happens when the user presses CONTROL+WORD:

- at all times, the System Screen maintains a data structure associating each of the 14 possible application buttons to public names of applications
- the address of this data structure, within the System Screen dataspace, is known to the Window Server (in fact it is kept at `DatApp1`)
- when CONTROL+WORD is pressed, the Window Server consults this data to determine the public name that is currently associated with this application button (ie *Tele* in this example)
- the Window Server next checks whether the public name of the current foreground application matches *Tele*, reading the public name from `DatProcessNamePtr`
- if so, this application is sent a special key-press event, with keycode value equal to `W_KEY_MODE` (as defined in *wskeys.h*) - unless the SHIFT modifier is also held down, in which case the algorithm continues as below
- otherwise, the clients of the Window Server are scanned in current task order, to see whether any can be found with the required public name
- if any can be found, this is made foreground
- failing this, a message is sent to the System Screen to position, if possible, to the file list associated with the given public name
- if no such file list exists, the System Screen beeps and gives a suitable error message.

The crucial point in this is that, once again, the public name of the application has to be written to `DatProcessNamePtr`.

Incidentally, it is now clear why pressing the CONTROL+SYSTEM key (assigned to *RunImg*), or any other application button assigned to a pure file list application, often fails to have the desired effect (of bringing to foreground a running application listed in the relevant file list). The point is that these applications are generally run without any command line being passed to them, and so they cannot set up a suitable value at `DatProcessNamePtr` merely by analysing their command lines.

Also note that the assigned buttons differ in one aspect of their behaviour depending on the machine used. Consider an application, the built-in database say, that is currently running in the foreground. On the Series 3 pressing the Data button would change the application from search mode into change mode, and, on a second press, back into search mode. On the Series 3a pressing the Data button has no effect when there is currently only one copy of the application running. However when multiple copies are running, then pressing the Data button has the effect of sequentially bringing each copy into foreground - simultaneously holding down the shift key reverses the order of bringing into foreground. Try out the Data button while running multiple copies of the database ...

DatUsedPathNamePtr (0x3e)

The Epoc reserved static `DatUsedPathNamePtr` is read solely by the System Screen, which assumes that if it is non-null for an application, `DatUsedPathNamePtr` points to a full path specification of the file currently open in the application. As described above in the section on `DatProcessNamePtr`, these filenames are used when generating the file lists in the System Screen:

- any open file matching an entry in the non-bold section of the file list *replaces* that entry
- the filename is parsed and rearranged, eg from the form *LOC::A:\WRD\SHOPPING.WRD* into *Shopping[A]*.

In case DatUsedPathNamePtr is null, the value of the string at DatProcessNamePtr (if any) is used instead: failing that, the process name (as returned by p_pname) is used.

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

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

DatStatusNamePtr (0x3c)

The Epoc reserved static DatStatusNamePtr is used to determine which text string should be displayed as the name of the application in any status window shown for that application.

If non-zero, this is assumed to point to a string giving the text to use, with the text being clipped at the first dot encountered, and in any case after eight characters. The text is also converted into standard capitalised form. Thus if DatStatusNamePtr points to "DIARY.AGN", the text *Diary* will be displayed in the status window.

The rules for what text to display when DatStatusNamePtr is null are the same as those employed when DatUsedPathNamePtr is null (see above).

DatLocked (0x3a)

When the user attempts to terminate an application using the "Quit application" command in the System Screen, or to change the file currently open, by pressing ENTER on another entry in the file list for that application, the System Screen checks the value of the Epoc reserved static DatLocked for that application.

If this is non-zero, a message *Application is busy* is displayed, and the user's request is refused.

Applications which enter a state in which they are unable to respond to such requests from the System Screen should accordingly set DatLocked to TRUE. Good programming practice dictates that DatLocked be set back to FALSE again as soon as possible afterwards.

The Series 3 command line

The command line communicates the following information to a Series 3 application about to start:

- the *public name* of the application
- the *default extension* for files used, if any
- any *alias information* specified in an alias file
- the full path name of the file to open, if any
- whether this file should be opened or created anew
- exceptionally, whether the application is to connect to the Window Server *in background*.

When a program starts, its command line is placed in an allocated cell within the heap of the application, with the address of this cell being written to the Epoc reserved static DatCommandPtr. See the section on p_execc in the *Plib Reference* manual for some general information about DatCommandPtr.

The command line for *any* Epoc program always starts with a zero-terminated string given the full path name of the process being run. The byte after this gives the length of any following data. Ordinarily, when referring to "the command line", it is this latter data that is in mind.

For example, suppose the user presses TAB inside the *Prog* file list in the System Screen, navigates using the file selector to *loc::m:\dat\data.dbf*, and then presses ENTER. The full command line passed to the application thereby chosen (*Word*) is as follows:

```
ROM::WORD.APP<0><29>OProgram<0>.OPL OROPO<0>LOC::M:\DAT\DATA.DBF<0>
```

The <29> immediately following the zero at the end of the first zero terminated string indicates that the remainder of the command line is 0x29 bytes long - as is indeed the case.

The next byte after this is the so-called *command byte*:

- a command byte of 'O' means, for a file-based application, that a named file is to be opened
- a command byte of 'C' means, for a file-based application, that a named file is to be created
- a command byte of 'D' means the application is to connect to the Window Server in background.

A command byte of 'D' arises only for the built-in applications, and is not considered in the remainder of this documentation. (When it *does* arise, it is handled automatically by code in *hwim.dyl*, which silently translates it into one of the other two cases.)

Following the command byte, there is a zero-terminated string giving the public name of the application.

After this comes another zero-terminated string, containing both the default extension and (if present) the alias info. The alias info, if present, is separated from the default extension by a space.

Finally, yet another zero-terminated string gives the full path name of the file to open or create.

With regard to the above example:

- The command byte of the application is 'O'
- The public name of the application is "Program"
- The default extension is ".OPL"
- The alias info is "OROPO"
- The name of the file to open is "LOC::M:\DAT\DATA.DBF".

Summary of command line format

In summary, the format of the command line of a Series 3 application is as follows:

```
<command byte><public name><0>[<default extension>[<space><alias info>]<0><full
pathname><0>]
```

Supplying a command line from the SIBO Debugger

Ordinarily, applications are executed from the System Screen, which automatically constructs a suitable command line.

When executing an application from the Debugger (or from an alternative "Shell" program), the command line has to be supplied explicitly. Some examples follow:

- SDBG TELE "CTELE",0,"LOC::M:\TEL\TELE.TEL" - debug the application *tele* giving it the public name Tele, and have it Create the file LOC::M:\TEL\TELE.TEL on start up
- SDBG DA2 "ODAYS",0,"LOC::A:\ANN\DAYS.ANN" - debug the application *da2* giving it the public name Days, and have it Open the file LOC::A:\ANN\DAYS.ANN on start up
- SDBG JO1 "CJOKER",0,0 - debug the application *jo1* (which is not file-based), giving it the public name Joker.

These examples take advantage of the following processing of the command line by the Sibo Debugger:

- parameters entered as a string ("...") are passed on to the program with a zero terminating the string
- parameters given in numeric form (eg 0) are passed on to the program as single bytes
- adjacent parameters separated by commas are concatenated.

One drawback of the command line processing of the Sibo Debugger should be pointed out: everything is automatically upper-cased. This means that if an application button has, for example, been assigned to the public name *Days*, no application run in this way from the Sibo Debugger will ever be tasked to as a result of the user pressing the corresponding application button (for public names are in general *case-sensitive*).

From command line to reserved statics

As mentioned earlier in this chapter, an application should analyse its command line on start-up, and should write various values from this command line into Epoc reserved statics as a result.

As an example of how this could be done, there follows the source code for two Hwif routines:

```
GLDEF_C VOID hSetUpStatusNames(TEXT *pb)
    {
    TEXT buf[P_FNAMESIZE];
    P_FPARSE crk;

    DatUsedPathNamePtr=pb;
    p_fparse(pb,0,&buf[0],&crk);
    DatStatusNamePtr=pb+P_FSYSNAMESIZE+crk.device+crk.path;
    }

GLDEF_C INT hCrackCommandLine(VOID)
    {
    INT ret;
    TEXT *pb;

    pb=DatCommandPtr;
    pb+=p_slen(pb)+1;
    if (!*pb++)
        ret=0;
    else
        {
        ret=(*pb++);
        DatProcessNamePtr=pb;
        pb+=p_slen(pb)+1;
        pb+=p_slen(pb)+1;
        if (*pb)
            hSetUpStatusNames(pb);
        else
            DatStatusNamePtr=DatProcessNamePtr;
        }
    return(ret);
    }
```

For details of the `DatCommandPtr` and other reserved statics see the *Processes and Inter-Process Messaging* chapter of the *Plib Reference* manual.

Applications that disregard their command line

Simple applications - especially those that are not file based - have no need to pay any attention to the command line passed to them by the System Screen. In this case, the various relevant Epoc statics are left at their default (zero) values. This fact is picked up by the System Screen and by other parts of the OS, with the following results:

- The name displayed in any status window and in the file list in the System Screen is just that of the application *.app* file
- If the user requests the application to be shut down, from the System Screen, the application is shut down by the OS, without the application itself being informed of this fact (just as if the user had selected the *Kill* option in the System Screen).

In case an application wishes to do its own processing in response to a *Shutdown* request issued by the user in the System Screen, it must therefore make a call to a routine such as `hCrackCommandLine` during its initialisation. This is true even if the application is not file-based.

One final drawback of an application not processing its command line is that users will be unable to assign application buttons with any effect to that application. Suppose a user assigns CONTROL+WORLD to a version of the *Spy* application, for example, that fails to write anything suitable to `DatProcessNamePtr`. If the user subsequently presses the key combination CONTROL+WORLD, the *Spy* application will fail to be brought into foreground - thus spoiling the whole purpose of assigning the application button.

Creating directories when required

In contrast to file selectors on other systems, those on the Series 3 allow users to specify paths that do not yet exist. This can happen fairly commonly, for example as follows:

- The user inserts a brand new solid state disk into drive A
- A *Save as* or *New file* menu command is invoked
- The user adjusts the disc selector to this new disk
- The user types eg "Backup" into the filename editor.

Then assuming the default directory for the application is *\DIR* and the default extension is *.EXT*, the filename returned to the application is

LOC::A:\DIR\BACKUP.EXT

even though the directory *\DIR* does not exist yet, on the specified disk.

It is the responsibility of application programs to test for this case and to create the required directories.

Messages from the System Screen

Shutdown messages

Series 3 applications can receive *Shutdown* messages from the System Screen, as an instruction to shut themselves down tidily, saving any changes to file as required.

These messages can arise when the user presses DELETE while highlighting a running task in the System Screen. However, as mentioned above, if the application has set its `DatLocked` to `TRUE`, the System Screen instead presents an *Application is busy* message.

Incidentally, applications are sent *Shutdown* messages only if they have a non-zero value of `DatProcessNamePtr`. The System Screen assumes that any application that has left this Epoc reserved static at its default (zero) value is unlikely to be prepared to respond to *Shutdown* messages. In that case, the System Screen instead calls `p_pterminate` to terminate the application.

Finally, note that any application which has `4000` included in its application type number will *never* be sent a *Shutdown* message from the System Screen; instead, the System Screen will display the message *Cannot quit application*. Note that this blocking mechanism was designed for internal use only, and should not be used without good reason.

Switchfiles messages

Series 3 applications can receive *Switchfiles* messages from the System Screen, as an instruction to close down their existing open file, and to open or create another one.

These messages can arise when the user presses ENTER while highlighting a file within that application's file list in the System Screen. However, if the application has set its `DatLocked` to `TRUE`, the System Screen instead presents an *Application is busy* message.

Switchfile messages will only ever be sent to applications with basic type number `2` or `3`. Applications whose type numbers include `8000` can receive *Switchfile* messages of the Open sort, but not of the Create sort.

How messages from the System Screen are received

There are two parts to an applications receiving a *Shutdown* or *Switchfiles* message from the System Screen:

- the application receives notification that *some* message from the System Screen has been sent
- the application calls `wGetCommand` to determine the contents of the message.

In turn, the initial notification can be received in either of two ways, depending on whether the application is receiving events from the Window Server directly (by calling `wGetEvent` or a variant), or as "extended keypresses" via the console device (as occurs for example in Hwif programs):

- for the protocol in the first case, see the discussion on `WM_COMMAND` in the *Window Server Reference* manual
- for the protocol in the second case, see the discussion on `P_EVENT_READ` in the *Console* chapter of the *I/O Devices Reference* manual.

In either case, the initial event prompts the application to call `wGetCommand`, to obtain the so-called *new command line* giving more details about the event.

Contents of the new command line for System Screen messages

The parameter passed to `wGetCommand` must be the address of a buffer having at least `P_FNAMESIZE` (`128`) bytes. The new command line is written into this buffer.

The first byte of the new command line will be one of `'X'`, `'O'`, or `'C'`:

`'X'` means the command is a *Shutdown* message

`'O'` means the command is a *Switchfiles* message, with a specified file to be opened

`'C'` means the command is a *Switchfiles* message, with a specified file to be created.

In the case of *Switchfiles* messages, the remainder of the new command line gives the full pathname of the file to open or create.

An example of code that responds to notification of a message from the System Screen is as follows:

```
LOCAL_C VOID ProcessSystemCommand(VOID)
    {
    UBYTE buf[P_FNAMESIZE];

    wGetCommand(&buf[0]);
    if (buf[0]=='X')
        ExitApplication();
    SaveIfChanged();
    if (buf[0]=='C')
        CreateNewFile(&buf[1])     /* remainder of message is ZTS of file to create */
    else        /* buf[0]=='O' */
        OpenExistingFile(&buf[1]);   /* remainder of message is ZTS of file to open */
    }
```

Other possible types of messages

At the time of writing, the command byte of new command lines (as read by `wGetCommand`) is restricted to one of the three values `'X'`, `'O'`, or `'C'`. It is possible, however, that some future application might send messages to other applications having different command bytes.

These messages would be sent by means of the `wSendCommand` function, as described in the *Window Server Reference* manual. (Note that these messages are *not* the same thing as IPC (inter-process communication) messages; nor are they the same as object-oriented messages.)

In order to be future proof, an application should arguably test explicitly for all expected values of the command byte, and should ignore values other than those expected. The above code fragment would therefore need to be modified.

Multi-lingual aliasing of Word.app

This section can be omitted by all readers, except those who wish to write a multi-lingual alias of *Word.app*.

The reason why *.als* files as created by *makeals.exe* cannot be used in this case is that these files have to contain the public name of the application being aliased. However, the public name of *Word.app* can vary from language to language, and there is no facility to track this within an ordinary *.als* file.

To surmount this problem, the *.als* file can be re-written *as a program*.

For example, the following code provides a successful multi-lingual alias for *Word.app*:

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

#define R_STRARRAY_APPNAMES 78

GLREF_D UBYTE *DatCommandPtr;

GLDEF_D TEXT olibDyl[]="OLIB.DYL";
GLDEF_D TEXT shellImg[]="ROM::SYS$SHLL.IMG";
GLDEF_D TEXT wordNameFmt[]="ROM::%s.APP";
GLDEF_D TEXT aliasInfo[]={'S','R','S','C','O'};

LOCAL_C TEXT *skipStr(TEXT *p)
    {
    return(p+p_slen(p)+1);
    }

#pragma save,ENTER_CALL

LOCAL_C INT getWordFspec(
/*
Get full file spec of Word.APP using shell's resource file.
Returns 0 if successful, leaves if error.
*/
    TEXT *fSpec)         /* To receive name */
    {
    TEXT *pAppNames;
    VOID *rsc;
    HANDLE cat;

    p_findlib(&olibDyl[0],&cat);
    rsc=f_newlibh(cat,C_RSCFILE);
    p_send3(rsc,O_RS_INIT,&shellImg[0]);
    p_send4(rsc,O_RS_READ,R_STRARRAY_APPNAMES,&pAppNames);
    p_atos(fSpec,&wordNameFmt[0],skipStr(pAppNames+1)); /* Copy to 2nd string */
    p_free(pAppNames);
    p_send2(rsc,O_DESTROY);
    return(FALSE);
    }
```

```
#pragma restore

GLDEF_C VOID main(VOID)
    {
    TEXT fSpec[P_FNAMESIZE];
    UBYTE comBuf[E_MAX_COMMAND_BUFFER+1];
    TEXT *pCommand;
    TEXT *pAlias;
    TEXT *pEndAlias;
    TEXT *p;
    HANDLE pId;
    INT len;

    p=skipStr(DatCommandPtr);
    len=(*p);
    pCommand=(p+1);
    pAlias=skipStr(pCommand);
    pEndAlias=skipStr(pAlias)-1;        /* Point to end 0 */

    p=p_bcpy(&comBuf[0],pCommand,pEndAlias-pCommand);
    p=p_bcpy(p,&aliasInfo[0],sizeof(aliasInfo));
    p_bcpy(p,pEndAlias,len-(pEndAlias-pCommand));

    if ((pId=p_enter2(getWordFspec,&fSpec[0]))<0)
        goto fail;
    if ((pId=p_execc(&fSpec[0],&comBuf[0],len+5))<0) /* Run Word */
        goto fail;
    p_setpri(pId,p_getpri(p_getpid())-1);
    pId=p_presume(pId);                    /* Won't run till I've exited */
fail:
    p_exit(pId);
    }
```

This program uses the fact that the filename of the *Word.app* application is always stored in the 78th resource within the resource file of the shell application. (Hence the `#define` of `R_STRARRAY_APPNAMES` as `78`.) This resource actually contains an *array* of strings giving the filenames of the built-in applications, with the filename of *Word.app* as the second element in the array.

See the chapter *Resource Files* in the *Additional System Information* chapter for background on creating and using an instance of the `rscfile` class.

The program also analyses its own command line, and constructs a suitable one to pass on to *Word.app*. The detailed working of the program can be followed using the information given earlier in this chapter.

CHAPTER 3

ENHANCED SOUND OUTPUT

Introduction

The Series 3 and Series 3a provide distinct sets of sound services.

The Series 3 as supplied can emit only buzzer sounds, DTMF dialling tones, and simple alarm sounds. However, by loading a suitable device driver, such as *SNDFRC.LDD*, the machine can also be made to emit sequences of musical notes of variable duration, thus greatly extending its sound capabilities. The first section of this chapter describes use of the *SNDFRC.LDD* attached device driver from within a simple demonstration program.

The Series 3a has considerably greater sound capabilities than the Series 3. In addition to emitting buzzer sounds, DTMF dialling tones and simple alarm sounds, the Series 3a can play simultaneously two sequences of musical notes, and can play and record digital sound files - for details of playing and recording digital sound files see the *General System Services* chapter of the *Plib Reference* manual. The second section of this chapter describes a simple program that demonstrates the playing of sequences of notes using the built-in `SND:` device driver.

Warning: any attempt to load and use the *SNDFRC.LDD* attached device driver on the Series 3a is a serious error - the machine will in all probability hang, necessitating a soft reset.

Sound on the Series 3

Introduction

This section explains how to create a wider range of musical sound output, via the loudspeaker, than is possible by merely using the Series 3's built-in `SND:` device driver.

These services rely on a dynamic extension to the Series 3 operating system, known as a *loadable device driver*.

With this device driver installed, strings of sound covering two octaves in semitone intervals can be generated. Control is also possible over the duration and loudness of the notes emitted.

The sndfrc and snddvr device drivers

The chapter *Example Device Drivers* in the *Additional System Information* manual describes two different enhanced sound drivers, *sndfrc.ldd* and *snddvr.ldd*, from the point of view of how to write device drivers. The current chapter focuses on the question, not how to *write* these drivers, but how to *use* them.

In fact, this chapter only considers the driver *sndfrc.ldd*, which is arguably the superior of the two. See *Example Device Drivers* for a discussion on how the two device drivers differ.

Once this device driver file has been installed, a device with the name `MUS:` can be opened by applications.

Installing sndfrc.ldd

Any program which wishes to use the services of *sndfrc.ldd* needs to check, during its initialisation, that this driver has been installed. This is necessary because, in contrast to some other device drivers such as the serial port device driver and the basic sound device driver, the `MUS:` device driver is not built into the ROM of the Series 3.

The way to check the device driver is loaded is to make the call

```
p_loadldd("SNDFRC.LDD");
```

where the full path of the *.ldd* file can be given. (The *.ldd* file has to be copied onto the Series 3.)

The return values zero and `E_FILE_EXIST` can both happily be ignored. Other errors are more serious - they probably mean that the file *sndfrc.ldd* cannot be located. In this case, the program cannot continue (at least, not as according to its original intention).

Opening a channel to MUS:

Another pre-requisite to using the services of *sndfrc.ldd* is to open a channel to `MUS:`. This is done in the standard manner for all i/o devices:

```
p_open(&handle,"MUS:",-1);
```

If this call is successful, it writes back the handle of the channel established to the device driver. All subsequent requests from the program (until such time as the channel is closed) should be made via this handle.

Possible errors from the `p_open` call include:

- "invalid arguments" - which probably means *sndfrc.ldd* has not been installed (or, having once been installed, it has since been de-installed)
- "in use" or "locked" - another application is currently making use of the loudspeaker.

In the second of these two cases, a brief retry philosophy might be adopted. If the channel still cannot be opened, a suitable error message should be displayed - leaving it up to the user to retry at some later time.

Actually creating sounds

The way sounds are actually caused to be emitted is by using the `P_FWRITE` service of the `MUS:` channel.

As for all device drivers, the `P_FWRITE` request can be made synchronously (eg using the utility function `p_write`) or, for more quality applications, asynchronously. If the request is made asynchronously, it allows the use of the `P_FCANCEL` service to interrupt and terminate a sequence of notes as they are playing.

For example,

```
p_ioc5(handle,P_FWRITE,&musstat,&buf[0],&len);
```

to play a buffer of notes asynchronously.

The parameter `len` (passed by reference) gives the number of notes in the buffer. The maximum allowed value of `len` is 500. (For arbitrarily long sequences of notes, call the `P_FWRITE` service more than once.)

Each note is specified by one `UWORD` in the buffer - so that `buf` would be declared as

```
UWORD buf[ ]
```

For each note, the `UWORD` contains three pieces of information: *tone*, *length*, and *loudness*.

There are only four possible values of loudness: `0` (the quietest), `1`, `2`, and `3` (the loudest). The loudness is multiplied by `64` before being added into the `UWORD` for the note.

The duration is measured in 1/100ths of a second, and can have any value from `1` to `255`. The duration is multiplied by `256` before being added into the `UWORD` for the note.

The allowed values of tone range in principle from `0` to `0x3f`. See below for more details.

Example

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


GLDEF_C INT main(VOID)
    {
    INT ret;
    VOID *mcb;
    UWORD buf[10];
    UWORD len;
    WORD musstat;

    ret=p_loadldd("SNDFRC.LDD");
    if (ret && ret!=E_FILE_EXIST)
        return(ret);
    ret=p_open(&mcb,"MUS:",-1);
    if (ret)
        return(ret);
    buf[0]=0x30+(40<<8);
    buf[1]=0x32+(100<<8);
    buf[2]=0x34+(40<<8)+(1<<6);
    buf[3]=0x35+(100<<8)+(1<<6);
    buf[4]=0x37+(40<<8)+(2<<6);
    buf[5]=0x39+(100<<8)+(2<<6);
    buf[6]=0x29+(40<<8)+(3<<6);
    buf[7]=0x3b+(100<<8)+(3<<6);
    len=8;
    p_ioa5(mcb,P_FWRITE,&musstat,&buf[0],&len);
    p_iowait();
    return(0);
    }
```

This plays a scale of eight notes, with notes having wavering length and increasing loudness.

Possible tones

There are three types of tones that the Series 3 loudspeaker hardware can emit: DTMF tones, modem tones, and musical tones.

For standard dual DTMF tones, set the *tone* part of the `UWORD` to: `0x10` for DTMF digit `0`, `0x11` for digit `1`, ..., `0x19` for digit `9`, `0x1a` for "digit" `a`, ..., `0x1d` for "digit" `d`, `0x1e` for `*`, and `0x1f` for `#`.

For modem tones, `0x24` gives `1300 Hz`, `0x25` gives `2100 Hz`, then `1200`, `2200`, `980`, `1180`, `1070`, `1270`, `1650`, `1850`, `2025`, and `0x2f` gives `2225 Hz`.

As for musical tones, twenty five notes are possible, incrementing by semi-tones over a 2-octave interval from `D#5` to `D#7`. The corresponding tone values are `0x30`, `0x31`, `0x32`, `0x33`, `0x34`, `0x35`, `0x36`, `0x37`, `0x38`, `0x39`, `0x3a`, `0x29`, `0x3b`, `0x3c`, `0x3d`, `0x0e`, `0x3e`, `0x2c`, `0x3f`, `0x04`, `0x05`, `0x25`, `0x2f`, `0x06`, and `0x07`.

Pauses

In order to pause, in the middle of a buffer of notes, set the tone value to `0` for one note.

When to open and close MUS:

An application that makes use of `MUS:` services from time to time ought to call `p_close` to free the sound channel whenever it is not immediately needed. This allows other applications to make temporary use of the sound channel - eg for alarms or for standard DTMF dialling dialogs.

If you wrote a game which opened `MUS:` at its beginning, and only made sounds from time to time, and left this game in background while you went to the *Data* application to look up a telephone number, you would find the DTMF dialler would be unable to emit any sounds, and would report "Sound system in use" - even though the game is silent at the time.

Far better in these situations for a program to open `MUS:` just before it needs to use this channel, and then close it again immediately afterwards.

When to install and de-install the ldd file

Once *sndfrc.ldd* has been installed, it occupies about 1.7 K of RAM. For this reason, it would seem to be best to de-install it, when the application terminates. The way to do this is to call (see the *Plib Reference* manual for more details)

```
p_devdel("MUS:",E_LDD);
```

This call will fail if another application currently has an open channel to `MUS:`. Applications should ignore any errors from `p_devdel`.

Note however that if an application has:

- called `p_loadldd` to ensure `MUS:` can be found
- called `p_open` to open a channel to `MUS:`
- played some notes
- called `p_close` to free up the sound channel

then it cannot rely on `MUS:` still being installed if it calls `p_open` again at a later stage. For another application may have called `p_devdel`, successfully, in the meantime.

The upshot of this is that applications should call `p_loadldd` prior to *any* call to open a channel to `MUS:`.

Sound on the Series 3a

This section does not make reference to the recording and playing of digital sound - for details see the *General System Services* chapter of the *Plib Reference* manual.

The Series 3a's built-in `SND:` device driver can be used to simultaneously play two tunes on the built-in speaker. Although the sound quality is not as high as with digital sound files, the memory requirements are much less. For example to play a tune lasting six seconds would require a digital sound file of size 49,184 bytes. A comparable figure using the `SND:` device driver would be less than 1 Kb.

The demonstration program *sound.c* (in *\SIBOSDK\DEMO* on the supplied disks) plays two sequences of notes using both channels of the built-in `SND:` sound device driver (only one application can have access to these channels at any given time). The tune is the so-called "ice cream van" tune that you may already have met in the *Sound* chapter of the *i/o Devices* manual - it is in any case recommended that you read that chapter before proceeding.

The program demonstrates the following:

- the opening and closing of a channel to the `SND:` device driver.
- the sensing and setting of the volume level and the number of beats per minute.
- the writing of notes to the two sound channels.

A number of points are worth making:

- a side effect of opening a channel to the `SND:` device driver is to power up the speaker. As a consequence the `SND:` channel should be closed as soon as the sound has been played - failure to do so could unnecessarily drain the batteries.
- the `SND:` device driver, and hence the speaker, can only be used by one application at a time. As alarms and keyclicks will be disabled well written programs should close the `SND:` channel as soon as the sound has been played.
- a serious of notes separated by silences can be created by setting the frequency to zero during the silent periods.
- the sound will not play until a `P_FSSOUNDCHANNELn` request has been made on *both* channels. The playing of sound on the two channels is thus automatically synchronised.
- *both* `P_FSSOUNDCHANNELn` requests *must* be made asynchronously using `p_ioc` or the `p_ioc5` variant. On a low battery the request will fail to complete with an error message written to the status word. Use of `p_iow`, `p_iow4`, `p_ioa` and/or `p_ioa5` would hang the machine on a low battery - a very serious programming error.

- sound can be played on only one channel by passing the other channel a length of zero for the note buffer - i.e. zero notes.
- the *P_FSET* service sets *both* the volume and the beats per minute. The *P_FSENSE* service can be requested first to ensure that one or other parameter remains unchanged.

To create a *sound.img* file simply type `make sound` in the appropriate directory. This file can then be copied to a `m:\img` directory on the Series 3a and run via the RunImg application in the usual manner.

The code in *sound.c* is as follows:

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

GLDEF_C VOID waitstat2(WORD *pstat1, WORD *pstat2)

/* Wait for *pstat!=E_FILE_PENDING and *pstat2 != E_FILE_PENDING   */

    {
    INT i;

    i = -1;
    do
        {
        p_iowait();
        i++;
        }
        while (*pstat1 == E_FILE_PENDING && *pstat2 == E_FILE_PENDING);

    if (*pstat2 == E_FILE_PENDING)
        pstat1 = pstat2;
    p_waitstat(pstat1);

    while (i--)
        p_iosignal();
    }

GLDEF_C VOID play_notes(WORD *buf1, WORD *buf2, WORD l1, WORD l2, INT volume, INT
beatsPerMinute)

    {
    VOID *pcb;
    WORD sndstat1,sndstat2;
    E_SOUND sound;
    INT err;

    if ((err=p_open(&pcb,"SND:",-1))<0)
        {
        p_close(pcb);
        p_exit(err);
        }

    if ((err=p_iow3(pcb,P_FSENSE,&sound))<0)
        {
        p_close(pcb);
        p_exit(err);
        }

    if (beatsPerMinute >= 0)
        sound.beatsPerMinute = (UBYTE) beatsPerMinute;

    if (volume >= 0)
        sound.volume = (UBYTE) volume;

    if ((err=p_iow3(pcb,P_FSET,&sound))<0)
        {
        p_close(pcb);
        p_exit(err);
        }
```

```
        p_ioc5(pcb,E_FSSOUNDCHANNEL1,&sndstat1,&buf1[0],&l1);
        p_ioc5(pcb,E_FSSOUNDCHANNEL2,&sndstat2,&buf2[0],&l2);
        waitstat2(&sndstat1,&sndstat2);

        p_close(pcb);

        if (sndstat1 != 0 || sndstat1 != 0)
            p_exit(0));

        }

    GLDEF_C INT main(VOID)

        {
        WORD notes1[] = {1048,24,524,12};
        WORD notes2[] = {1048,4,1320,4,1568,4,2092,4,1568,4,1320,4,1048,12};
        WORD len1 = sizeof(notes1)/4,len2 = sizeof(notes2)/4;
        INT i;

        for (i = 0; i < 6; i++)
            {
            play_notes(&notes1[0],&notes2[0],len1,len2,i,-1)
            p_sleep(1);
            }

        for (i = 0; i < 6; i++)
            {
            play_notes(&notes1[0],&notes2[0],len1,len2,-1,140 + i*20)
            p_sleep(1);
            }
        return(0);
        }
```

The `main` routine initialises the note buffers, then repeatedly passes the buffers, the buffer lengths, the volume and the beats per minute, to the subroutine `play_sound` that plays the tune. The tune is repeated first at the default beats per minute for all six allowed volume levels, and then at the default volume for six values of the beats per minute. The default is specified by passing a negative integer for the volume and/or the beats per minute.

The subroutine `play_sound` opens the `SND:` channel, senses and sets the volume and beats per minute, plays the notes and closes the `SND:` channel. In the case of an error in, for example, sensing, `play_sound` closes the `SND:` channel and returns with an error code. As mentioned earlier it is essential that the `P_FSSOUNDCHANNELn` requests be made asynchronously using either `p_ioc` or `p_ioc5` Plib library routines, as these guarantee completion even in the event of low batteries.

A large fraction of the code in `play_sound` is concerned with error checking in order to ensure that the routine behaves in a sociable way. In particular the `SND:` channel is closed as soon as an error is discovered so to conserve power - this is essential when running on batteries and such measures should be standard in any quality application.

The subroutine `waitstat` waits on the process i/o semaphore until completion of the two asynchronous requests specified by the `pstat1` and `pstat2` status words (for further details of such matters see the *Asynchronous Requests and Semaphores* chapter of the *Plib Reference* library). It is in fact a version of the Plib library routine `p_waitstat` that waits on two status words rather than one.

The `return(0)` statement at the end of the main routine informs the Series 3a that the program has ended normally: it can also be omitted entirely. Use of the `return` statement with no return value is not recommended: in practice this will return a random error code possibly leading to the display of a spurious full screen error message.

CHAPTER 4

USE OF SPY.APP

Introduction

This chapter describes the *Spy* application for the Series 3. This application contains many features that may help to "debug" problems with applications on the Series 3.

The version of *Spy* described in this chapter is suitable for use on both the Series 3 and the Work*about*, and can also be used on the Series 3a. A built version of *spy.app* that is specifically designed for use on the Series 3a will be found, following installation of the core SDK software, in the *\sibosdk\s3atool* directory.

Building spy.app

The *Spy* application is released in source form, as one of the Hwif demonstration programs.

To build it, proceed in the same way as to build any of the other Hwif demonstration programs:

- move into the *\sibosdk\hwdemo* directory
- type `make spy`
- the resulting image file *spy.img* can be renamed to *spy.app* and copied into a *\app* directory on the Series 3 or the Work*about*
- the application can be installed in the System Screen and, on the Series 3, even assigned an application button - eg CONTROL+WORLD.

The main display

The main display is a scrolling list of processes currently running on the Series 3. The *Change processes* menu option allows customisation of which processes are shown. "System" processes are simply ones whose names start with "Sys$", and include:

- *sys$shll* - which the user sees as the System Screen
- *sys$wsrv* - the Window Server, which coordinates access to the screen and keyboard
- *sys$fsrv* - the File Server, which coordinates access to the filing systems
- *sys$mang* - the Manager, which keeps track of all resources used by processes (so that, for example, they can be properly tidied whenever processes exit)
- *sys$ncp* - the "brains" behind Remote Link (when it is running).

The Null process, *sys$null*, which performs the vital task of switching the Series 3 off following sufficient inactivity, is omitted from the list displayed, for various technical reasons.

First letter matching works in the main window, so that eg pressing 'C' enough times will position the highlight to the *Calc* process.

Arrows are drawn in the top right and bottom right corner, Agenda-wise, whenever there are more processes beyond the visible boundaries of the list.

The data displayed is updated every time *Spy* comes into foreground, and also whenever the *Update* menu option is selected. By default, it is also updated regularly on a timer, though this can be disabled by a menu option. The *Refresh rate* option governs how frequently updates take place, when the timer is enabled.

There are in all twelve pieces of data that can be displayed for each process, but only three of these can be seen at any one time. Use the *Change data* menu option to choose which.

Many of the data items can be meaningfully displayed either in Hex or in Decimal. Another menu option controls this.

Heap statistics

Five of the twelve possible items of data concern the allocator heap of the process. Each process has its own heap, which can vary in size according to the needs of the program. Thus a Word Processor editing a large document will typically have a larger heap than a Word Processor editing a smaller document.

Each heap is divided into "alloced cells" and "free cells". The items "Cells allocated" and "Cells free" count these, and the items "Bytes allocated" and "Bytes free" sum how many bytes belong in each category.

This data can be of great help in developing applications. It is of course vital that an application frees cells it no longer requires - otherwise these cells go to what is called "alloc heaven". Something to watch for in particular is alloc heaven following an out-of-memory failure. Typically, a process such as launching a dialog involves a number of different allocs; if any one of these fails, all the allocs which have already succeeded must be undone. System code provides mechanisms such as "automatic destruction" and "automated clean-up" to help applications here, but applications can use these incorrectly at times - hence the need for real-time checking.

Stack statistics

Whenever a process starts, its stack is filled up with `0xFF`'s. This makes it easy to see how much stack has been used, at any one time. Quality programs need to avoid having too large a stack - the built-in applications default to a stack of `0xa00`. On the other hand, the operating system panics them (panic 69) if it ever discovers that their stack is less than `0x100`. This is because whenever an interrupt occurs, it runs in the stack of the current process.

The *Reset least stack* menu option simply refills the bottom of the process's stack with `0xFF`'s (ie up to its present stack pointer).

Segment statistics

The "Segment size" of a process gives the size of its data segment - which consists of the heap, static data private to the application, the stack, and finally the Epoc reserved statics at the bottom end. The quoted "Segment size" of an application can sometimes give a misleading account of how much memory it is actually using - since there are free cells as well as alloced cells in the heap. From time to time, the operating system may try to compress these heaps, but it can only do this by removing any free cells *at the end* of the heap. Applications should strive to avoid ending up with large free cells in the middle of their heap - though this is a very difficult goal to achieve.

Tests for heap integrity

Whenever *Spy* collects heap statistics for an application, it also checks the heap integrity. Any defect (caused for example by writing beyond the end of an alloc cell, or freeing a cell that was never alloced) results in an immediate alert. This alert helps to pin-point problems which would otherwise only rear their head much later - long after the real damage has been done.

Process priorities

The Process Priority gives the pecking order of the processes, as regards gaining CPU from the multi-tasking scheduler. Most applications the user sees run at `0x80` when in foreground, and at `0x70` when in background. This prevents computationally busy background tasks from detracting from the performance of the foreground task.

Spy momentarily ups its own priority to a massive `0xc0` (the maximum allowed to non-OS processes) whenever it collects heap statistics from other processes, to lessen the chances of other processes manipulating their heaps at the same time as *Spy* is walking through them. Occasionally, *Spy* will find that a heap is momentarily marked as "locked" when it tries to survey it - this indicating that the Operating System is busy doing something there - in which case the heap statistics will all just be shown as `0` for that process.

Other data

The "Process ID" of a process is essentially the address of the control block of the application in the Operating System data space, although the top nibble reflects how many times that same slot has been re-used since the last reset (the top nibble will therefore always be zero for sys$mang, sys$fsrv, and sys$wsrv). When processes talk to each other, for example in conjunction with the *Bring* menu option, they need to know each other's PID ("Process ID").

The IO Semaphore count basically keeps track of how many outstanding events a process has to respond to. This will usually be -1 or zero, but if you task to the System Screen and then straight back to *Spy* again, you may see the count for sys$shll momentarily go as high as three.

Logging Window Server statistics

The menu command *Log client* produces information in text file form as to the structure of the windows, GCs, fonts, bitmaps, and other Window Server objects "owned" by an application. This information may of use in determining why certain drawing fails to appear on the screen. For example, it may be that a window is positioned wrongly, that the window is obscured by another, or that the current GC is set up incorrectly - any such failure can be seen from the log file.

The menu command *Log all clients* repeats this process for *all* the clients of the Window Server.

The *Log client* command can, in effect, be invoked even when *Spy* is in background. Just press the key combination SHIFT+CONTROL+PSION+N and a dump of the Window Server object usage of the foreground application will be created - by default in the file *\opl\wsreport.lis*. (This feature works because *Spy* has "captured" this key combination.)

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 Series 3a Technical Specification

Physical characteristics

Part numbers:	1600-0029-10 (256KB) 1600-0025-10 (512KB) 1600-0080-10 (1MB) 1600-0082-10 (2MB)
Size	165mm x 85mm x 22mm (6.5"x 3.0"x 0.9").
Weight	275g (including batteries).
Screen	480 x 160 pixel high contrast retardation film LCD. Size 131.6mm x 45.2mm, (4.915" x 1.637"). Pixel pitch 0.26mm x 0.26mm Pixel size 0.23mm x 0.23mm.
Keyboard	58 key, QWERTY layout, computer style keyboard (UK models). 8 touch sensitive icon buttons for application selection.
Sound and recording	Loudspeaker with DTMF dialling and digital sound playback. Microphone for digital sound recording.

Power supply

Internal	2 x AA batteries.
Backup	3V Lithium CR1620 battery.
External	A Psion Series 3 mains adaptor, (9-11V 250mA), (Note: earlier models 175mA). Vehicle power adaptor (24v & 12v cigarette lighter sockets), from Jan/Feb 1997.

Memory

Built in	2MB or 1MB Masked ROM and 2MB, 1MB, 512KB or 256KB RAM. Two SSD drives allow extra storage space on Flash/RAM SSDs, up to 8MB.

System information

Processor	NEC V30H running at 7.68MHz.
Operating system	EPOC. Microsoft MS-DOS compatible Flash Filing System.

Expansion

Peripherals	External peripherals (such as a modems and printers) can be connected via a fast serial interface (1.536 Mbits/sec), which accepts optional 3Link parallel and serial interfaces. The Serial 3Link allows communication with other computers.
	PC Card adapter. SMS cables for Nokia and Orange mobile phones.
	3Fax fax modem, (superseded by PC Card adapter).
Socket:	6-way two row rectangular (male, with retracting protective cover). See the section ‘The 15-way SIBO/RS232 connector' in the 'SIBO Expansion Ports' chapter in the *Hardware Reference* manual for pinout details.

Environment

Operating temperatures	0ºC to 50ºC.
EMC	FCC Part 15 Class B; CE Mark

Psion Series 3/3s Technical Specification

These models are no longer in production.

Physical characteristics

Size	165mm x 85mm x 22mm (6.5"x 3.0"x 0.9").
Weight	265g (including batteries).
Screen	240 x 80 pixel high contrast retardation film LCD. Size 97.3mm x 38.9mm. Pixel pitch 0.385mm x 0.43mm Pixel size 0.355mm x 0.4mm.
Keyboard	58 key, QWERTY layout, computer style keyboard (UK models). 8 touch sensitive icon buttons for application selection.
Sound	Piezo buzzer. Loudspeaker with DTMF dialing.

Power supply

Internal	2 x AA batteries.
Backup	3V Lithium CR1620 battery.
External	A Psion Series 3 mains adaptor, (9-11V 250mA), (Note: earlier models 175mA). Vehicle power adaptor (24v & 12v cigarette lighter sockets), from Jan/Feb 1997.

Memory

Built in	Series 3 has 384KB or 512KB Masked ROM and 128KB or 256KB RAM, Series 3s has 512KB Masked ROM and 256KB RAM.
	Two SSD drives allow extra storage space on Flash/RAM SSDs, up to 8MB.

System information

Processor	NEC V30H running at 3.84MHz.
Operating system	EPOC. Microsoft MS-DOS compatible Flash Filing System.

Expansion

Peripherals	External peripherals (such as a modems and printers) can be connected via the fast serial interface (1.536 Mbits/sec), which accepts optional 3Link parallel and serial interfaces. The Serial 3Link allows communication with other computers.
Socket:	6-way two row rectangular (male, with retracting protective cover). See the 'Reduced External Expansion' section of the 'SIBO Expansion Ports' chapter in the *Hardware Reference* manual for pinout details.

Environment

Operating temperatures:	0ºC to 50ºC.
EMC:	FCC Class B; EN55022 Class B

Psion Series 3c Technical Specification

Physical characteristics

Part numbers:	1600-0126-10 (1MB) 1600-0122-10 (2MB)
Size	165mm x 85mm x 22mm (6.5"x 3.0"x 0.9").
Weight	275g (including batteries).
Screen	480 x 160 pixel high contrast retardation film LCD. Size 131.6mm x 45.2mm, (4.915" x 1.637"). Pixel pitch 0.26mm x 0.26mm Pixel size 0.23mm x 0.23mm.
Keyboard	58 key, QWERTY layout, computer style keyboard (UK models). 9 touch sensitive icon buttons for application selection.
Sound and recording	Loudspeaker with DTMF dialing and digital sound playback. Microphone for digital sound recording.

Power supply

Internal	2 x AA batteries.
Backup	3V Lithium CR1620 battery.
External	A Psion Series 3 mains adaptor, (9-11V 250mA), (Note: earlier models 175mA). Vehicle power adaptor (24v & 12v cigarette lighter sockets), from Jan/Feb 1997.

Memory

Built in	2MB Masked ROM and 1MB or 2MB RAM. Two SSD drives allow extra storage space on Flash/RAM SSDs, up to 8MB.

System information

Processor	NEC V30H running at 7.68MHz.
Operating system	EPOC. Microsoft MS-DOS compatible Flash Filing System.

Communications

Infrared:	IrDA SIR optical link, for IR communications and printing.
Protocols:	XMODEM, YMODEM and ZMODEM, (except from Comms Script in early models), giving compatibility with most computer communications software.
Language	Full script language with sample scripts allows automated log-on to electronic mail and other systems, and control of modems.

Expansion

Peripherals	External peripherals (such as a modems and printers) can be connected via the RS232 port which accepts optional PC Link and Parallel Printer Link cables. The PC Link allows communication with other computers. Travel Modem. PC Card adapter. SMS cables for Ericsson, Nokia and Orange mobile phones.
Socket:	15-way Honda type custom connector See the section 'The 15-way SIBO/RS232 connector' in the 'SIBO Expansion Ports' chapter in the *Hardware Reference* manual for pinout details.

Environment

Operating temperatures:	0ºC to 50ºC.
EMC:	FCC Part 15 Class B; CE Mark
Safety:	EN60950

Psion Siena Technical Specification

Physical characteristics

Part numbers:	1010-0003-01 (1MB) 1010-0002-01 (512KB)
Size	150mm x 70mm x 18mm (5.9"x 2.7"x 0.7").
Weight	180g (including batteries).
Screen	240 x 160 pixel high contrast retardation film LCD. Size 60.0mm x 40.0mm, (2.36" x 1.57"). Pixel pitch 0.25mm x 0.25mm Pixel size 0.23mm x 0.23mm.
Keyboard	48 key, QWERTY layout, computer style keyboard (UK models). 20 key calculator keypad 8 touch sensitive icon buttons for application selection.
Sound	Piezo buzzer.

Power supply

Internal	2 x AAA batteries giving approximately 40hrs use (2 months typical usage).
Backup	3V Lithium CR1620 battery.
External	From a Siena SSD Drive

Memory

Built in	1MB Masked ROM and 512KB or 1MB RAM.

System information

Processor	NEC V30H running at 7.68MHz.
Operating system	EPOC. Microsoft MS-DOS compatible Flash Filing System.

Communications

Infrared:	IrDA SIR optical link, for IR communications and printing. The Siena does not have the DYL file in ROM to support the AccessIr API (for Infrared beaming), this is provided with this SDK and must be loaded before third party programs using this API can be used. The Siena does not have the DYL file in ROM to support the IrLPT API (for Infrared printing), this must be loaded before third party programs using this API can be used.

Expansion

Peripherals	External peripherals (such as a modems and printers) can be connected via the RS232 port which accepts optional PC Link and Parallel Printer Link cables. The PC Link allows communication with other computers. External SSD Drive, via 1.536Mbits/sec Fast Serial interface. PC Card adapter. SMS cables for Ericsson, Nokia and Orange mobile phones.
Socket:	15-way Honda type custom connector See the 'Reduced External Expansion' section of the 'SIBO Expansion Ports' chapter in the *Hardware Reference* manual for pinout details.

Environment

Operating temperatures:	0ºC to 50ºC.
EMC:	FCC Part 15 Class B; CE Mark
Safety:	EN60950

Psion Siena SSD Drive Technical Specification

Physical characteristics

Part number:	1011-0005-01
Compatibility:	Psion Siena only
EMC:	FCC Part 15 Class B; CE Mark
Safety:	EN60950
Size	118mm x 69mm x 15mm (4.62"x 2.7"x 0.6"), (pre-production model).
Weight	87g, including flying lead, (pre-production model).
Power input:	A Psion Series 3c mains adaptor, (9-11V 250mA), (supplied with the drive).
Power output:	Power is supplied to a connected Siena via the RS232 connector.
Connectors:	15-way Honda type custom plug on flying lead to connect to the Siena 15-way Honda type custom socket to connect to a PC (for use with PsiWin)
SSD slots:	One SSD slot; accepts all capacities of Flash/RAM SSD.

Psion Serial 3Link Technical Specification

The Serial 3Link comes in two versions, one for IBM PC compatible computers and the other for Apple Macintosh machines.

Part number (PC):	1601-0001-01 (cable only) 1601-0039-11 (with PsiWin software)
Part number (Apple):	1601-0002-10 (with V1.41 software)
Compatibility:	Series 3, Series 3a (The Psion Series 3c and Psion Siena machines use a different type of communications cable, the PC Link cable.)
EMC	FCC Part 15 Class B; CE Mark

Safety:	EN60950
Physical	Pod with lead for connection to the Series 3 or LIF adaptor. Pod incorporates "auto wake up" switch. Replaceable lead to connect the pod to the other computer or peripheral. Complete connection is functionally equivalent to a 'null modem' cable.
Interface	RS232.
Memory	Masked ROM in the 3Link pod. This contains the script language and supporting communications software. The pod appears as SSD drive C.
Protocols	XMODEM and YMODEM, giving compatibility with most computer communications software.
Language	Full script language with sample scripts allows automated log-on to electronic mail and other systems, and control of modems.
Communications software	Link and/or RCom and/or PsiWin software supplied provides a simple interface for exchanging information with IBM PC compatibles and with the Apple Macintosh, giving direct remote file access.
Connectors - IBM PC	9-pin D-type (for connection to an IBM AT type PC or modem serial port). 25-pin D-type (for connection to an IBM XT type PC or modem serial port. **Note:** The model currently distributed through retail channels only has a 9-pin D-type connector, but the Serial 3Link assembly is available as separate cables and pod - see below).

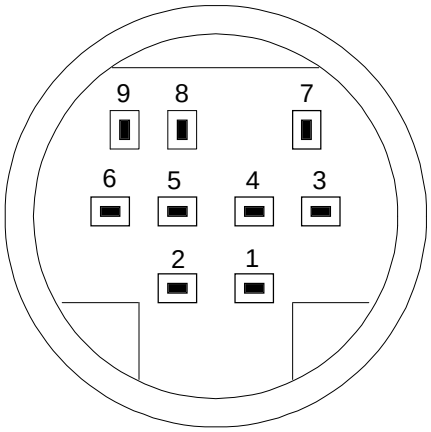
Pin name	Description	Pin number	Direction PC - 3Link
FG	Frame Ground (earth)	1	—
TD	Transmitted Data	2	→
RD	Received Data	3	←
RTS	Request To Send	4	→
CTS	Clear To Send	5	←
DSR	Data Set Ready	6	←
SG	Signal Ground (common return)	7	—
DTR	Data Terminal Ready	20	→

Connector - Apple Mac	8-pin round
Connectors - both versions	6-way two row rectangular (female) plug for connection to the Series 3 serial port, or Psion LIF adaptor (Work*about*). See the 'Reduced External Expansion' section of the 'SIBO Expansion Ports' chapter in the *Hardware Reference* manual for pinout details.

9-pin round (Internal connector on the 3Link pod)

Pin name	Description	Pin number	Direction Series 3 - Other
DCD	Data Carrier Detect	1	←
RD	Received Data	2	←
TD	Transmitted Data	3	→
DTR	Data terminal Ready	4	→
SG	Signal Ground (common return)	5	→
DSR	Data Set Ready	6	←
RTS	Request To Send	7	→
CTS	Clear To Send	8	←
RI	Ring Indicator	9	←

See below for diagram of pinout.



See the *Psion Series 3 3Link (RS232)* manual provided with the 3Link for full details of operation.

PC Serial 3Link assembly components

The PC Serial 3Link assembly is available as separate components with the following part numbers:

RS232 Cable 3Link Pod and cable to Psion	2500 0005 10
9 Pin Cable to PC	2303 0004 02
25 Pin Cable to PC	1404 0003
Double headed Cable to PC	2303 0016 02

PC Serial 3Link to Apple Macintosh converter

This converter changes a PC Serial 3Link into an Apple Macintosh Serial 3Link:

2 x disks, 1 x cable, 1 x manual	1601 0019 01

Modem Adaptor cable

This converter changes a Series 3a PC/Mac Serial 3Link cable so that the Psion can be connected to a Hayes compatible modem:

Series 3a Serial 3Link Modem Adaptor cable	1404 0002

Serial Printer cable (Series 3a) - Technical Specification

Part number:	1404 0001
Compatibility:	Psion Series 3/3s and Psion Series 3a only.
EMC:	FCC Part 15 Class B; CE Mark
Safety:	EN60950
Connectors:	6-way two row rectangular (female) plug for connection to the Series 3/3s/3a serial port. See the 'Reduced External Expansion' section of the 'SIBO Expansion Ports' chapter in the *Hardware Reference* manual for pinout details. RS-232 25 way D-type connector (male).

Serial Printer cable (Series 3c/Siena) - Technical Specification

Part number:	1602-0017-01
Compatibility:	Psion Series 3c and Psion Siena only.
EMC:	FCC Part 15 Class B; CE Mark
Safety:	EN60950
Connectors:	Low Profile Honda type custom connector RS-232 25 way D-type connector (male)

Psion PC Link cable Technical Specification

The Psion Series 3c and Psion Siena machines use this new type of communications cable.

Part number:	2013-0002-01 (cable only - available to developers by special request) 1011-0010-01 (with PsiWin software)
Compatibility:	Psion Series 3c and Psion Siena only. To connect to other devices with 25 way D-type connectors, adapters are needed, (see Psion 9-to-25 way D-type adapters Technical Specification)
EMC:	FCC Part 15 Class B; CE Mark
Safety:	EN60950
Connectors:	Low Profile Honda type custom connector RS-232 9 way D-type connector (IBM AT type)

Modem Adaptor cable

This converter changes a Series 3c/Siena PC/Mac Link cable so that the Psion can be connected to a Hayes compatible modem:

Series 3c Serial Link Modem Adaptor cable 1602 0016 01

Psion Mac Link cable Technical Specification

The Psion Series 3c and Psion Siena machines use this new type of communications cable.

Part number:	1601-0101-01
Compatibility:	Psion Series 3c and Psion Siena only.
EMC:	FCC Part 15 Class B; CE Mark
Safety:	EN60950
Connectors:	Low Profile Honda type custom connector RS-232 (Apple connector)

Psion 9-to-25 way D-type adapters - Technical Specification

The Psion series 3c and Psion Siena machines use a different type of communications cable from preceding machines;- the Psion PC Link cable. To connect to other 25 way D-type connector devices, adapters are needed as, unlike the 3Link cable, this is a single cable and can't be split to change the RS-232 half to connect to a serial printer or modem.

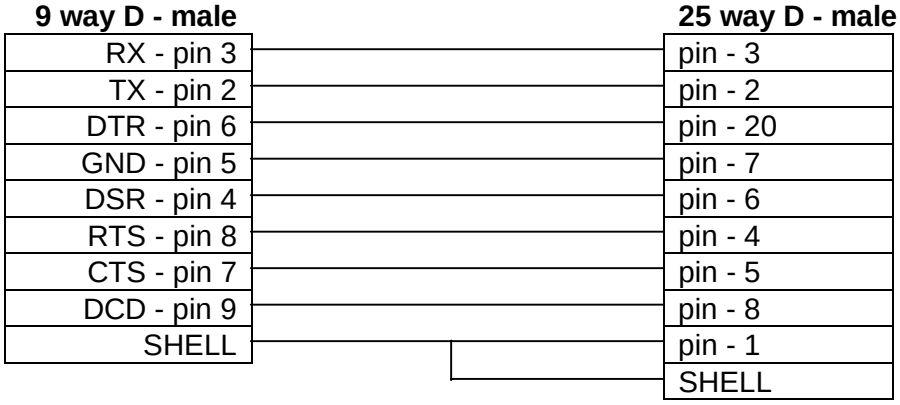
The new PC Link and 3Link cables use the pins on the D-type plugs in a unique way. This means that a unique mapping from 9-to-25 way pinouts is required, so special adapters (i.e. non-industry standard) have been designed to provide this unique mapping.

Except in special cases, these adapters replace the need for the old RS-232 serial printer and modem cables for 3Link users. In addition, these adapters allow connection of a 3Link or PC Link cable to a PC with 25 way communication ports.

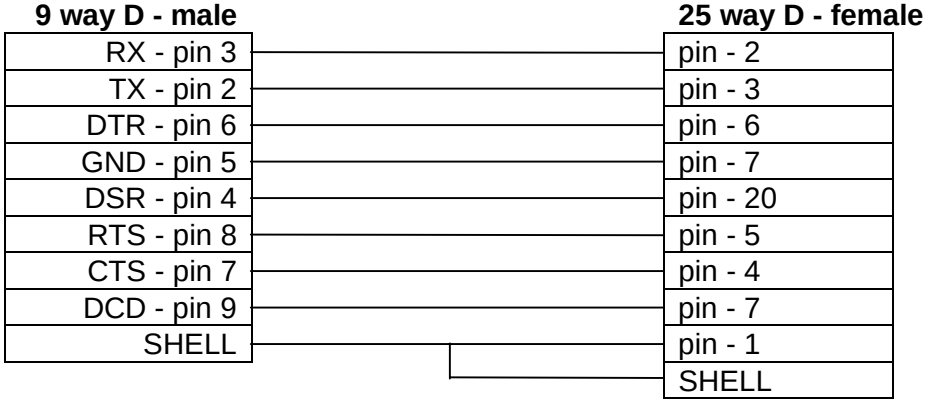
The three types of Psion branded RS232 9-to-25 way D-type adapters allow any Psion cable with a 9 way D-type connector to be connected to another machine with a IBM XT type PC 25 way D-type port or modem or printer serial port.

Adapter	Part number	EMC	Safety
Modem	1602 0016 01	FCC Part 15 Class B; CE Mark	EN60950
Serial Printer	1602 0017 01	FCC Part 15 Class B; CE Mark	EN60950
PC (XT)	1602 0015 01	FCC Part 15 Class B; CE Mark	EN60950

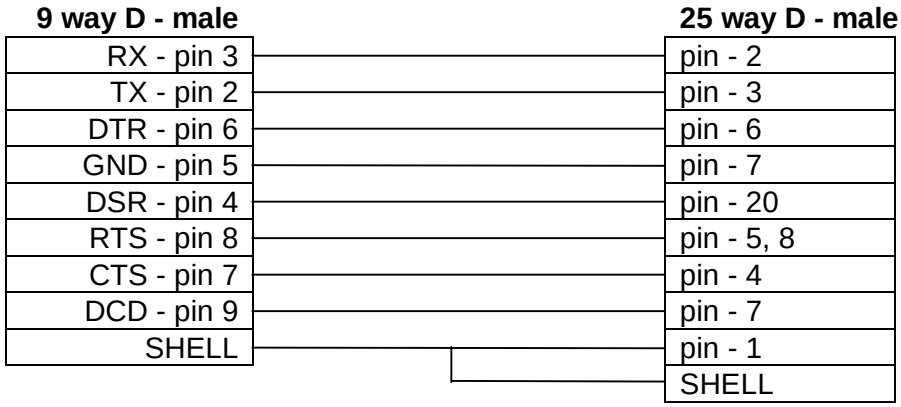
Modem 9-to-25 way D-type adapter - wiring diagram



PC (XT) 9-to-25 way D-type adapter - wiring diagram



Printer 9-to-25 way D-type adapter - wiring diagram



Psion Parallel 3Link Technical Specification

Part number:	1601-0003-10
Compatibility:	Psion Series 3, Psion Series 3 and Psion Work*about* with LIF adaptor only (The Psion Series 3c and Psion Siena machines use a different type of printer cable, the Parallel Printer Link cable.)
EMC:	FCC Part 15 Class B; CE Mark
Interface:	Centronics interface for parallel printers
Connectors:	6-way two row rectangular (female) custom plug, for connection to the Series 3 or Series 3a serial port, or Psion LIF adaptor (Work*about*). See the 'Reduced External Expansion' section of the 'SIBO Expansion Ports' chapter in the *Hardware Reference* manual for pinout details.
	Centronics (for connection to the parallel port of a printer)

Psion Parallel Printer Link cable - Technical Specification

Part number:	1011-0017-01
Compatibility:	Psion Series 3c and Psion Siena only The Parallel Printer Link is not for use with the Series 3, Series 3a, MC, HC or Work*about*, (The Psion Series 3 and Psion Series 3a machines use a different type of printer cable, the Psion Parallel 3Link cable.)
EMC:	FCC Part 15 Class B; CE Mark
Safety:	EN60950
Connectors:	Low Profile Honda type custom connector Centronics connector (to connect to a Centronics type parallel printer port)
Power:	One 9V PP3 battery. With typical usage, the expected life from a new battery is 150 hours. Power is only consumed when printing.

Psion PC Card Modem Adapter - Technical Specification

Variants

There are three variants:

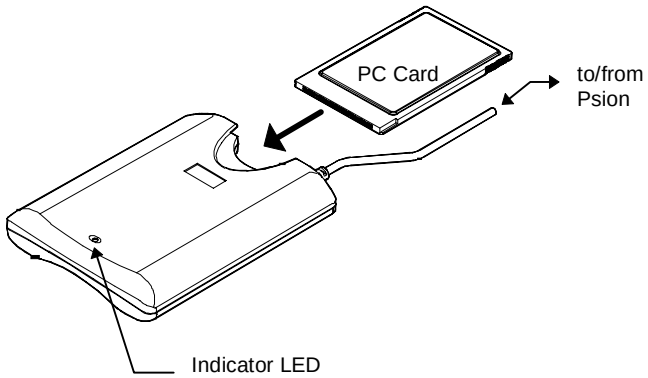
- Series 3a
- Series 3c
- PC

The PC Card Modem Adapter with a suitable PC Card modem, combined with the PsiFax software, supersedes the 3Fax modem and software for the Series 3a. For the Series 3c there is also the alternative Travel Modem.

Physical characteristics

Part number:	Series 3a variant: 2501-0223-01 (unboxed, not retail) Series 3c variant: 2501-0224-01 (unboxed, not retail) PC variant: 2501-0225-01 (unboxed, not retail)
Compatibility:	Psion Series 3a - Series 3a variant only Psion HC - Series 3a variant only, with LIF connector module & LIF adapter Psion Work*about* - Series 3a variant only, with LIF adapter Psion Series 3c - Series 3c variant only PC - PC variant only The PC Card Modem Adapter will work with Psion's corporate and personal email applications and standard communications applications. Current versions of 3Fax software will not work with the PC Card Modem Adapter. New versions of 3Fax software, renamed as PsiFax, are available to work with the adapter.
EMC:	EN55022, EN500082-1
Safety:	EN60950
Size	118.5mm x 91mm x 20.5 mm
Weight	105g without batteries
Processor:	Siemens C165 16 bit micro-controller
Memory:	OTP.
Power:	4 x AA batteries and/or 6V DC/1A mains adapter
Battery life:	Depends on the modem card you use. With a Psion Dacom 28.8 modem, typically 90 to 120 mins (Duracell batteries). If the unit is unused, approximately 6 months (Duracell batteries).
Interface to computer:	Series 3a variant: SIBO Fast Serial (ASIC5) Series 3c variant: RS232 PC variant: RS232
User interface:	A single LED and piezo buzzer. The LED flashes according to the status of the PC Card adapter (transmitting, low power warnings etc.). The piezo buzzer provides modem tones, dial tones and bleeps with recognition when a PC Card modem is inserted into the adapter.
Connectors:	For connection to the computer: Series 3a variant: 6-way two row rectangular (female) custom plug Series 3c variant: Low Profile Honda type custom connector PC variant: 9-way D type connector For connection to the PC Card modem: PCMCIA standard connector

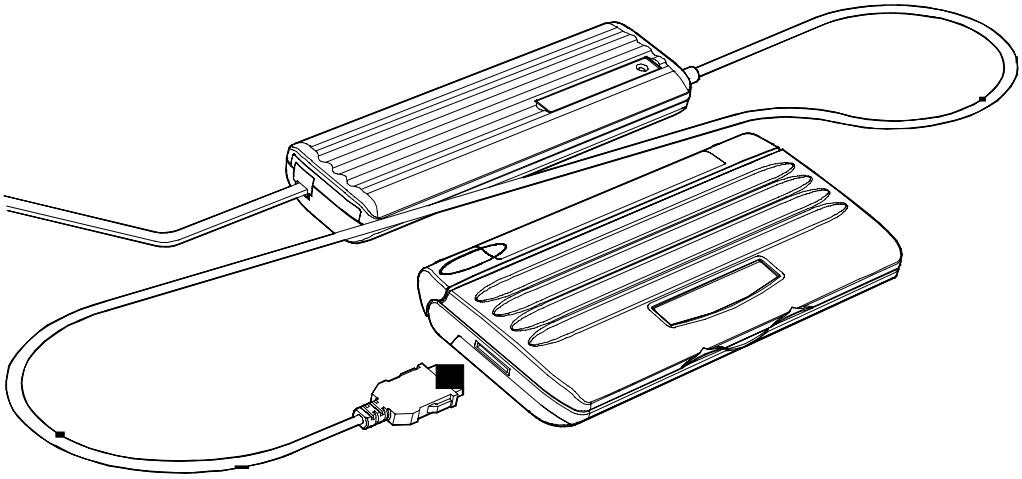
Supported PC Cards:	Psion Dacom Gold Card US Robotics WorldPort Pace Microlin Megahertz X JACK Series Nokia Cellular Data Card Philips Mobile Data Card Hayes PC Cards Dr. Neuhaus Fury Cards and many other popular models PC Modem Cards can be inserted at any time and recognised by the card adapter (hot swapping)
DTE formats	8N1,7E1,7O1
Flow control	RTS/CTS
Data transfer rates:	300 Baud to 57.6 KBaud with autobauding 300 Baud to 115 KBaud without autobauding Note: Because Psion Fast Serial operates at a maximum 19.2kbs the user will not be able to get the full 33.6 KBaud data rate available from V.34 PC Cards in the Series 3a variant of the PC Card Adpter. In this scenario the connection rate will drop down to 14.4 KBaud.



Indicator LED

Status	On Batteries	On Mains
Stand by	Off	Flashing green (slow)
Idle	Flashing green (slow)	Flashing green (slow)
Configuring PC card	Flashing green (fast)	Flashing green (fast)
Connected to remote modem	Continuous green	Continuous green
Low battery warning	Flashing red	N/A

Psion Travel Modem Technical Specification



Physical

Part number:	unknown at time of publication (available January 1997)
Series 3/3s compatibility:	No
Series 3a compatibility:	No
Work*about* compatibility:	No
Series 3c compatibility:	Yes
Siena compatibility:	No
EMC:	FCC Part 15 Class B; CE Mark
Environment:	Operating temperature: 0-50C Operating humidity: 0-95% non-condensing
Dimensions:	165mm x 40mm x 25mm
Power:	2 x AA batteries **or** optional rechargeable NiCad battery pack **or** optional 10V (250mA) DC Series 3c mains adaptor.

Communications

Functions:	Autodialling (tone and pulse) modem conforming to: V.21,V.22, V.23, V.22bis, V.27ter, V.29, V 32 and V 32bis standards. It supports V.25 auto answering recommendations.
Dialling:	May be used with either tone (MF) or pulse (LD) signalling BT lines.
Echo suppressor:	Echo-suppressor tone (V25) when auto answering.
REN:	1.
Fax operation:	Controlled automatically by software (not supplied with the Travel Modem). The Travel Modem can send faxes at up to 9600bps. The fax feature is compatible with Group 3 fax machines.

Computer connection

Connector:	Low Profile Honda type custom connector on flying lead for connection to a Psion Series 3c. See the 'Reduced External Expansion' section of the 'SIBO Expansion Ports' chapter in the *Hardware Reference* manual for pinout details.

Network connection

Data transfer rate:	up to 57600bps with V.32bis and V.42bis (compression is dependant upon file type) V.32 9600 bps full duplex V.32bis 14400 bps full duplex

Operational modes:	V.21 300bps full duplex V.22 1200bps full duplex V.22bis 2400bps full duplex V.23 1200/75 full duplex 75/1200 full duplex V.27ter 4800bps fax send and receive V.29 9600bps fax send and receive
Line connection:	2 wire PSTN via BT 600 type modular jack on flying lead
Signal level:	-9 dBm
Equalisation:	Transmit fixed compromise Receive automatic adaptive
Interface:	600 ohm
Error correction:	V.42 incorporating LAPM and MNP Class 4 MNP Class 10
Data compression:	V.42bis and MNP Class 5

Autodial/autoanswer

Dial method:	Tone or pulse, selectable
Call progress:	Loudspeaker with on/off control Extended results codes
Call control:	Extended "AT" command set
Automatic answer:	To CCITT V.25 recommendation
Mode selection:	Automatic configuration to V.21/V.22/V.23/V.22bis and V32/V32bis on receive
Call disconnection:	Loss of carrier, DTR or by command

BABT Approval

Approval number:	606866

Psion 3Fax Modem Technical Specification

The 3Fax modem has been superseded by the Series 3a PC Card Adapter with a suitable PC Card modem, combined with the PsiFax software on SSD, (for the Series 3c there is the Travel Modem).

Physical

Part number:	1601-0022-01
Series 3/3s compatibility:	No
Series 3a compatibility:	Yes (512KB, 1MB and 2MB models only)
Work*about* compatibility:	No
Series 3c compatibility:	No
Siena compatibility:	No
EMC:	FCC Part 15 Class B; CE Mark
Environment:	Operating temperature: 0-50C Operating humidity: 0-95% non-condensing
Dimensions:	165mm x 40mm x 25mm
Power:	2 x AA batteries **or** optional rechargeable NiCad battery pack **or** optional 10V (250mA) DC Series 3a mains adaptor.

Communications

Functions:	Autodialling (tone and pulse) modem conforming to: V.21,V.22, V.23, V.22bis, V.27ter, and V.29 standards. It supports V.25 auto answering recommendations.
Dialling:	May be used with either tone (MF) or pulse (LD) signalling BT lines.
Echo suppressor:	Echo-suppressor tone (V25) when auto answering.
REN:	1.
Fax operation:	Controlled automatically by the software supplied with the 3Fax modem. The 3Fax modem can send faxes at up to 9600bps. The fax feature is compatible with Group 3 fax machines.

Computer connection

Connector:	6-way two row rectangular (female) plug for connection to a Psion Series 3a. See the 'Reduced External Expansion' section of the 'SIBO Expansion Ports' chapter in the *Hardware Reference* manual for pinout details.

Network connection

Data transfer rate:	up to 9600bps with V.22bis and V.42bis (compression is dependant upon file type)
Operational modes:	V.22bis 2400 bps full duplex V.22 1200 bps full duplex V.23 1200/75 full duplex 75/1200 full duplex V.21 300 bps full duplex V.27 ter 4800 bps fax send and receive V.29 9600 bps fax send and receive
Line connection:	2 wire PSTN via BT 600 type modular jack 3 wire Bell Tinkle Suppression supported
Signal level:	-9 dBm
Equalisation:	Transmit fixed compromise Receive automatic adaptive
Interface:	600 ohm
Error correction:	V.42 incorporating LAPM and MNP Class 4
Data compression:	V.42bis and MNP Class 5

Autodial/autoanswer

Dial method:	Tone or pulse, selectable
Call progress:	Loudspeaker with volume control Extended results codes
Call control:	Extended "AT" command set
Automatic answer:	To CCITT V.25 recommendation
Mode selection:	Automatic configuration to V.21/V.22/V.23 & V.22bis on receive
Call disconnection:	Loss of carrier, DTR or by command

BABT Approval

Approval number:	NS/1397/3/R/604375

The greyed out table is statutory statements and not really part of the tech spec

Nokia 21XX to Series 3c/Siena SMS Cable - Technical specification

This SMS cable is available from the beginning of January 1997.

This SMS Link cable connects a Psion Series 3c or Siena to Nokia’s 21XX range of GSM/DCS/PCS digital mobile phones.

The cable is for use with applications based on version 2.01 of the SMS SDK running on a Siena or Series 3c.

The top level Psion part number for this SMS cable is 1601- 0106 -01.

Ericsson SMS SDK and SMS Cables

There will be 2 types of SMS products:

- ***SMS SDK*** **-** enables Psion Registered Developers to create applications for both vertical and horizontal markets that make use of the SMS capability provided by Ericsson's GSM/DCS/PCS digital mobile phones;
- ***SMS applications and SMS cables*** **-** Shrink wrapped and custom SMS applications available from both Psion and its Registered Developers. These applications will be invariably be packaged with the Ericsson SMS cable to connect the Psion to the Ericsson mobile phone.

The SMS SDK will be released to selected registered developers from the end of December 1996.

The SMS SDK is distributed for free.

Product Description

Ericsson SMS SDK

The SMS SDK provides the following capabilities:

- Mobile Originate (MO) and Mobile Terminate (MT) of SMS messages;
- Access to the address book memory within the phone and SIM;
- Sensing of key presses and screen indications on the phone;
- Ability to send the full range of key presses from the Psion to the phone;
- Sensing the model number and firmware version of the phone;
- SMS delivery status reports (DSRs).

SMS cables

An SMS Link cable connects a Psion Series 3c or Siena to Ericsson's GSM/DCS/PCS range of digital mobile phones (GH377, GF388, GA318, PH337, CH337 etc.)

Note: Ericsson SMS links will not be available for the Psion 3a.

The interface between the Ericsson handset and Psion 3c/Siena is housed within the Ericsson connector, thus providing the user with only a thin cable connection.

The top level Psion part number for this SMS cable is 1601-0107-01.

Cellnet SMS*link*

Cellnet SMSlink is a joint development between Cellnet and Psion. It is an SMS product for the popular Series 3a and Nokia 2110 combination.

The Product includes:

- Cable to connect Series 3a to Nokia 2110/2110i, and Philips PR747 GSM mobile phones
- Series 3a software for composing, sending and receiving text messages
- PC software (on floppy disc) to send text messages direct from PC to GSM mobile phones. (Requires a modem) Note: This is Freeware supplied by Cellnet.
- User Guide

Cellnet SMSlink allows you to:

- Compose, edit, send and receive short messages of up to 160 characters
- Simple message management using the inbox and outbox, and unread message box
- Send and receive electronic business cards
- Send messages to a selected group of contacts
- Incorporates a contact manager similar to the built-in Data application
- Works with contact databases generated using Data or compatible applications
- Sort and search on contact database entries
- 10 pre-set messages, and you can also create your own.
- Appointment details sent in a message may be extracted and merged with Agenda.

Compatibility

Compatible with Psion Series 3a computers - 512k and above

This application will be distributed on SSD and floppy disk. Floppy disc version requires PsiWin or 3link for installation.

Version	**Top level part number**
SSD:	1601-0065-01
Floppy disc:	1601-0066-01

Availability

Has been available from August 1996.

Please Note: Our license with the development company only allows us to market this product in the UK. Therefore it is not available to export markets.

Vodaphone Telenote Link

Vodaphone Telenote Link is a joint development between Vodaphone, Nokia and Psion. It is an SMS product for the popular Series 3a and Nokia 2110 combination.

Versions

There are two versions of the Telenote link:

- Version 1, (available from the beginning of December 95), is for use with the Nokia 2110 and Philips PR747 digital handsets;
- Version 2, (available from February 96), is for use on the Orbitel 905 digital handset.

The product is issued either on SSD and floppy disk.

System Requirements

Telenote Link requires a Psion Series 3a 256K or larger internal RAM palmtop computer, or a Psion Work*about*, and a Nokia 2110 (or equivalent) digital phone on the Vodafone digital mobile phone network.

The Telenote Link product includes software for a PC which allows its user to originate messages, and using a modem, (not supplied) to dial direct into the Vodafone Short Message Centre.

Telenote Link Functions

The Vodafone Telenote Link for the Psion Series 3a allows the user you to create, store and receive short messages in a simple intuitive manner using his Series 3a and Nokia GSM phone. The software for the Series 3a also includes a comprehensive address book function and the ability to create frequently used messages (e.g. “Please call me”, or “I will be 15 minutes late for the meeting”), which can be sent quickly with a minimum number of key strokes.

The Telenote Link product allows the user to:

- Send messages from his Series 3a and Nokia phone to another handset on the Vodafone GSM network, or any other GSM network world-wide which supports SMS and with which Vodafone have a roaming agreement;
- Send messages from a PC (provided the user already has a modem) to any GSM handset on the Vodafone network (this feature will work with handsets other than those based on the Nokia 2110).

The Telenote Link product does not allow the user to:

- Send or receive SMS to/from a Cellnet, Orange or Mercury digital phone;
- Send or receive SMS to/from a phone on a network which does not support SMS, or with which Vodafone does not have a roaming agreement. Most foreign GSM networks will support it, but some (particularly France) are behind in deploying the service;
- Send or receive e-mail to/from his Series 3a/Nokia;
- Send or receive Faxes to/from his Series 3a/Nokia.

Telenote Link Package

The Telenote Link Package includes:

- Cable to connect a Series 3a to Nokia 2110 (or derivatives such as Philips) GSM phones on the Vodafone network;
- Software for the Series 3a supplied on floppy disk or SSD. Floppy disk software (requires a PC and PsiWin to download to Series 3a);
- Software for a PC allowing use of a modem (not supplied) to direct dial into the Vodafone Short Message Centre, allowing the user to originate messages from his desktop. This is particularly useful for a receptionist/secretary who needs to pass on messages to somebody who is away from the office;
- Comprehensive user guide.

Version	Top level part number
SSD:	1601-0057-01
Floppy disc:	1601-0056-01

Availability

Current product.

Orange Messaging Link

Orange Messaging Link is a joint development between Orange and Psion. It is a messaging product which is based on the GSM SMS service but for the Orange network. (Orange is a Personal Communications Network (PCN) which uses GSM technology but at a higher frequency and with lower power handsets) It works with Series 3a and Nokia Orange and Nokia Orange 5.1 handsets.

The product includes:

- Cable to connect Series 3a to Nokia Orange and Nokia 5.1 handsets
- Series 3a software for composing, sending and receiving text messages
- User Guide
- A guide prepared by Orange on how to access their network from a PC with a modem to send messages.

Orange Messaging Link allows you to:

- Compose, edit, send and receive short messages of up to 160 characters
- Simple message management using the ***Sent***, ***Received***, ***Not sent***, and ***Phonebook*** dialogues
- Send messages to a selected group of contacts
- Works with contact databases generated using the series 3a Data application.
- Create your own preset messages

Compatibility

Compatible with Psion Series 3a computers - 512k and above

This application software is distributed on SSD or floppy disk. The floppy disc version requires PsiWin or 3Link for installation.

Version	Top level part number:
SSD:	1601-0067-01
Floppy disc:	1601-0068-01

Availability

Available from mid-November 1996.

Please Note: Our license with the development company only allows us to market this product in the UK. Therefore it is not available to export markets.

APPENDIX B

DIFFERENCES BETWEEN PSION SERIES 3 MODELS

At the time of writing there are four current and six superseded machines in the Series 3 family. The main differences between the computers are tabulated overleaf.

Current models in the Series 3 family

Feature \ Model	Series 3c	Series 3c	Siena	Siena
Internal RAM size	2MB	1MB	1MB	512KB
Internal masked ROM size	2MB	2MB	1MB	1MB
Screen (pixels)	480x160	480x160	240x160	240x160
Screen (characters)	80x17	80x17	40x17	40x17
Grey scale supported	3	3	3	3
SSD drives	2	2	optional extra	optional extra
Communications connector	R232 low profile, with power output	R232 low profile, with power output	R232 low profile, with power input	R232 low profile, with power input
Time management/organiser				
- Open book diary with reminder alarms	day/week	day/week	day/week	day/week
- Planner views	year/busy	year/busy	busy	busy
- 'Things to do' manager	multiple lists with due dates and alarms	multiple lists with due dates and alarms	multiple lists with due dates and alarms	multiple lists with due dates and alarms
- Chronological list and anniversaries view	3	3	3	3
Voice/sound recording and playback	“Sound” app	“Sound” app		
Spreadsheet	3	3	3	3
Database list view	3	3	3	3
Calculator	desk, advanced	desk, advanced	desk, advanced	desk, advanced
Jotter	3	3		
File Manager	3	3		
Print preview	3	3	3	3
Fax facility (send only)	optional extra	optional extra		
Infrred communications	3	3	3	3
Zoom-in zoom-out through four font sizes	3	3	3	3
Rich text file (RTF) support	3	3	3	3
Spellchecker and Thesaurus	3	3	optional extra	optional extra
Patience card game	3	3		
Extended OPL language	3	3	3	3
Startup tips	3	3	3	3

Superseded models in the Series 3 family

Feature \ Model	Series 3a	Series 3a	Series 3a	Series 3a	Series 3s	Series 3	Series 3
Internal RAM size	2MB	1MB	512KB	256KB	256KB	256KB	128KB@
Internal masked ROM size (* 2MB if Spell app included)	2MB/1MB*	2MB/1MB*	1MB	1MB	512KB	384KB or 512KB	384KB or 512KB
Screen (pixels)	480x160	480x160	480x160	480x160	240x80	240x80	240x80
Screen (characters)	80x17	80x17	80x17	80x17	40x9	40x9	40x9
Grey scale supported	3	3	3	3			
SSD drives	2	2	2	2	2	2	2
Communications connector	SIBO	SIBO	SIBO	SIBO	SIBO	SIBO	SIBO
Time management/organiser							
- Open book diary with reminder alarms	day/week	day/week	day/week	day/week	day	day	day
- Planner views	year	year	year	year			
- 'Things to do' manager	multiple lists with due dates and alarms	multiple lists with due dates and alarms	multiple lists with due dates and alarms	multiple lists with due dates and alarms	single list, up to 50 items	single list, up to 50 items	single list, up to 50 items
- Chronological list and anniversaries view	3	3	3	3			
Voice/sound recording and playback	"Record" app	"Record" app	"Record" app	"Record" app			
Spreadsheet	3	3	3	3	3		
Database list view							
Calculator	advanced	advanced	advanced	advanced	advanced	advanced	advanced
Jotter							
File Manager	optional extra	optional extra	optional extra	optional extra			
Print preview	3	3	3	3			
Fax facility (send only)	optional extra	optional extra	optional extra				
Infrared communications							
Communications application	3	3	3	3	optional extra	optional extra	optional extra
Zoom-in zoom-out through four font sizes	3	3	3	3			
Rich text file (RTF) support	3	3	optional extra	optional extra	optional extra	optional extra	optional extra
Spellchecker and Thesaurus	3	3	optional extra	optional extra	optional extra		
Patience card game	3	3	optional extra	optional extra			
Extended OPL language	3	3	3	3			
Startup tips							

INDEX