

SIBO 'C' Software Development Kit

THE SIBO DEBUGGER

Version 2.10

February 3, 1995

(C) Copyright Psion PLC 1990-95

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

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

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

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

Contents

1 Introduction	1
Debugging modes	1
SIBO architecture overview.....	2
System memory	2
Segment register usage and interrupts	2
2 Starting up the Debugger	5
Preparing a process for debugging	5
JPI TopSpeed C symbol files	5
The debugger and symbol information	6
Debugger initialisation.....	6
Debugger command line.....	6
Configuration files.....	7
Remote Debugging.....	8
Local debugging	9
Simultaneous local and remote debugging	9
Initial display	9
3 The Graphics Interface	11
Tasks	11
The titled window	11
Resize	11
Move	11
Zoom	12
Iconising a window	12
The command menu bar	12
Accelerators.....	12
Dialog boxes.....	13
Buttons	13
Pop-out menus	13
Tick boxes and diamonds.....	13
The file selector dialog	13
Help	14
4 Top-Level Views	15
Process list top-level view	15
The current target.....	15
Debugger menu	16
Next top-level view	16
Version	16
Create File Manager	16
Run image	16
Exit.....	16
Local/Remote CPU menu	16
Connect to Remote/Local.....	16
Information.....	16
Process menu.....	17

Break into	17
Status	18
Check data segment	18
View menu	18
Process list	18
File.....	18
Segments.....	18
Devices	18
Environment variables	18
Regenerate list.....	18
File top-level view	18
View menu	18
Delete.....	18
Locate menu.....	19
Go to line.....	19
Search.....	19
Search again	19
Segment top-level view	19
Devices top-level view	19
Environment variable top-level view.....	19
Variable menu	20
Modify.....	20
Set format.....	20
Clear	20
Follow	20
Previous	20
Search.....	20
Search again	20

5 The Process Window	21
Tracking views	21
Panics	22
The code view.....	22
Assembly language INT instructions	22
Process menu	22
Next top-level view.....	22
Status	22
Reload	23
Unload.....	23
Exit	23
View menu.....	23
Source module.....	23
Code segment	23
Data segment	23
Registers.....	23
Stack	23
Symbols	23
Magic statics	24
Variable	24
File.....	24
Delete.....	24
Run menu	24
Step	24
Trace.....	24
Run.....	25
Run to here	25
Break menu	25
Breakpoints	25
Break here.....	26
Data menu	26
Set display mode	26
Check data segment	26

AL as error code.....	26
AL as panic code.....	26
Locate menu	26
Follow.....	26
Previous.....	27
Goto address	27
Goto line	27
Search	27
Search again.....	27
Breakpoints in dynamic libraries and other shared code	27
The data view.....	28
Locate menu	28
Follow.....	28
Previous.....	28
Goto address	29
Data menu.....	29
Set data format	29
Modify	29
The registers view.....	29
Data menu.....	30
Set radix	30
Modify	30
The stack view	30
Operating system stack frames.....	30
Locating the origin of a panic.....	31
Locate menu	31
Follow.....	31
Previous.....	31
Goto address	31
Data menu.....	31
Modify	31
The symbols view	32
The variable view	32
Display formats for variables	32
Basic data types	33
Pointers.....	33
Structures and unions	33
Arrays.....	33
Enums	34
Bitfields.....	34
Data menu.....	34
Modify	34
The magic statics view	34
The file view.....	34

6 The File Manager	35
Moving around the file manager	35
Operations on files	36
Selecting files.....	36
Tagging files	36
Copying files	36
Renaming files.....	37
Deleting files	37
Attributes	37
Changing the order of a directory listing.....	37
Operations on directories.....	38
Creating a directory.....	38
Removing directories	38

Operations on devices	38
-----------------------------	----

7 Troubleshooting	39
--------------------------------	-----------

No source code displayed	39
--------------------------------	----

Communications link broken.....	40
---------------------------------	----

Poor, distorted or missing display.....	40
---	----

CHAPTER 1

INTRODUCTION

The Psion SIBO Debugger is tailored to the EPOC operating system environment and enables you to debug applications written for the Psion SIBO family of machines. This family currently includes the MC200, MC400, HC and Series 3 ranges.

It is a source level debugger, using the symbol table information produced by the JPI TopSpeed compiler. Currently only the C programming language is supported.

The debugger is supplied with a built-in file manager, described in a later chapter of this manual, which provides file, directory and device manipulation.

The debugger runs on a host machine and can be used to debug Psion SIBO applications running on either the local host or, via its remote debugging facility, on a remote SIBO machine. The debugger can simultaneously debug up to eight processes - four on each of the local and remote machines.

To debug a remote application you will need a development PC with a serial port (and, optionally, a Bus mouse). For remote debugging the PC should be connected via a serial cable to a remote SIBO machine which may be:

- an MC200 or MC400 (version 2.30 or above)
- an HC (any version)
- a Series 3 (version 1.77 or above) with a serial link expansion module.
- a Series 3a (version 3.20 or above) with a serial link expansion module.

Versions earlier than 2.30 of the MC or 1.77 of the S3 will cause the debugger to terminate.

Debugging modes

You may use the debugger in one of three possible ways:

- as a 'conventional' debugger
- to bring a running process under the debugger's control
- to locate a panic in a running process

The 'conventional' use is to load a process from within the debugger, set breakpoints, step, trace and run, as with any other debugger.

The second type of use is of value when debugging a process that can not easily be loaded from the debugger. An example would be to debug a replacement shell on the HC, since it will be automatically loaded and run on system start-up. This technique is described in the *Process list top-level view* section of the chapter *Top-Level Views*.

In order to locate a panic you simply run the debugger and then independently run and exercise the process under test. When the process panics it is automatically brought under the debugger's control. The steps needed to locate the application code which gave rise to the panic are described in the *Stack view* section of the *Process Window* chapter.

SIBO architecture overview

This section gives a brief overview of the relevant aspects of the SIBO programming environment. It should be read in conjunction with the *Memory Allocation* chapter in the *PLIB Reference* manual.

System memory

The EPOC operating system manages all the memory within a machine. The sections of memory that the debugger is primarily concerned with are the allocated memory segments - contiguous regions of memory that contain 'live' information, either code or data.

The EPOC operating system maintains within its data space an allocated memory segment table. This table has room for 96 entries, each of which contains:

- a physical 8086 segment register address of the start of the segment
- an access count
- a unique segment name

The position of such an entry within the table is known as the segment handle for the relevant allocated memory segment.

Memory segments are dynamic in size; they may grow or shrink depending on the amount of memory actually being used within the segment. Although a memory segment that contains code will not, in general, change size, one containing data, particularly an application process data segment, is quite likely to change size. The debugger takes account of this and any views on data segments are resized appropriately.

Each memory segment has an access count that indicates how many times the segment has been 'opened'. Only when the access count is reduced to zero will the memory segment be freed. This mechanism allows code sharing, where multiple processes of the same application share a single segment containing the application code. The debugger understands this principle and breakpoints are associated with a particular process, rather than with the code segment itself.

As part of its memory management system, EPOC may move allocated memory segments. This ensures that, as memory segments are allocated, freed or changed in size, the pool of free system memory exists as a single contiguous region. The physical address of an allocated memory segment may therefore change over time, but the segment handle within the segment table will always remain constant.

The debugger automatically tracks the movement of memory segments. It does not display the segment registers or the absolute segment address since these values do not have much meaning; they may change at any time. The debugger handles segments symbolically by the name of the segment, but places no significance on the segment name. It can not, for example, determine the nature of a segment's contents from its name.

Segment register usage and interrupts

Although memory segments move, the majority of programmers need not concern themselves with this. Only machine code programmers who want to manipulate the 8086 segment registers need read the remainder of this section.

Many EPOC system services, including memory segment movement, are performed under interrupt control.

If a segment register is used to point at or within a memory segment the operating system will modify the segment register correctly when memory moves. If a segment register is to be modified the programmer should ensure that interrupts are disabled during the modification. Interrupts should be enabled as soon as the segment register content has been modified.

Conversely, if a segment register is to be used as a scratch register then interrupts should remain off for the duration of such usage, since the operating system will modify all segment registers when it moves memory.

If an application calls an operating system service that causes the process to wait on a semaphore, the DS and ES segment registers must contain the segment address of the calling process data segment. Such services are `p_read`, `p_write`, `p_seek`, `p_close`, `p_iow` and `p_msendreceive`.

An application should, if possible, avoid disabling interrupts. If it is necessary to disable interrupts, they should be disabled for as short a time as possible. Leaving interrupts disabled for more than 1 millisecond will, at the very least, cause significant degradation to system performance.

If the application leaves interrupts disabled for more than about one second, a watchdog NMI (non maskable interrupt) will occur and the operating system will terminate the process that has interrupts disabled.

1. The first part of the document is a letter from the President of the United States to the Congress, dated January 1, 1801. It is a very important document, as it is the first time that the President has addressed the Congress since the establishment of the office.

2. The second part of the document is a report from the Secretary of the Treasury, dated January 1, 1801. It is a very important document, as it is the first time that the Secretary of the Treasury has reported to the Congress since the establishment of the office.

3. The third part of the document is a report from the Secretary of the Navy, dated January 1, 1801. It is a very important document, as it is the first time that the Secretary of the Navy has reported to the Congress since the establishment of the office.

4. The fourth part of the document is a report from the Secretary of the War, dated January 1, 1801. It is a very important document, as it is the first time that the Secretary of the War has reported to the Congress since the establishment of the office.

5. The fifth part of the document is a report from the Secretary of the Interior, dated January 1, 1801. It is a very important document, as it is the first time that the Secretary of the Interior has reported to the Congress since the establishment of the office.

CHAPTER 2

STARTING UP THE DEBUGGER

Preparing a process for debugging

In order for the debugger to provide source level debugging you must build the application in such a way that the appropriate symbol files are generated.

The debugger looks for a *.map* file, a *.sym* file and a number of *.dbd* files. A *.dbd* file is created by the JPI compiler during the compilation of a source module and has the same file name as the source module file.

The *.map* file is generated while linking the application and has the same name as the *.img* file. It is used, primarily, to obtain symbolic information for library routines.

The *.sym* file is created by the EMMAKE utility program (provided there is symbolic information to write out) at the same time as it creates the *.img* file. It has the same file name as the *.img* file and contains all the information required to load the *.dbd* files.

Each source module linked to produce the *.img* file requires a *.dbd* file to describe its contents for symbolic debugging. The debugger does not require a *.dbd* file for every module (or, in fact, for any module) but it will not be able to present source level debugging for any module that does not have a corresponding *.dbd* file.

JPI TopSpeed C symbol files

To allow the compiler to generate source level symbolic information (*.dbd* files) the VID debug pragma should be set to either min or full. This can be done either within the JPI project system or within the *.pr* files. A *.pr* file should, for example, contain the line:

```
#pragma debug(vid=>full)
```

or

```
#pragma debug(vid=>min)
```

There is further information on this topic in the *Building an Application* chapter of the *General Programming* manual. It should be noted that the JPI compiler generates different code for each level of the VID pragma. The more debugging information generated, the more actual executable code is produced. This has an unfortunate side effect in that bugs may come and go, depending on the state of the VID pragma.

If a bug disappears when the module is compiled with debug information on then the bug is likely to be concerned with register corruption.

If a bug only appears when the module is compiled with debug information on then the bug is likely to be concerned with stack memory overwrites.

The debugger will check the date of each of the *.dbd* files it attempts to load against that of the image file containing the process to be debugged. If the *.dbd* file has a later date it will not be loaded.

The VID debug pragma must also be set to min or full while linking the application in order for the *.sym* file to be created.

The JPI environment shipped with this version of the debugger has the optimise for speed pragma set to off. It should always be set to off when building an application that is to be debugged. Arguably, since

turning this pragma on produces larger (although marginally faster) code, it should always be set to off, since code size is of great importance for SIBO machines.

The debugger and symbol information

The debugger maintains symbol information on a per memory segment basis.

When a process is loaded the operating system typically creates two segments, a code segment and a data segment. The debugger knows which segments these are from the process table entry for the loaded process and attempts to load symbol information for each of the newly created segments.

The symbol information for each segment is totally independent of any other information. This allows the debugger to perform symbol information sharing if, for example, multiple processes of the same application are being debugged.

A process may execute code in many different segments. When process execution stops within a segment the debugger will automatically attempt to load the symbol information for that segment, provided it is not already loaded. The debugger uses the segment name to infer the name of the *.sym* file that, in turn, contains the information required to load the *.dbd* files.

The debugger loads the source level symbol information into memory segments on the local machine. All memory segments are required to have a unique name. The debugger uses segment names beginning with at least a two character sequence of any one of YC, YD, ZC, ZD and ZS for different parts and types of symbol information. A view of the segment table of the local machine will show these segments.

Debugger initialisation

On start up the debugger determines the type of screen the PC has, and loads an appropriate screen driver. The debugger supports VGA and Hercules screens.

Once initialised, the debugger reads its command line and configuration files and interprets them as follows:

Debugger command line

The debugger takes a command line of the following format:

```
sdbg [flags] [process name [process command line]]
```

The optional flags are:

-L	to specify local debugging
-Pn	to specify the serial port to use, n takes the value 1 or 2 for COM1 or COM2
-Bn	to specify the baud rate to run at. MC200/400 machines can run at 19200 baud, the HC and Series3 machines at 9600 baud and the Series3a machine at 19200 baud.

If no flags are specified the debugger will run a remote debugging session. Unless an *mclink.trm* file exists (in which case this file determines the port and baud rate) connection will be via COM1 at 9600 baud.

Once a connection with the remote machine has been established the debugger will automatically load any process whose name is included in the command line.

The process command line, if present, is passed to the loaded process when it is run.

Note that the contents of the debugger command line are converted to upper case. If the process name or the process command line need to contain lower case characters you should load the process from within the debugger, rather than by means of the debugger command line.

Examples:

```
sdbg -l
```

starts up the debugger to debug processes on the local machine, without loading any image file.

```
sdbg -l print.img
```

will load the (local) image file *print.img* to run on the local machine.

```
sdbg -p2 -b19200 print.img "This is a remote print"
```

will load the (local) image file to run on a remote machine that is connected to COM2, running at 19200 baud. The command line "THIS IS A REMOTE PRINT" is passed to the loaded process.

```
sdbg rem::m:\test.img "rem::a:\testfile.doc"
```

will load the image file *test.img* from the remote machine's m:\ directory to run on the remote machine, connected to COM1, running at 9600 baud. Note that a file path passed in the process command line is interpreted by (and hence relative to) the remote process. In the above example the file *TESTFILE.DOC* is expected to be found on drive A of the *local* machine.

The process command line may contain any mixture of quoted strings and single byte numeric values, separated by commas. The required content depends on both the particular process and the SIBO machine on which the process is to run. Command line requirements, if any, are described in the appropriate programming guide (see, for example, the *Communicating with the System Screen* chapter of the *Series3 Programming Guide*).

Configuration files

A debugger configuration file is a text file, with name *sdbg.cfg*. Each line starts with a keyword, possibly followed by one or more values. An exclamation mark (!) indicates a comment; following text in that line is ignored.

On start-up the debugger will look for and read two configuration files, the first from the directory in which the debugger *sdbg.exe* exists and the second from the current directory. Typically, the first of these configuration files would contain system-wide keyword definitions and the second would contain application-specific definitions.

The following keywords are recognised:

INITIAL_IP	specifies the symbolic address to which a loaded process should run before the debugging cycle starts. If the symbolic address cannot be found then the debugging cycle begins with the process start up code. If more than one INITIAL_IP is defined, then the last definition is taken.
SOURCE_PATH	specifies a path, in addition to the current directory, which the debugger will search to find the <i>.map</i> , <i>.dbd</i> , <i>.sym</i> and source files required for source level debugging. All paths must be fully specified paths, rather than relative paths. The SOURCE_PATH definitions are cumulative, with paths being searched in the order in which they are defined. Putting the most common path first will speed up searching for symbol and source files.
BREAKPOINT	specifies a symbolic address for an initial breakpoint to be applied to a process. The definition is ignored if the symbolic address can not be found. The BREAKPOINT definitions are cumulative.
F1 to F10	specify the assignment of the function keys F1 to F10 to accelerator key presses. If more than one function key is defined, then the last definition is taken.
TAB_WIDTH	specifies the number of character spaces a tab character represents in the display of a source file. If more than one TAB_WIDTH is defined, then the last definition is taken.
BEEP_OFF	disables the beep which accompanies a transiently displayed error message.
NO_COMMAND_LINE	specifies a null process command line, disabling the prompt for an initial command line when a process is loaded from the Load option in the target view's Process menu.

You may specify more than one value in each BREAKPOINT or SOURCE_PATH command, provided that successive values are separated by commas as in the following example:

```
SOURCE_PATH = c:\sibosdk\hwdemo\d:\dirname\
BREAKPOINT = p_panic,p_notifyerr
```

Note that spaces are not allowed within such comma-delimited lists.

The debugger is supplied with a default configuration file which is placed in the same directory as *sdbg.exe* by the installation process. The following is a commented version of the content of this file.

```
! Sample configuration file for the Debugger

! Specifies the paths, in addition to the current directory,
! to search for source, .MAP, .DBD and .SYM files
! (example lines, commented out)
! SOURCE_PATH=c:\sibosdk\hwdemo\ ! note the terminating '\'
! SOURCE_PATH=d:\dirname\

! Pass zero-length command line to load and run commands
! (example line, commented out)
! NO_COMMAND_LINE

! Turns off the beep that normally accompanies the temporary
! display of an error message
! (example line, commented out)
! BEEP_OFF

! Specifies the tab stop width used when displaying source code
TAB_WIDTH=4

! Specifies breakpoints to set up
BREAKPOINT=p_panic

! Specifies where the debugger will run the process to before it
! reports the loading of the process is complete.
INITIAL_IP=main

! Specifies the assignment of accelerators to function keys
! F1 to F10 may be assigned
F2=B    ! set a Break point at the current line of code
F3=X    ! delete the current foreground view
F4=H    ! run to the highlighted position
F5=M    ! view a source module
F6=V    ! view the highlighted variable
F7=T    ! trace one instruction
F8=S    ! step one instruction
F9=R    ! run
F10=G   ! go to address
```

Note that the function key assignment descriptions given above apply only to menu items in the process window view. If a different type of view is foreground then the accelerators will, in general, invoke a different set of menu items from the current menu.

Remote Debugging

Subject to available memory, you may simultaneously debug up to four processes on a remote machine.

Since the debugger communicates with the remote machine via a channel of the Link process, you must run the Link application on the remote machine before debugging a remote process. It is also advisable to disable auto switch-off.

- on a Series3 set the Remote link option in the Special menu of the System application to *On*. Select Options in the Special menu of the System application and set Auto switch off to *No*. (It is also advisable to set the Update lists item in Options to *System button*.)
- on a Series3a set the Remote link option in the Special menu of the System application to *On*. Select the Auto switch off item in the Control menu of the System application and set Auto switch off to *No*. (It is also advisable to set Update lists in the Set preferences item in the Special menu to *System button*.)
- on an HC enter 'auto -1' and then run the Link application from the system command line.
- on an MC200 or MC400 select and run the Link application icon in the System display. Select the Auto Switch Off option in the Options menu and tick the Always On check box.

It is recommended that the remote machine is connected to a mains power supply since communications hardware is quite power-hungry and will drain the batteries quite quickly.

The debugger communicates with the remote machine via a process called `sys$stub`. If no such process is available on the remote machine, or the version that is available is out of date, the debugger will automatically copy a new version of `sys$stub.img` to the remote machine. The copy is made from the

directory containing *sdbg.exe* (a *sys\$stub.img* is placed there during installation) to the default drive of the Link application process (typically M:\). Because of the need to copy this file, the first invocation of the debugger will take longer to start up than subsequent invocations.

Once a connection between the debugger and the `sys$stub` process has been established, all commands are identical to the local debugging configuration.

Local debugging

Subject to available memory, you may simultaneously debug up to four processes on the local machine. Debugging locally is much faster since the communications overhead is greatly reduced compared with remote debugging. (Although minimised, the communications overhead is the dominant factor in any of the debugger commands.)

In principle, any process can be debugged locally. CLIB programs, whose user interface consists only of console I/O function calls, can be debugged locally quite successfully. Bear in mind, however, that the screen size of different target machines varies. It is strongly recommended that the application be run on the target machine before being released. The default screen size for a CLIB application can be varied by setting the `_DefScreenRect` data structure as appropriate for the target machine.

The user interface libraries and the graphics window servers on the MC200/400, HC, Series3 and Series3a differ from each other. Applications which use these user interface components thus need to be debugged on the appropriate machine. If an application is designed with separate user interface dependent and user interface independent sections, all user interface independent code can be debugged locally.

Simultaneous local and remote debugging

Subject to available memory, you may simultaneously debug up to four processes on the local machine, together with up to a further four processes on the remote machine.

This is particularly useful, for example, in order to debug client-server applications that communicate via a Link channel.

If you wish to debug both local and remote processes you may start the debugger for either local or remote debugging, subsequently making a connection to the other machine, as described later. You have more direct control over the serial port and baud rate if you start up the debugger for remote debugging.

Initial display

A single process list view is created when the debugger is started up. This will contain a list of the processes on either the remote or the local machine, depending on whether the debugger was started up for remote or local debugging.

If the debugger command line included the path of an image file then this process will be loaded and run, up to the position specified by the `INITIAL_IP` command in the configuration file. If the keyword `NO_COMMAND_LINE` does not appear in either configuration file, and if you did not include a process command line in the debugger command line, you will be prompted for a process command line (just press ENTER if you do not need to pass a command line to the process).

A process window is created to display a debugging view of this process. It will appear in front of the process list view.

The fourth part of the document discusses the role of the management in the financial reporting process. It explains how management is responsible for the preparation and presentation of the financial statements. The document also discusses the importance of management's communication with the auditor and the board of directors.

The fifth part of the document discusses the role of the board of directors in the financial reporting process. It explains how the board is responsible for the oversight of the financial reporting process. The document also discusses the importance of the board's communication with management and the auditor.

The sixth part of the document discusses the role of the external auditors in the financial reporting process. It explains how the external auditors provide an independent opinion on the financial statements. The document also discusses the importance of the external auditors' communication with management and the board of directors.

The seventh part of the document discusses the role of the internal auditors in the financial reporting process. It explains how the internal auditors provide an independent opinion on the internal controls. The document also discusses the importance of the internal auditors' communication with management and the board of directors.

The eighth part of the document discusses the role of the external auditors in the financial reporting process. It explains how the external auditors provide an independent opinion on the financial statements. The document also discusses the importance of the external auditors' communication with management and the board of directors.

The ninth part of the document discusses the role of the internal auditors in the financial reporting process. It explains how the internal auditors provide an independent opinion on the internal controls. The document also discusses the importance of the internal auditors' communication with management and the board of directors.

The tenth part of the document discusses the role of the external auditors in the financial reporting process. It explains how the external auditors provide an independent opinion on the financial statements. The document also discusses the importance of the external auditors' communication with management and the board of directors.

The eleventh part of the document discusses the role of the internal auditors in the financial reporting process. It explains how the internal auditors provide an independent opinion on the internal controls. The document also discusses the importance of the internal auditors' communication with management and the board of directors.

The twelfth part of the document discusses the role of the external auditors in the financial reporting process. It explains how the external auditors provide an independent opinion on the financial statements. The document also discusses the importance of the external auditors' communication with management and the board of directors.

CHAPTER 3

THE GRAPHICS INTERFACE

This chapter briefly describes the graphics interface used by the debugger and the included file manager.

Apart from a few keypress variants, it is similar to that used on the MC200 and MC400 machines. If you are familiar with either of these you will probably not need to make more than an occasional reference to this chapter.

Tasks

A running application, or task, is presented within a titled window, with an accompanying command menu bar. The debugger itself uses a number of titled windows, many of which may be visible at the same time, to present various aspects of the debugging process.

A task may be controlled by means of either a mouse or keypresses.

If more than one task is running (that is, if you are using both the debugger and the file manager) you are interacting with only one of them at any given time. This is the foreground task - indicated by the highlight in its title bar.

You can switch from one task to another either by clicking on the task (hold down the ALT key, if you do not want the click to be received by the task itself) or by pressing the INSERT key to cycle round the tasks.

The titled window

A titled window is a rectangular region of the screen, surrounded by a border. The title bar, across the top of the window, contains textual information and three controls to change the size, shape and position of the window.

Moving the mouse pointer into any of these three control areas causes the pointer to be replaced by an appropriate icon.

Resize

The resize control is at the left-hand end of the title bar. Click in this region, or press ALT-[, to activate the resize control. Resize triangles appear on the corners and sides of the window.

With a mouse, you may drag any of these arrows to change the size of the window, or drag in the central area to change the window's position.

Pressing one of the four cursor keys moves the window, and holding down the SHIFT key while pressing a cursor key changes the size of the window (holding down the CTRL key speeds up these processes).

When the window outline is as you want it, press the ENTER key. The triangles disappear and the window takes the new shape. Or press ESC to cancel the resize.

Move

The move control occupies the central region of the title bar. Dragging in this region changes the window's position.

To move the window by means of keypresses, use the window movement keys as described for the resize control.

Zoom

The zoom control occupies the right-hand end of the title bar.

Click in this region, or press ALT-], to switch the window between its current size and its maximum size. (If these two sizes are the same, the zoom control will have no apparent effect.) Repeating the process will reverse the change.

Iconising a window

The rectangular control at the left of the command menu bar is the iconising control. This control is disabled for the debugger itself, but is available in the file manager.

Click on this control, or press ALT-ESC, to shrink the application window to its iconised form. This is useful to clear a cluttered screen, without having to exit the application.

Double click on the icon, or bring the icon to the foreground (with the INSERT key) and press ENTER, to restore the application window, ready to resume work.

The command menu bar

Command menu bars may contain two kinds of controls, rectangular buttons and angled menus. Select one of these by clicking on it, or by holding down the ALT key and pressing one of the number keys along the top row of the keyboard. The buttons and menus are numbered from left to right (not counting the iconise control) for example, pressing ALT-3 in the debugger's process list top-level view will select the Process menu.

A button represents a single command option; selecting one has the immediate effect of executing the corresponding command. Selecting a menu displays a menu list.

When a menu list is displayed, the LEFT and RIGHT cursor keys will switch to neighbouring menu lists. Select an item within the list by clicking on it, or by using the UP and DOWN cursor keys to move the highlight to the required item and pressing ENTER.

Press ESC to cancel a menu selection or move the mouse pointer away from the menu list and click.

There are 3 kinds of menu item, which behave in different ways when you click on them:

- items which lead to a dialog box, needing or providing further information; these items are indicated by ... after the descriptive text
- items which cause something to happen immediately, shown as just descriptive text
- items which are crossed out since they are not available to you at present

Accelerators

Many commands have a keyboard short-cut (or accelerator) to select them without first having to display the menu list. These accelerators are shown on the right hand side of the menu lists.

In the Debugger the commands that have accelerators may be selected by holding down the ALT key and pressing the letter, or just by pressing the letter. For example, the Create File Manager command, which starts up the built-in File Manager, may be selected by pressing ALT-F or, more simply, by pressing F. Note that, in contrast, the accelerators in the File Manager itself may only be accessed by an ALT-keypress combination.

Remember that some of the debugger accelerator keys may be assigned to the function keys F1-F10 in the configuration file.

Dialog boxes

Dialog boxes contain a number of items, or controls, of varying types. Click on a control to select it, or press the TAB key to move the highlight onto the next control within the dialog box: press SHIFT-TAB to move back to the previous one.

Buttons

A button is selected by clicking it, or by moving the highlight to it and pressing ENTER.

Most dialog boxes contain two special exit buttons, labelled CANCEL and ENTER. The ENTER button, selected by pressing ENTER, confirms the current set of choices and exits the dialog. The EXIT button, selected by pressing ESC, aborts the dialog, ignoring any changes that may have been made.

Pop-out menus

Click on a menu, or move the highlight to it and press the SPACEBAR to display its contents. Click on the desired item, or move the highlight with the up and down cursor keys and press ENTER, to select an item.

Tick boxes and diamonds

A tick box offers a Yes/No choice. You set a tick to indicate that you want that option.

Diamonds offer a set of choices which are mutually exclusive - you can choose one and only one.

In either type, click on an item to set or clear it. Alternatively, press TAB until the item you want is highlighted. Then press the SPACEBAR to tick/untick its box or to shade its diamond.

The file selector dialog

The file selector dialog is a good example of a dialog box in that it incorporates most of the elements discussed earlier.

A detailed description of this dialog is included here because it is used in many places in the debugger. For example, in the process list top-level view, selecting the file item in the view menu starts a file selector dialog.

You use the file selector from within an application whenever you want to save, open or create a file.

There are several ways of selecting a file with this dialog box:

- if you know the name of the file and exactly where it is located, you can type the full file name into the Selected File edit box.
- use the pointer or keyboard short-cuts to highlight a directory in the left-hand list box and display its contents (file names and directory names) in the right-hand list, then either select a file name from this list, or type a new name into the Selected File edit box.
- edit the drive, directory and wildcard specification in the Current Directory box (- you can use the Extensions pop-out list in just the same way as in the file manager). Then press TAB or ENTER to see the contents of the directory you want, then select a file name from the list or type a new one into the Selected File edit box.

If you don't specify an extension for your selected file, then the one in the Current Directory box is added automatically. If you really don't want an extension for your file, then type a dot after the name.

Use TAB to move around within the file selector dialog. ALT-SPACEBAR moves to the Current Directory edit box and ALT-DOWN ARROW selects the Extensions pop-out list. The three buttons that change directories are selected as follows:

Alt-right arrow	DESCEND
Alt-left arrow	ASCEND
Alt-up arrow	DEVICES

These keyboard short-cuts are the same as for the file manager, described in a later chapter.

Selecting ASCEND removes the last directory level from the current directory display box and updates both the left and right hand list boxes.

Selecting **DESCEND** adds the directory level currently highlighted in the left hand list box to the current directory display box and then updates both the left and right hand list boxes.

Selecting **DEVICES** displays the top level list of file-system/drives in the left hand list box and displays the contents of the highlighted 'device' in the right hand list box.

Help

Context sensitive help is supplied when the key combination **CTRL-ALT-TAB** is pressed. This displays a dialog box titled **HINTS** and contains two list boxes. The right hand box displays a list of topics while the left hand box displays help information related to that topic.

To change the topic selected, simply use the **UP** or **DOWN** arrow keys to highlight a different topic. The help information in the left hand box changes automatically. The same effect can be achieved using a mouse by simply clicking on the desired topic.

Typically, help information includes various key press combinations and resulting actions.

The dialog can be terminated by pressing **ENTER** or **ESC** or, if using the mouse, by clicking on the **EXIT** button.

CHAPTER 4

TOP-LEVEL VIEWS

The debugger presents the user with a number of independent windows, each with its own menu bar. These are known as top-level views and provide views of a range of aspects of a target machine.

The following types of top-level view are available:

Process list	a list of all processes running on a machine
File	a view of a particular file (assumed to be text)
Segments	a list of all existing segments on a machine
Devices	a list of all existing devices on a machine
Environment variables	a list of all environment variables on a machine
Process window	the main debugging view of a single process

Each of these, with the exception of the process window, is described more fully in the following sections. The process window is described in a separate chapter.

You may bring a particular top-level view and its corresponding menu bar to the front by clicking on it with a mouse. Alternatively you can press CTRL-TAB or use the *Next top-level view* option, with accelerator ALT-N (and which, depending upon the front top-level view, is in either the Debugger menu or the Process menu) to cycle through the views.

Various commands have the effect of creating a new top-level view or bringing one of the top-level views to the front.

In addition to commands whose action is specific to a particular top-level view, many commands are, for convenience, replicated in the menu bars of several views. To avoid undue duplication, these common commands are described once, in the documentation of the first view in which they appear.

If shown, the function key assignment for a command is that made in the default configuration file, described in an earlier chapter of this manual.

Process list top-level view

You may have up to two process list views, one for the local machine and one for any connected remote machine. The title bar of the view informs the user of the machine to which it relates.

A process list view presents the user with a list of processes on either the local or the remote machine. The list is a snapshot of the relevant machine at the time the list is built. The list can be updated at any time by selecting the *Regenerate list* option from the View menu.

A process can be selected from the list by moving the highlight. Various operations can be performed on the selected process.

The current target

The machine on which a process that is being debugged is running is known as the target machine. There are therefore two target machines when the user is simultaneously debugging processes on both the local and remote machines.

At any one time the user is interacting with one particular process. The machine on which this process is running is known as the current target.

When many top-level views exist, it may not always be obvious which machine is the current target. Since all top-level views are derived (that is, created either directly or indirectly) from a process list

view, the process list view from which the current front window is derived always defines the current target.

For example, if the current top-level view is a file view it could be displaying a file from either machine. If, however, it was created from the remote file list view, then the current target is the remote machine.

Selecting the *Process list* option from the View menu will always bring the process list view of the current target to the front.

Debugger menu

Next top-level view Alt-N, N or Ctrl-TAB

Cycle to the next top-level view.

Version

Display a copyright message, the debugger version number and the amount of free memory.

Create File Manager Alt-F or F

Create and run an independent file manager application. This enables you to copy, delete or otherwise manipulate files without having to exit the debugger. It is particularly useful for copying files between the local and remote machines.

Run image Alt-K or K

Present a file selector to select and run an image file. The resulting process runs on the current target machine. For more detail on the dialog, see the section on the file selector dialog in *The Graphics Interface* chapter.

Exit Alt-E or E

Exit the debugger after requesting confirmation.

Local/Remote CPU menu

Connect to Remote/Local

Connect to either the remote or the local machine, depending on whether the current target is either the local or remote machine respectively.

If the connection does not previously exist and is successfully made, an appropriate process list view is created and brought to the front, setting the current target.

If the connection currently exists, the appropriate process list view is simply brought to the front, setting the current target.

Information

Display the machine type and version information about the software components of the current target.

The software built into the ROM of a SIBO machine consists of the EPOC operating system, together with a number of independently built sections of code, many of which exist as separate processes. The version of the software in a particular machine is characterised by the version number of the operating system and of the ROM as a whole.

Process menu

Load

Alt-L or L

Present a file selector to select an image file to load and run as a process on the current target, under control of the debugger.

You are prompted for a process command line, unless the `NO_COMMAND_LINE` keyword appears in either configuration file. The process command line content is as discussed in the earlier description of the debugger command line.

The debugger checks the configuration files for any `BREAKPOINT` keywords. For each one found it attempts to evaluate the symbolic address and, if successful, adds that address to the breakpoint table held for the process.

Execution halts at a temporary breakpoint placed at the address specified by any `INITIAL_IP` in a configuration file. If no `INITIAL_IP` is specified, or if the specified address cannot be evaluated, no process code is executed and execution halts with the instruction pointer positioned at the process entry point.

Once execution has halted the debugger creates a process window. It determines the initial display mode by checking to see if source code information is available for the code at the current instruction pointer address (looking in the current directory and in any paths specified in the configuration files).

If source code information is available the process window is set up to contain a single code view, showing source code at the current instruction pointer. Otherwise the process window is tiled with an assembly language code view, a registers view, a stack view and a data view.

The debugger will automatically download a process to the remote machine if the selected image file is on the local machine and the current target is the remote machine. This can take a significant length of time. Copying the file to the remote machine, for example by using the built-in file manager, will speed up the process, but has the disadvantage that the file must be recopied each time it is changed.

If the process takes a significant length of time to reach the `INITIAL_IP` address, the debugger will present a special top-level view allowing the user to un-load the process, re-load the process, exit the debugging session or set other breakpoints in the process. Early versions of the operating system do not permit the setting of other breakpoints in this situation. If the version of the operating system on the target machine does not support this option then the debugger will report an error.

Break into

Alt-B, B or (F2)

This command brings a running process under the control of the debugger. This is done by allowing the user to set breakpoints in a process at a point that the process will hit in the future, probably in response to some user input.

Select a debuggable (for example, not in the ROM- the debugger cannot set breakpoints in hardware!) running process in the process list menu and select the Break into option. This brings up a special version of a process window with a modified command menu, displaying the code of the selected process. Note that, at this stage, the process is still running.

Select one or more breakpoint addresses, of which at least one should be at a point in the code that the process will hit at some future time. These breakpoints are stored but have not, as yet, been applied to the code.

Use the *Apply breakpoints* option in the Process menu to apply the breakpoints to the code. When the process hits one of these breakpoints it is brought under control of the debugger and the process window reverts to its normal menu.

This mechanism allows multi-process applications to be debugged without any special code being required in the process that launches other processes.

As a typical example, a parent process is debugged to the point where it calls `p_execc` to load another process. If this is successful, regenerate the process list so that it includes the loaded process. This process will be in the suspended state, awaiting the parent process to resume it by calling `p_presume`. Before allowing the parent to call `p_presume`, select the loaded process from the process list and use the *Break into* option. Since execution of the loaded process has not yet started, `main` is a suitable position at which to apply a breakpoint.

Status**Alt-T, T or (F7)**

Display status information about the process highlighted in the process list.

Check data segment**Alt-U or U**

Perform an integrity check on the heap space of the process highlighted in the process list. A dialog shows the result of this check, together with information about stack usage by the process and segment size.

View menu**Process list****Alt-P or P**

Bring the process list view for the current target to the front. If a process list is already highlighted, then selecting this menu item does nothing.

File**Alt-J or J**

Present a file selector dialog to choose a source file to display in a file view. For more detail on this dialog, see *The Graphics Interface*.

Segments**Alt-S, S or (F8)**

Create and display a segment view for the current target.

If the segment view exists it is simply brought to the front.

Devices**Alt-V, V or (F6)**

Create and display a devices view for the current target.

If the devices view exists it is simply brought to the front.

Environment variables

Create and display an environment variable view for the current target.

If the environment variable view exists it is simply brought to the front.

Regenerate list**Alt-R, R or (F9)**

Regenerate the list of processes in the process list view of the current target.

File top-level view

You may have up to two file views, one for the local machine and one for any connected remote machine. The title bar of the view informs the user of the machine to which it relates.

A file view presents the user with a view of a source file.

This top-level view is created by selecting the file item in the view menu of a process list top-level view as discussed earlier.

View menu**Delete****Alt-X, X or (F3)**

Delete the front top-level view.

Locate menu

Go to line

Go to the specified line in the source file.

Search

Alt-Y or Y

Perform a case-insensitive forward search for the specified text. If found, position to the line containing the text. If not found, an error is reported by displaying the message "no matching string found".

Search again

Alt-Z or Z

Perform a case-insensitive forward search from the current position for the text specified in a previous Search command. If found, position to the line containing the text. Again, if not found, an error is reported by displaying the message "no matching string found".

Segment top-level view

You may have up to two segment views, one for the local machine and one for any connected remote machine. The title bar of the view informs the user of the machine to which it relates.

A segment view presents the user with a list of the segments that exist on either the local or the remote machine. The list is a snapshot of the relevant machine at the time the list is built, but the list can be updated at any time by selecting the *Regenerate list* option from the View menu.

The segment list is a symbolic display of the segment table that exists on the target machine. The list will generally contain a small number of additional entries, representing system libraries built into the ROM as *.dyl* files. The debugger simulates these segment handles since many applications use these system libraries.

No new menus or menu items are introduced in this top-level view.

This top-level view is created by selecting the segments item in the view menu of a process list top-level view as discussed earlier.

Devices top-level view

You may have up to two device views, one for one for the local machine and one for any connected remote machine. The title bar of the view informs the user of the machine to which it relates.

A device view presents the user with a list of the segments that exist on either the local or the remote machine. It is a symbolic display of the machine's device table. The list is a snapshot of the relevant machine at the time the list is built, but the list can be updated at any time by selecting the *Regenerate list* option from the View menu.

No new menus or menu items are introduced in this top-level view.

This top-level view is created by selecting the devices item in the view menu of a process list top-level view as discussed earlier.

Environment variable top-level view

You may have up to two environment variable views, one for the local machine and one for any connected remote machine. The title bar of the view informs the user of the machine to which it relates.

An environment variable view presents the user with a list of the environment variables that exist on either the local or the remote machine. The list is a snapshot of the relevant machine at the time the list was built, but the list can be updated at any time by selecting the *Regenerate list* option from the View menu.

You can select a particular environment variable by moving the highlight. Use the *Follow* option of the Variable menu to view the contents of the selected variable.

This top-level view is created by selecting the environment variables item in the view menu of a process list top-level view as discussed earlier.

Variable menu

Modify

Alt-W or W

Present a dialog to modify the content of an environment variable. Data is assumed to be of the current type and radix, set by the *Set format* option.

One or more values, separated by commas, may be typed into the dialog's edit box. These values are written into successive positions in the environment variable, starting at the selected item, overwriting any previous content.

Tick the *Set length* check box to adjust the length of the environment variable's data so that it terminates after the last value written from the edit box.

Note that this menu item is only available if the contents of a selected environment variable is being displayed after having chosen the *Follow* menu item in *this* menu.

Set format

Alt-O or O

Set the format and radix for the viewing and setting of environment variable data.

The data format may be set to one of BYTE, WORD, LONG, FLOAT, or DOUBLE and the radix may be set to decimal, octal or hexadecimal (the radix has no effect for FLOAT and DOUBLE formats).

Again, this menu item is only available if the contents of the selected environmental variable are being displayed.

Clear

Clear the content of the environment variable whose value is currently displayed. Note that the warning message "variable has no value" will be displayed after selection of this menu item.

Again, this menu item is only available if the contents of the selected environmental variable are being displayed.

Follow

Alt-F or F

Switch from displaying a list of environment variables to displaying the data stored in the currently selected environment variable. A movable highlight marks the current value.

The content of an empty environment variable, for example, after using the Clear option, is displayed as a question mark (?).

Previous

Alt-P or P

Return from the display of an environment variable value to the environment variable list.

Search

Alt-Y or Y

Perform a case-independent forward search from the current position for the specified text. If found, select the environment variable whose name contains the text. If not found, an error is reported by displaying the "no matching string found" message.

Search again

Alt-Z or Z

Perform a case-insensitive forward search from the current position for the text specified in a previous *Search* command. If found, select the environment variable whose name contains the text. If not found, an error is reported by displaying the "no matching string found" message.

If there has been no previous search, the behaviour is as for *Search*.

CHAPTER 5

THE PROCESS WINDOW

A process window is the top-level view that is concerned with debugging a single process. Its title shows the name of the process, on which machine the process is running and the status of the process - either Halted or Running.

For a halted process the title bar also shows the number of system ticks that have elapsed from the last time that the process started to run to the time when it was halted.

There is one process window for each process that is running under the debugger's control and so there may be up to eight process windows, showing up to four local and four remote processes.

Each process window contains a number of sub-views showing a range of views of the process. Each sub-view has its own titled window and menu bar. The different types of sub-view are:

Code	an application code module
Code segment	content of the code segment
Data segment	content of the data segment
Registers	contents of the processor registers
Stack	stack content
Symbols	symbol table data
Variable	the value of a variable
Magic statics	values of the reserved static variables
File	content of any text file

The initialisation of a process window is explained in the previous chapter, in the description of the Load option of the process list top-level view. Depending on whether source code is available at the position (INITIAL_IP) where execution first halts, the initial process window contains either a single source code sub-view, or a tiled combination of assembly language code, registers, stack and data segment sub-views. In both cases the code view is the front view.

In the process window ALT-], ALT-[and ALT-SHIFT-] zooms, moves and resizes the front sub-view as described in *The Graphics Interface* chapter. To zoom, move or resize the process window itself, use ALT-CTRL-], ALT-CTRL-[and ALT-CTRL-SHIFT-].

Alternatively, if a mouse is available use it to zoom, move or resize either the process window itself or the front sub-view by selecting and clicking on the appropriate window control.

In addition to commands whose action is specific to the process view or a particular sub-view, many commands are, for convenience, replicated in the menu bars of several views. To avoid undue duplication, these common commands are described once, in the documentation of the first view in which they appear.

If shown, the function key assignment for a command is that made in the default configuration file, described in *Starting up the Debugger* earlier in this manual.

Tracking views

Most of the sub-views that may appear in a process window will update their contents after the execution of code. (The exceptions are the symbols and file views, which assume that their data do not change.)

In addition to updating its contents, the initial code view will move to ensure that the view contains the data at the current instruction pointer value. Such a view is known as a tracking view.

The optional stack view is also a tracking view, following the stack pointer as well as updating its data.

Panics

The operating system will summarily terminate (or panic) a process if it detects any of a number of serious error conditions in that process.

The debugger intercepts all processes that are panicked, regardless of whether the process is currently being debugged or not. If such a process is panicked, a notifier appears, reporting the panic. The process is brought under control of the debugger and displayed in a newly-created process window.

When this happens you can use the procedure explained later, in the description of the stack view, to determine the origin of the panic.

The code view

A code view may display code from any segment and is capable of switching to another segment at any time.

In addition to the initial tracking code view you may create a further four non-tracking code views. This number may be reduced by the presence of code segment and data segment views. You may delete any of these additional code views, but the initial code view is undeletable.

A code view has three modes of displaying code. These are, from lowest to highest:

- assembler
- mixed assembler and source
- source

The *Set display mode* option from the Data menu allows you to select the required display mode. Subject to the availability of source symbol information, the code window will display the code in the highest mode that is compatible with the selected mode. All code views are created with a source level display mode.

Provided the current instruction pointer value matches the address of a line of code being displayed, the tracking code view contains a pointer symbol to indicate the instruction pointer position. (It is possible, when displaying source code, that the instruction pointer value does not match the address of any source line, in which case the pointer symbol will not be visible.)

Note that if the code view is switched from source to assembler or mixed and then back to source again, it is possible that the view will continue to display assembler or mixed code. This is most likely to occur where, in between switching views, the code view has been scrolled to the pre-amble at the front. Where this is the case, simply scroll down again to regain the source view.

If a process changes code segment during execution the debugger will, when the process stops execution, automatically attempt to load any symbol information it can find for that segment (if not already loaded).

The initial 'minimized' (ALT-J) size of a tracking code view is such that the registers, stack and data views, when created, will tile the process window.

Assembly language INT instructions

EPOC uses the INT instruction to implement operating system calls. When viewing code, nearly all INT instructions are symbolically disassembled to the appropriate operating system call.

Process menu

Next top-level view

Alt-N, N or CTRL-TAB

Cycle to the next top-level view.

Status

Display status information about the process being debugged; for example, the process name and the process id.

Reload

Terminate the process then reload it from the original *.img* file, with the original command line. Any breakpoints currently set in the process are preserved.

Only a process that was originally loaded by the debugger can be reloaded.

Unload

Terminate the process being debugged and close the process window. No other processes currently being debugged are affected.

Exit**Alt-E or E**

Exit the debugger after requesting confirmation.

View menu

The menu items in this menu select and display the sub-views of the current process as described at the beginning of this chapter.

Source module**Alt-M, M or (F5)**

Select and display, in a non-tracking code view, a source code module, with the initial view being of the start of the module.

Two list boxes allow selection of one of the application's code segments (it may only have one code segment) and a source module within that segment. The lists only contain those segments and modules for which source code is available.

The primary use of these views is to facilitate the setting of breakpoints.

Code segment**Alt-C or C**

Create a non-tracking code view with initial address of zero within the process code segment (this will typically be assembler).

This provides very similar functionality to the *Source module* option, but allows the creation of a non-tracking code view even if no source code is available.

Data segment**Alt-D or D**

Create a data view with initial address of zero within the process data segment. This menu item will be disabled (as will source module, code segment and symbols menu items) if five such views exist.

Registers**Alt-I or I**

Create a register view. This menu item will be disabled if a register view exists.

Note that, as discussed in the *Introduction*, the segment registers will not be displayed.

Stack

Create a stack view. This menu item will be disabled if a stack view exists.

Symbols

Create a symbols view. This menu item will be disabled (as will source module and code segment menu items) if four such views exist.

This presents a list box containing the names of the segments for which the debugger has loaded a global symbols (*.map*) file. Select the required segment and press ENTER.

Magic statics

Create a view of the magic statics variables for the process. This menu item will be disabled if four such views (including variable views) exist.

Variable

Alt-V, V or (F6)

Create a view of the currently highlighted variable, provided the debugger can find a source level descriptor of that variable. This menu item will be disabled (as will the magic statics menu item) if four variable views exist.

This option is mainly of use when viewing the code in source mode. Trying to use it when viewing in either of the other two modes will generally result in the debugger reporting that no such variable can be found.

File

Alt-J or J

Create a view of any text file. This menu item will be disabled if four file views exist.

Delete

Alt-X, X or (F3)

Delete the front view. The tracking code view can not be deleted.

Run menu

Step

Alt-S, S or (F8)

Execute one unit of code, executing any intervening function calls. The unit is dependent on the current tracking code view display mode. The action is unaffected by the presence or absence of a breakpoint at the current instruction pointer address.

If the tracking code view is currently displaying source code, process code is executed until the next line of source is reached. Any intervening function calls will be executed in full (unless they contain breakpoints). Stepping on a return instruction will cause the process to stop in the calling function.

If the tracking code view is currently displaying assembler or mixed source and assembler, one machine code instruction is executed unless it is a CALL instruction, in which case the code of the function call will be executed in full (unless it contains breakpoints).

The exception to these rules is when stepping over INT instructions corresponding to calls to the two operating system calls `LibLeave` (`p_leave` in PLIB) and `LibSendExit` (return from a message sending call, with no PLIB equivalent). These change the contents of the instruction pointer by effectively performing a far return. If the debugger is requested to step over either of these operating system calls, it attempts to halt execution at the destination address. If this is not possible (say, because the destination is in the hardware ROM) an error is reported.

If there are breakpoints within any call that would otherwise be executed in full, execution will terminate at the first breakpoint encountered.

If this is an unwanted breakpoint at this time you can select the *Previous* option within the Locate menu of the code view to go back to the code position that was stepped from, move the highlight to the line of code the *Step* would have gone to and use the *Run to here* option.

Trace

Alt-T, T or (F7)

Execute one unit of code, tracing into any function call. The unit is dependent on the current tracking code view display mode. The action is unaffected by the presence or absence of a breakpoint at the current instruction pointer address.

If the tracking code view is currently displaying source code execution continues until either the next line of source is reached, or a function call or return is encountered. If a function call is encountered the process will enter that function call before execution halts.

If the tracking code view is displaying assembler or mixed source and assembler this request will cause the process to execute one machine code instruction.

Tracing an INT instruction (usually an operating system function call) will, in general, cause the debugger to step over the INT instruction. Although there is nothing, in principle, which prevents tracing through ROM code, it is not possible to trace into an INT instruction. This would break fundamental operating system rules which do not allow interrupts whilst building an operating system stack frame.

There are exceptions to this rule when the INT instruction corresponds to a call to an operating system function that effectively performs a far call or a far return. The relevant operating system calls, and their PLIB equivalents, are:

LibEnter	p_enter
LibLeave	p_leave
LibSend	p_send
LibSendSuper	p_supersend
LibSendExact	p_exactsend
LibEnterSend	p_entersend
LibSendExit	

Note that LibSendExit is only called by the operating system and does not have, or require, any C equivalent.

If one of these INTs is encountered by the debugger it will determine the destination address, generate a temporary breakpoint there and allow the process to run to that breakpoint. If the temporary breakpoint would be in the hardware ROM, the debugger reports an error.

Run

Alt-R, R or (F9)

Run the process until it terminates or until a breakpoint is hit.

If there is a breakpoint at the current instruction pointer, one machine code instruction is traced before the breakpoints are applied. This allows code that is performing a repetitive task to execute the next cycle before hitting the same breakpoint. In this situation, many debuggers require you to trace one instruction manually before running to execute the next loop.

Run to here

Alt-H, H or (F4)

Run to the position indicated by the highlight. The debugger generates a temporary breakpoint for that address and allows the process to run.

Not all source code lines have an address, if one of these is selected then the debugger will report an error and not run the code.

To run to the start of a function you must set the highlight on the first line of code within the function, rather than on the function declaration.

This is because the JPI compiler does not output the source line number and machine code offset for any function declaration line of code. (It may also be noted that the first line of source code in a function generally does not mark the first machine code instruction executed within a function - there will usually be additional code to generate a stack frame.)

Break menu

Breakpoints

Alt-K or K

Set or clear one or more breakpoints.

The dialog shows a list of the current breakpoints. You may enter additional breakpoints, or either delete or temporarily disable any of the existing ones.

To add a breakpoint, enter into the edit box a function name or a function name plus an offset at which execution is to be halted and press ENTER. If an offset value is included, be absolutely sure that this points to a valid instruction or the results will be unpredictable.

Each breakpoint has an associated pass count (set, by default, to 1). This is the number of times which the process must pass the breakpoint address before the debugger actually halts execution. This feature is useful for setting breakpoints within loops, to examine the process state after a certain number of times around the loop. The maximum pass count that can be set is 65535.

Selecting the *Enabled* check box changes the enabled state of the highlighted breakpoint.

Ticking the *Disable breakpoints* check box disables all breakpoints. It is a convenient means of temporarily disabling breakpoints whilst, for example, stepping through code. On clearing the *Disable breakpoints* check box, all breakpoints revert to their enabled states, as shown in the list box.

Note that the Enabled tick box, the Pass_Count display and the Delete button will not be displayed in the dialog box if there are NO existing breakpoints.

Break here**Alt-B, B or (F2)**

Set a breakpoint at the highlighted line.

Not all source code lines have a corresponding address. If you attempt to set a breakpoint on such a line the debugger will report an error.

To set a breakpoint at the start of a function you must set the highlight on the first line of code within the function, rather than on the function declaration (or use the *Breakpoints* option and type in the function name). For an explanation, see the earlier description of the *Run to here* option in the Run menu.

Data menu**Set display mode****Alt-O or O**

Select the preferred display mode as one of:

- assembler (lowest)
- mixed assembler and source
- source-level code (highest)

Depending on the availability of source information, the code is displayed in the highest mode that is compatible with the selected mode.

Check data segment**Alt-U or U**

Perform an integrity check on the heap space of the process being debugged from the current window. This is equivalent to a call to the PLIB function `p_atchk`. A dialog shows the result of this check, together with information about stack usage by the process and the segment size.

AL as error code

Display the error text, if any, associated with the value in the AL register.

Many operating system calls generate errors. These are indicated by setting the Carry flag and placing a negative error code into the AL register. (The PLIB library functions `sign` extend that number into a negative error number in the AX register and thus return a negative result.)

AL as panic code

Display the panic text, if any, associated with the value in the AL register.

The operating system will summarily terminate (or panic) a process if it detects any of a number of serious error conditions in that process. On such a termination, a reason code is placed in the AL register to assist in identifying the nature and possible cause of the error condition.

The debugger intercepts all processes that are panicked, regardless of whether the process is currently being debugged or not.

Locate menu**Follow****Alt-F or F**

Attempt to determine the address to which the process would go if the current instruction were executed, and move the highlight in the code view to that address. The intervening code is *not* executed.

For a **CALL** instruction the debugger will go to the start address of the function being called.

For a **BRANCH** instruction (conditional or unconditional) the debugger will follow the branch, *irrespective of the current value of the flags register*.

For an **INT** instruction the debugger will usually go to the instruction following the **INT**. Exceptions to this are **INT** instructions corresponding to the operating system calls **LibSend**, **LibSendSuper**, **LibSendExact**, **LibEnterSend**, **LibEnter**, **LibLeave** or **LibSendExit** for which the highlight will be set to the appropriate destination address. For further explanation, see the description of the *Trace* option in the **Run** menu. Note that the debugger uses the current register set to determine the destination address, on the assumption that the **INT** instruction was reached by executing code. If the registers have not been set up to contain the correct values the debugger will generally report an error, but may go to the wrong address.

For any other instruction the debugger will go to the next instruction/source line.

Previous

Alt-P or P

Return to the last previous stored position in the view of the code.

Up to the last eight previous positions are stored automatically by any of the options that move to another position in the code (for example, *Step*, *Trace*, *Goto address*).

Goto address

Alt-G, G or (F10)

Display code at the specified symbolic address.

The address can be specified as a valid decimal or hexadecimal value.

To go to an address within the current segment, simply enter the offset within that segment. To go to a different segment, enter the symbolic name of that segment, followed by a colon (:) and the offset within that segment. To go to an address that is currently stored in a register you may enter the register's symbolic name, for example, entering **IP** (or **ip**) moves to the position indicated by the instruction pointer.

If the address can be evaluated the code view is moved to the specified address.

Goto line

Alt-L or L

Display code at the specified source code line number.

This option has no effect if the current code view is not showing source code.

Search

Alt-Y or Y

Perform a case-independent search forwards from the current position for the specified text. An error is reported if the text is not found.

This option is only effective if the current code view is showing source code.

Search again

Alt-Z or Z

Search forwards from the current position for the next occurrence of the text specified in the most recent *Search*. An error is reported if the text is not found.

If no previous search has been made, then the effect is the same as for *Search*.

This option is only effective if the current code view is showing source code.

Breakpoints in dynamic libraries and other shared code

Processes running under the EPOC operating system frequently make use of shared code. Two simultaneously running processes of the same program, for example, share the same code segment. There is only one loaded copy of the code of a dynamic library, irrespective of the number of processes that are currently accessing it.

Setting a breakpoint in shared code therefore means that all processes sharing the code will, at some time, encounter the breakpoint.

The debugger is notified of all such events, but can distinguish between breakpoints that are encountered by a process under its control and those that are not. Processes not under the debugger's control are thus not affected by the breakpoints, except by a decrease in their speed of execution (provided, of course, that the breakpoints were set by the debugger).

When a process opens a dynamic library, the code may or may not be physically loaded, depending on whether the code is currently in use by another process. Similarly, the code may or may not be unloaded when a process closes a dynamic library.

The debugger can not guarantee to distinguish between these two cases, or to know whether the code has been unloaded, or been unloaded and either reloaded or replaced by other code while not being accessed by the process being debugged. If you are in a situation where this may have occurred, using the *Source module* option of the View menu will force the debugger to re-evaluate its internal record of the contents of memory segments. As part of this process the debugger will disable breakpoints that are set in segments that no longer exist.

To guard against breakpoint errors you should not attempt to set a breakpoint in a dynamic library until you are sure that it is loaded, that is, until the process you are debugging has opened the library.

For similar reasons, it is advisable to either disable or remove breakpoints in:

- dynamic library code, before the library is closed by the process being debugged
- before exiting any code that may be shared with another process

The data view

A data view presents a view of the contents of a single data segment. Each process may have up to five data views.

By default the initial view of the data is as bytes, in a byte dump format. When the data view display format is of bytes you can move the highlight between the left-hand byte display and the right hand text representation with CTRL-LEFT ARROW and CTRL-RIGHT ARROW.

When the process stops running, a data view is automatically updated to reflect any changes in the data currently being displayed.

Typically a data segment dynamically changes size during process execution, as demands are made on the heap. The debugger resizes the view limits as appropriate.

Although a data segment may be up to 512k bytes in size (the maximum size of an EPOC operating system segment) most segments, including the process data segment, do not exceed the 64k byte directly addressable limit of the 8086 processor. Data window addresses are automatically displayed either as 16 bit or 32 bit numbers, depending on whether the data segment size is less or greater than 64k bytes.

Locate menu

Follow

Alt-F or F

Display data at the address indicated by the highlight in the data window. If the address is out of range for the current data segment the debugger will report an error.

If the display is of bytes then the highlighted value is assumed to be the low byte of the address and the next byte in the display is taken as the high byte. If the display is of words or longs then the address is simply the highlighted value. An error message is presented if the display is of floats or doubles.

This option is useful, for example, to follow linked lists in the data space.

Previous

Alt-P or P

Return to the last previous stored position in the view of the data.

Up to the last eight previous positions are stored automatically by use of the *Follow* or *Goto address* options.

Goto address**Alt-G, G or (F10)**

Prompt the user for a destination address and, if the address is valid, move the view to that address.

The address can be specified as a valid decimal or hexadecimal value.

For an address within the current data segment, just enter the segment offset. To move to another segment, enter the symbolic name of the segment, followed by a colon (:) and the segment offset.

An error is reported if the destination address is outside the valid address range for the relevant data segment.

Data menu**Set data format****Alt-O or O**

Select the data display format to be in bytes, words, longs, floats or doubles. For the first three of these formats the radix may be selected to be octal, decimal or hexadecimal. Floats and doubles may be displayed only in decimal.

Modify**Alt-W or W**

Change one or more items of data within the data segment, where each item corresponds with a unit of displayed data in the current format.

The user is presented with a dialog whose title indicates the current segment name and offset (corresponding to the highlight in the display). Modifications to the data will be made from this address onwards.

The dialog accepts a series of comma delimited items which should represent values to be entered at successive addresses, in the context of the display. For example, if the current display format is of word values, all the items are expected to evaluate to words and will replace successive words in the data segment.

Quoted strings (using either double or single quotation marks) are valid input when the display is of bytes. Use single quotes to insert the characters as typed, and double quotes to insert the characters as a zero-terminated string. In all cases, non-quoted strings are assumed to be symbolic values (including register symbolic names).

Items can include decimal and hexadecimal values provided that they evaluate to sensible values for the current display format.

If an item can not be validly evaluated, that item is highlighted and an error status message is displayed. All items up to the one containing the error will have been written to the appropriate addresses.

The registers view

Each process can have only one registers view, showing the register set for the process being debugged. The register set is evaluated each time the process is halted after executing some code. The registers and status bits that have changed from the previous set are highlighted.

By default the view shows only the current register set. Increasing the size of the window allows it to show up to the last 8 register sets.

The processor flags register is displayed as a series of bits having the value of 0 (clear) or 1 (set). The characters that represents the status bit flags are as follows:

O	Overflow
D	Direction
I	Interrupt
S	Sign
Z	Zero
A	Auxiliary carry
P	Parity
C	Carry

The segment registers are not displayed. As is described earlier, in the *Introduction* chapter, segment addresses do not have any great significance in application programs running on SIBO machines, since the EPOC operating system may move memory segments. For this reason the registers view does not display the contents of the segment registers.

Also, some 8086 flags are not displayed. The Trace flag, for example, is used by the debugger itself to trace instructions. An application may not set the Trace flag; if it attempts to do so it will be suspended by the debugger.

Data menu

Set radix

Select display of register values in octal, decimal or hexadecimal.

Modify

Alt-W or W

Present a dialog to modify the contents of the registers and/or the status bits.

To set a register to symbolic value select the *Evaluate* button to bring up a dialog in which the symbolic name can be entered.

The stack view

Each process may have only one stack view. It is a tracking view, displaying the stack of the process being debugged.

The stack is displayed as a series of 16 bit values. When the process stops running the stack view is automatically updated to reflect any changes in the SP register and the data on the stack.

If the debugger loaded the process it knows the stack size (this information is stored within the loaded *.img* file) and thus limits the upper address of the display to the stack top. If the debugger did not load the process now under control of the debugger and it cannot find the *.img* file the process was loaded from, the stack will extend to the end of the data space of the process.

Operating system stack frames.

All operating system calls generate an operating system stack frame. This consists of the interrupt frame generated by the INT instruction which implements the call, followed by the DS and ES segment registers, the BP register, the operating system stack frame linkage value and finally various values - including any registers that need to be preserved. (Interrupts are disabled for the duration of this process, to prevent memory being moved while the stack frame is being built.)

The typical instructions executed when an operating system function is called are:

int	xx	generates Flags, CS and IP on stack
push	ds	
push	es	
push	bp	
push	[DatOsFramePtr]	a magic static value
mov	[DatOsFramePtr], sp	
mov	bp, sp	

The stack view typically presents this as:

0908	DatOsFramePtr
090A	BP register
090C	ES register
090E	DS register
0910	IP return address
0912	CS return address
0914	Flags return value

The BP register may be used by the operating system as a scratch variable, but the value at DatOsFramePtr - memory address 30 (0x1e) in the process data space - always contains a pointer to the last stack frame.

The CS and IP return address will indicate the code address that the operating system call was made from. Viewing this code will enable the user to track further back and find the local function calls and parameters to those functions.

Unfortunately the CS return address is the absolute segment address and the debugger has no mechanism by which it can automatically determine the code segment to which this value refers. You can determine the code segment by creating a segment view from the appropriate target and looking in the address field for the CS value.

Very occasionally, memory segments may have moved between the time the debugger read the stack data and the time the segment list was created and so it is possible that the CS value does not match any item in the list. In such a case, regenerating both the segment view and the stack view (delete it and recreate it) will resolve any differences.

Locating the origin of a panic

When a process is panicked by the operating system, the debugger stops the process at the point at which the process was panicked.

To find out what code the process was executing, you have to trace back through the stack frames to build up a list of the operating system calls that have been made. Before doing so you must first perform the following steps:

- use a data view (or magic statics view) to find the contents of memory location `DotOsFramePtr` - memory address 30 (0x1e) - in the process data space
- move the data view to that memory location and read its contents
- move the stack view to this address

The stack may now be interpreted as described earlier.

The extra level of indirection is required because the process, when panicked, makes a further operating system call to tell the debugger that it has panicked. In this situation the SP address must be adjusted via the data space contents, as described above.

Locate menu

Follow

Alt-F or F

Move the stack display to the address given by the contents of the currently highlighted stack item.

This option is useful, for example, for following operating system call frames.

Previous

Alt-P or P

Return to the last previous stored position in the view of the stack.

Up to the last eight previous positions are stored automatically by use of the *Follow* or *Goto address* options.

Goto address

Alt-G, G or (F10)

Prompt the user for a destination address and, if the address is valid, move the view to that address. You may enter the symbolic name of a register (BP and SP are particularly relevant) to go to the address contained in that register.

The address can be specified as a valid decimal or hexadecimal value.

An error is reported if the destination address is outside the valid stack address range.

Data menu

Modify

Alt-W or W

Modify one or more stack values.

The user is presented with a dialog whose title indicates the current segment name and offset (corresponding to the highlight in the display). Modifications to the data will be made from this address onwards.

The dialog accepts a series of comma delimited items which should represent 16 bit values to be entered at successive addresses. Items can include decimal and hexadecimal values. For example, if the highlight is at address 0x9c0 then entering 1,2,3 will set the word at address 0x9c0 to 1, the word at 0x9c2 to 2 and the word at 0x9c4 to 3.

If an item can not be validly evaluated, that item is highlighted and an error status message is displayed. All items up to the one containing the error will have been written to the appropriate addresses.

Quoted strings are not valid data in the stack view. (If you want to modify a string on the stack, you can position a data view to the appropriate location and use its *Modify* option.)

The symbols view

A symbols view displays a list of symbols and their corresponding addresses for a memory segment, read from a *.map* file.

If a program has multiple symbols for a single address, the debugger will only load the first symbol encountered for that address when reading the *.map* file. The symbols view enables you to determine which of these symbols the debugger has loaded.

A common way that this may arise is in the case of a function can take a variable, but limited, number of parameters. To allow JPI prototyping to check the number of parameters being passed to the function, you might declare a separate function for each of the parameter variants. The actual code, however, may only exist as one routine.

The variable view

Each process may have up to four variable views, each of which provides a symbolic display of a program variable. A variable view will be updated automatically when the process stops execution after a trace, step or run.

To view a variable you should move the code view highlight to the variable to be viewed and select the *Variable* option from the View menu.

The debugger follows C scope rules when selecting the variable to be viewed, that is, automatics are selected before statics.

Before viewing an automatic variable (declared on the stack) the instruction pointer must be within the function in which the variable is declared - and you must have run, traced or stepped to at least the first line of the source code. (When the debugger traces into a routine it does not execute the stack frame generation code automatically. Creating a variable view immediately after stepping into a function will therefore not display data at the correct address.)

Beware of selecting a variable when instruction pointer is in another function which declares a variable of the same name - the debugger will generate a view of the variable in the function containing the instruction pointer, rather than the selected one.

The variable name is handled in a case-insensitive manner and the debugger will not, for example, distinguish between the variables *myvariable* and *myVariable* if they are both declared in the same function. It is, in general, unwise to choose names that differ only slightly from each other.

The debugger does not inform you if the variable goes out of scope. In such a case the variable contents will typically show nonsense values.

Display formats for variables

The variable view provides a multi-level view of complex data types. Each level of the view presents a cross section of the data structure under examination. The levels can be traversed by using the *Follow* and *Previous* options of the Locate menu.

Within a variable view, ENTER and ESC are keyboard shortcuts for *Follow* and *Previous*.

The basic data types such as chars, ints, longs, floats and doubles form terminals of a data structure; the debugger cannot traverse the data structure through any of these variable types. These differ from arrays, structs unions and pointers; the debugger can traverse these data structures to provide the next level view.

Basic data types

Signed and unsigned chars, signed and unsigned ints, signed and unsigned longs, floats and doubles have the following display format:

(address)	name	type	value
-----------	------	------	-------

The address field is the hexadecimal address of that variable within the process data segment. The name field is the symbolic name of the variable, as selected in the source code view. The type field indicates the variable type. The value field displays the current value of that variable, with the decimal value in brackets, if appropriate.

These variables can not be followed. The debugger will simply present the same view if the *Follow* option is selected.

For example:

(0x832)	i	signed int	0x43 (0067)
(0x834)	lng	unsigned long	0x21 (00000033)
(0x838)	d1	double	1.2e10

Pointers

All pointers are 16 bit values and are displayed as follows:

(address)	name	type *	ptr_value (data)
-----------	------	--------	------------------

The address field is the hexadecimal address within the process data segment of that variable. The name field is the symbolic name of the variable, as selected in the source code view. The type field indicates the type of the variable at which the pointer points. The * character(s) indicate the current level of indirection of the displayed data; multiple * characters indicate that there are several more levels of indirection before the terminal data is reached.

The ptr_value field displays the current value of the pointer. The data field is only displayed if the pointer is at the first level of indirection. It shows the first element of the data to which the pointer points. The data value is shown in decimal with, if appropriate, a preceding hexadecimal value.

For example, pointers to each of the above example fields would show:

(0x802)	p	unsigned char *	0x832 (0x43,0067)
(0x804)	plng	unsigned long *	0x834 (0x21,00000033)
(0x806)	pd1	double *	0x838 (1.2e10)

If the pointer is one to a basic data type (bytes, ints, longs, floats and doubles) using the follow option will display up to 128 bytes at the address pointed at in the mode of the pointer, ie a pointer to longs will display up to 128 bytes (32 longs) as longs.

If the pointer is one to a more complex data type, that data type will be displayed.

Structures and unions

If the data element being viewed is a structure or union, each field of the structure, or element of the union, is displayed on a separate line. If one of the elements is itself a structure, this is indicated. Selecting such a line by highlighting it and using the *Follow* option from the Locate menu will generate a new view level, with the sub-structure fields now being individually displayed.

Arrays

If the elements of an array are of a single basic data type, the array elements are presented in a layout similar to that of a data view. The display format is automatically determined from the array element type.

For arrays of complex data types, each array element is displayed on a separate line, as for structures.

Enums

Enums are displayed in the same way as the basic data types. Provided the current value of the enum is within its defined range, an additional value field shows the symbolic name of the enumeration value.

Bitfields

A bitfield is treated as a structure, with each component field being displayed as a basic data type. The addresses of all the component fields will, however, be identical.

The *Modify* option of the Data menu can not be used to modify an individual component field. It will only modify the bitfield as a whole and it is up to the user to specify the relevant bit pattern if only one field is to be modified.

Data menu

Modify

Alt-W or W

Change the values of one or more variables.

The user is presented with a dialog whose title indicates the current segment name and offset (corresponding to the currently highlighted data). Modifications to the data will be made from this address onwards.

The dialog accepts a series of comma delimited items which should represent values to be entered at successive addresses, in the context of the display. For example, if the display is of a pointer, each item is expected to evaluate to a word.

You are advised to exercise caution when entering more than one item since this will, in general, modify more than one variable.

Quoted strings (using either double or single quotation marks, as for the *Modify* option of the data view) are valid input when the display is of bytes. In all cases, non-quoted strings are assumed to be symbolic values.

Items can include decimal and hexadecimal values.

If an item can not be validly evaluated, that item is highlighted and an error status message is displayed. All items up to the one containing the error will have been written to the appropriate addresses.

The magic statics view

A magic statics view is a form of variable view which displays a list of the magic statics (or reserved statics) for the process. See the description of reserved statics in the *Processes and Interprocess Messaging* chapter of the *PLIB Reference manual*.

No new menus or menu items are introduced in this view.

The file view

You may have up to four file views for each process, subject to available memory.

A file view presents a view of a text file, similar in appearance to that available in a file top-level view.

As with all file views, the file to be viewed is selected by making appropriate choices in the file selection dialog as discussed in *The Graphics Interface* chapter.

CHAPTER 6

THE FILE MANAGER

The file manager supplied as part of the debugger is identical to that supplied with MC400 and MC200 machines. It provides a comprehensive set of file management functions to perform operations on files, directories or devices. You may, for example, use it to:

- browse through a filing system
- copy, delete and rename files
- make, delete or copy directories
- back up files to or from a remote system

The file manager presents a titled window containing a file name edit box, an Extensions button and two list boxes.

The left hand list box shows a list of all parallel directories, at the same level as the current directory. Initially, this list box shows a list of devices, including devices on any connected remote machine.

The right hand list box contains a directory listing of all files specified by the current path and file name (which may contain wildcards) shown in the file name edit box (and selected in the left hand list box). The list is headed by any subdirectories, regardless of the wildcard specification, displayed in bold.

Moving around the file manager

Click on the appropriate item, or use the following keypresses:

- TAB moves between the two list boxes
- ALT-SPACEBAR moves to the file edit box, enabling you to edit its contents
- ALT-DOWN ARROW selects the Extensions button, enabling you to select an extension from the list of those present in the current directory to replace that currently in the file edit box

You can move into a subdirectory listed in the right hand list box by highlighting it and pressing ENTER.

In addition to the normal ALT-number means of selecting menus from the menu bar:

- ALT-LEFT ARROW selects the Ascend menu button, which moves up a directory level
- ALT-RIGHT ARROW selects the Descend menu button, which moves into a subdirectory
- ALT-UP ARROW selects the Devices menu button, which moves straight to the devices level

Many commands bring up a dialog box at the bottom right of the screen. Press CTRL-TAB to move between the main window and such a dialog box or use the mouse to point and click, if available.

There is no need to remove one dialog from the screen before selecting another item - the new dialog will automatically remove the old one.

Operations on files

Selecting files

Edit the contents of the file name edit box directly, or use the Extensions button and directory-changing menu buttons to see an appropriate list of files. To select a file, highlight it by clicking on it, or by using the up and down arrow keys.

To select several files with related names, use wildcards in the file name edit box to select the required group. If the names of the files not related you can still select them all at once by *tagging* them, provided they are all in a single directory.

Tagging files

To tag a file, highlight it then either select Tag File from the Tag menu, press SHIFT-UP ARROW or SHIFT-DOWN ARROW, or hold down SHIFT and click on the file. A tick appears next to the file name. (Note that you can't tag sub-directories.)

To remove this tag, select Untag from the Tag menu (or, as for tagging, press SHIFT-UP ARROW or SHIFT-DOWN ARROW, or SHIFT-CLICK). In effect, actions such as SHIFT-CLICK toggle the tag indicator.

To tag all the files in one directory, highlight the directory name in either list box and select Tag All from the Tag menu.

If you want to tag most, but not all, files in a directory, it is quicker to use Tag All and then Untag those you don't want.

When you have tagged files, the tags remain until you do one of the following:

- copy or delete the tagged files
- set their file attributes
- move to a different directory and tag a file there.

You can move around the directory structure while files are tagged and they will still be tagged when you come back to that directory.

To clear all the tags at once, select Clear Tags from the Tag menu.

Copying files

First select the file or group of files you want to copy by any of the methods described above, then select Copy from the File menu. The Copy Files dialog appears with your selection entered in the Copy edit box.

Select the place to where you want to make the copy, either by typing into the To edit box, or by moving back to the main window and selecting the destination device and directory.

If appropriate, you can give the copy a different name, by typing into the To edit box.

Tick the Include SubDirectories check box to include files that match the current file name specification in all subdirectories. A typical use would be with a wildcard of * to copy a whole directory structure.

Tick the Modified Files Only check box to copy only those files with the modified attribute set. During the copy the modified attribute is removed from the original files, but is retained in the copy files. This is useful for backing up files which have been modified since the last backup.

Click on COPY, or press ENTER to copy the files.

One or more of the files specified in the To box may already exist in the destination directory. In this case, a message box appears to inform you that a file of the same name already exists. You are offered these buttons:

- CANCEL - cancel the whole operation.
- SKIP - don't replace this file but go on to the next one (if you specified more than one). If the file manager comes across another file with a name which already exists, you are offered these same four choices again.
- REPLACE - replace the existing file with the new file (and go on to the next one if you specified more than one)

- **REPLACE ALL** - replace the existing file with the new one and do the same for all subsequent files; all files are copied and you are not offered these choices again.

You may then select and copy further files, if you wish. Click on EXIT, or press ESC, when you have finished.

Renaming files

Rename in the File menu offers a dialog box asking for the file name to change. You can specify this file by either:

- typing the file name into the edit box
- selecting the file from the right-hand list box so that its name is entered automatically into the Rename edit box.

The rename dialog asks for the new name. This must be in the current directory.

If you want to rename a file to a different directory, use Copy in the File menu, specifying the new name and directory, then delete the old version.

Deleting files

Specify the file(s) to delete in the same way as for copying.

As with copying, you can delete more than one file at a time by using wildcards or by tagging. If using a wildcard, you can also delete matching files in sub-directories by ticking the 'Include SubDirectories' tick box.

A message appears, asking for confirmation before deleting anything.

Attributes

You can set or clear file attributes by selecting Attributes from the File menu. The Attributes dialog box is displayed, containing the four choices, Modified, Read Only, Hidden and System.

As with Copy and Delete, tagging, or wildcards and the Include SubDirectories tick box can be used to set attributes on a number of files at once.

Modified	The Modified attribute is the equivalent of the DOS archive attribute. The file manager automatically sets the Modified attribute on any file which you create or change without backing up. For each file that has this attribute set, the text 'Mod' appears after the file name when it is displayed in the main window.
Read Only	Setting this attribute for a file allows you to view and edit the file as usual, but you can no longer save changes to it. For each file that has this attribute set, the text 'RdO' appears after the file name when it is displayed in the main window.
Hidden and System	These two attributes are for files used by the operating system; you would not normally change them. Files with the Hidden attribute set have 'Hid' after their name in the file manager; those with the System attribute have 'Sys'. Files in either category will not be listed in a file selector dialog.

Changing the order of a directory listing

Select File Order from the File menu to specify the criteria by which you prefer the contents of a directory to be listed.

To start with, they are displayed by Name - in alphabetical order, first directories, then files. If you tick Descending Order, files now appear before directories in the list, and both groups are listed Z to A.

A choice of one of the following criteria is offered:

Name	lists files alphabetically according to their file names.
Extension	lists files alphabetically according to their file extensions.
Date	lists files according to when they were last saved, from oldest to newest.
Time	lists files according to the time of day they were last saved, from oldest to newest.
Size	lists files according to their size in bytes.

In each case, Descending Order reverses the order. If you had selected Date, Descending Order would display the newest files first. In all cases it also places files before directories.

Note that after re-ordering the list, the list box display is always reset to the top of the list.

Operations on directories

Creating a directory

First, select the place where you want to create your new directory. Then select Make Directory from the File menu. The Make Directory dialog appears, containing your selection in its Make edit box. Type the name of your new subdirectory onto the end of the text in this edit box and press ENTER.

When making a directory, any intermediate directories are created as necessary.

Removing directories

To remove directories, highlight the directory in the main window, then select Remove Directory from the File menu. The Remove Directories dialog appears, with the highlighted name in its edit box.

On pressing ENTER, a message appears to warn you that any files in the directory will be automatically deleted as the directory is removed.

You should be aware that Remove Directory *always deletes all sub-directories of the directory specified, and all files contained in all these directories.* (It is not the same as the DOS RMDIR or RD command.)

Operations on devices

Devices include the internal memory drive and Solid State Disks of a connected remote EPOC machine, and the drives on the local filing system. With the Device menu you can use:

- Copy - to back up the entire contents of a device to another device
- Name - to give a volume name to a device (remote SSD devices only)
- Info - to find out the name and size of a device, and see how much free space remains on it

The Format option has no effect in the PC environment.

CHAPTER 7

TROUBLESHOOTING

No source code displayed

If the *Source module* option in the View menu shows the source module for which source code is required, the debugger has successfully loaded the *.dbd* symbol file, but has failed either to find or to load the actual source file. This may be for one of two reasons:

- the source file is not in any of the specified search paths - add the path to one of the two *sdbg.cfg* configuration files.
- the debugger refuses to load the file because the source file modification date is later than that of the *.dbd* file - recompile the file and relink the application.

If the source module does not appear in the list presented by the *Source module* option, the debugger has failed either to find or to load the *.dbd* file. This may be for one of the following reasons:

- the file does not exist - ensure that the VID debug pragma is set to min or full, recompile and relink.
- the file is not in any of the specified search paths - add the path to one of the two *sdbg.cfg* configuration files.
- the debugger refuses to load the file because the file modification date is later than that of the *.img* - recompile the file and relink the application.
- a *.sym* file, which identifies the *.dbd* files, is missing - ensure that the VID debug pragma is set to min or full and relink.

The debugger will only attempt to load symbol information when a view is moved to a different segment. Thus if an application loads a dynamic library or a device driver, a view has to be moved to the segment containing that code in order to force the loading of source level information. This can be done, for example, by using the *Goto address* option of the Locate menu to go to address zero in the segment containing the newly loaded code.

A further reason may be that the application has been built with the optimise for speed pragma set to on - code that is to be debugged should always be built with this pragma set to off.

When code is compiled with the optimise for speed pragma set to on, the JPI compiler places directives in the output object file telling the linker to start each section of generated code (each function) on an even byte boundary. The linker performs this by padding out the code as necessary. Unfortunately it pads it out with NULL bytes instead of NOP instructions. When the debugger disassembles code (a function that is performed very frequently, even when debugging at source level) the NULL is interpreted as an opcode and is disassembled into a 3 byte instruction, using the first two bytes of the code of the following function. This invariably leaves the start of the next instruction (as far as the disassembler is concerned) in the middle of an instruction. As a result of this the debugger is unable to match disassembler output addresses with any source code line numbers and thus fails to present the user with source code.

If, having checked all the above points, you still do not have source code displayed, this may mean that there is not enough free memory to load the source module data. In this case you should either increase the amount of free memory (if possible) or alter the application's *.pr* project file so that fewer modules are built with debug information.

A typical modified .pr file could contain:

```
#system epoc img
#model small jpi
#pragma debug(vid=>full)
#compile module1
#compile module2
#pragma debug(vid=>off)
#compile module3
#compile module4
#compile module5
#pragma link(hwif.lib)
#pragma debug(vid=>full)
#link appname
```

Communications link broken

The debugger will only report a broken communications link if it attempts to communicate with the remote machine while the link is broken.

The communications link can be broken in several ways:

- the remote machine has switched off
- a pack door has been opened
- the link cable has been removed.

If the link is broken due to the remote machine switching off (either via auto switch off or opening the pack door on an HC) switch the machine back on, wait a few moments to allow the link channels to be re-established and then retry the required operation.

If the link is broken by opening a pack door, simply close the pack door and retry the required operation.

If the link is broken due to the cable being removed, replace the cable and retry the required operation.

If a remote process hits a breakpoint whilst the link is broken, the debugger will not receive notification of the event. In such a case the only option is to terminate the debugging session and restart it.

Poor, distorted or missing display

During the start-up process the debugger detects the type of the host machine's graphics display and loads a window server process containing the appropriate screen driver. It is possible that, for some unusual configurations, the wrong screen driver may be selected. This will result in a poor or distorted display, or even a failure to display anything.

The two files concerned are:

WSRVMCHR.IMG	for Hercules
WSRVMCVG.IMG	for VGA

The installation procedure copies these files into the same directory as *sdbg.exe*.

If you suffer from this problem and you know what kind of display your machine uses, copy the correct file to overwrite the other one. If, for example, you know that your machine uses Hercules graphics then, in the directory containing *sdbg.exe*, type:

```
copy wsrvmchr.img wsrvmcvg.img
```

If you are not sure what kind of graphics display your machine uses, first copy both files into another directory.

Then copy one of these files back to overwrite both files in the directory containing *sdbg.exe* and try running the debugger again. Repeat this, overwriting both files with the second of the copies, until the debugger display is correct.