

SIBO 'C' Software Development Kit

GENERAL PROGRAMMING MANUAL

Version 2.30

March 1, 1999

(C) Copyright Psion PLC 1990-98

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

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

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

TopSpeed is a registered trademark of Clarion Software Corporation. 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 Installation 1-1

Installation.....	1-1
Directory structure.....	1-2
Reconfiguring the TopSpeed project system.....	1-2
Customising the redirection file.....	1-3

2 A Brief Overview Of The SIBO SDK..... 2-1

Manuals	2-1
Where to start.....	2-2
The terms SIBO and EPOC explained	2-2
Fatal programming errors (Panics).....	2-3
Panic numbers.....	2-3
C++ and Object Oriented programming	2-3

3 Building An Application 3-1

Introduction	3-1
Equipment required.....	3-1
Which files are needed	3-2
A first example application	3-2
A first look at project files	3-2
Creating the .img file	3-3
Copying the program to the target machine	3-3
Running the program on the target computer.....	3-3
Stepping through the program with the SIBO Debugger.....	3-4
A PLIB version of Hello World	3-4
PLIB and CLIB contrasted	3-4
The code for p_hello.c	3-5
The epocinit statement in .pr files.....	3-6
Housekeeping batch files and re-using project files.....	3-6
A general project file: unnamed.pr	3-6
A batch file for test compilation.....	3-6
Some complications.....	3-7
TSC versus TSCX	3-8
More complicated programming setups.....	3-9
Simple PLIB examples	3-9
Multi-file programs	3-9
Graphics programs.....	3-10
HWIF programs	3-10
Customised libraries.....	3-10
Applications containing assembler as well as C	3-11
Other advanced use of the project file system.....	3-11
Greater control over the image file created	3-12
Priority, minimum heap, and version number.....	3-12
Use of edump.exe	3-12
Differences between .app files and .img files	3-13
Add-file lists	3-13
Changing the set of add-files in an image.....	3-14
Further uses of emake.exe	3-14

4 Notes on CLIB..... 4-1

Functions missing from the MS-DOS JPI C library.....	4-1
int86x.....	4-2
intr.....	4-2
File handle conversion routine.....	4-2
The console channel.....	4-3
MS-DOS file names.....	4-3
Floating point emulator.....	4-3
Panic 80.....	4-3
Building the CLIB library and header objects.....	4-4
Cautionary note.....	4-4

5 Fundamental Programming Guide-lines 5-1

Introduction.....	5-1
Other related documentation.....	5-1
The source code for the examples.....	5-2
User interface.....	5-2
A first look at multiple event sources.....	5-3
Remarks on timers.....	5-4
Remarks on keypresses.....	5-4
More on p_iowait.....	5-4
Status words: the other side of p_iowait.....	5-5
Prioritisation of event sources.....	5-5
I/O devices in general.....	5-5
The console device.....	5-6
How to cancel a timer.....	5-6
Where to declare status words.....	5-7
A first look at error handling.....	5-7
Error handling in Events.....	5-7
Fatal errors and non-fatal errors.....	5-8
When resources need to be tidied explicitly.....	5-8
Inter-Process Communication: Events2 and Subproc.....	5-8
A third event source.....	5-8
How Events2 passes data to Subproc.....	5-9
The code in Subproc.....	5-10
Mechanisms for inter-process communication.....	5-10
Debugging cooperating applications.....	5-11
Error handling in Events2.....	5-11
Socially responsible programming in Events2.....	5-11
Data received from a serial port.....	5-11
Opening the serial port.....	5-12
Active and inactive event sources.....	5-12
Debugging applications with serial comms.....	5-13
The role of p_tickle.....	5-13
Yielding CPU in compute-intensive programs.....	5-13
Idle objects.....	5-14
The meaning of calling p_iosignal.....	5-14
One drawback of continuous activity.....	5-14
Remarks on process priorities.....	5-14
The call p_ioyield.....	5-14
General remarks.....	5-16
Multi-threadedness and multi-tasking.....	5-16
Subprocess or idle object?.....	5-16
Window redraws as an event source.....	5-16
The APPMAN and ACTIVE classes in OLIB.....	5-16

6 Copy-Protecting Software..... 6-1

Introduction	6-1
The free space method.....	6-1
First generation copying methods.....	6-2
Preserving checksums	6-2
What kind of change should be made	6-2
Restricting the number of copies made	6-3
Low level SSD information	6-3
Copy-protection by changing the ROM	6-3

7 Compatibility 7-1

Introduction	7-1
What machine am I running on?	7-1

CHAPTER 1

INSTALLATION

The *Psion SIBO 'C' Software Development Kit* (SDK) enables you to develop applications in 'C' for the Psion SIBO family of hand-held and notebook computers:

- the HC range of corporate handheld computers
- the Series3 range of palmtop computers
- the MC range of laptop mobile computers.

The SDK is PC based - that is, you write and build your applications on a IBM PC or compatible and then run (or debug) the application on a SIBO computer.

The SIBO SDK comes in three variants: *Professional (Prof)*, *Standard (Std)*, and *Documentation (Doc)*. The Prof and Std variants contain differing amounts of the *TopSpeed C Package* (produced by the Clarion Software Corporation):

- the Prof variant contains the complete TopSpeed C Package
- the Std variant omits the TechKit and C Library Source Kit portions of the TopSpeed C Package.

Parts of the TopSpeed C Package are required in order to build any SIBO application. These are the parts included in the Std variant of the SDK.

The TopSpeed C Package may be used independently of the SDK to write C programs to run on IBM PCs and compatibles.

All three variants of the SDK contain the same three volumes of SDK documentation, comprising 14 different manuals all told, together with associated software produced by Psion. This software incorporates libraries, header files, auxiliary programming tools, example programs, and much more besides.

Installation

There are three phases to installing the SIBO SDK:

- installing the small code model of the TopSpeed C Package
- installing Psion software and example programs
- customising the TopSpeed system using special Psion files.

For information on installing the TopSpeed system, see the TopSpeed C documentation. (You may install other code models as well as the small code model, but they will not be used by the SIBO SDK).

The Psion software consists of *required* parts and *optional* parts. **For a full description of all the available files, see the *read.me* files in the root directories on the supplied disks.**

The files on the disks are in compressed ("zipped") form. The disks contain the *pkzunzip* program, which can be used to decompress ("unzip") the other files as they are copied.

The installation procedure ensures that each group of files is copied to its correct directory. The *read.me* files on the disks specify which files are present in each group.

Directory structure

The TopSpeed software is usually located in a `\ts\` directory tree.

The Sibosdk software is usually located in a `\sibosdk\` directory tree.

There is no need for the `\ts\` and `\sibosdk\` directory trees to be on the same partition of your hard disk. Eg the TopSpeed system could be on drive `c:`, with the Sibosdk system on drive `d:`.

Both the `\ts\sys` and `\sibosdk\sys` directories should be added to the MS-DOS path. Alternatively, you may copy the contents of `\sibosdk\sys` (with the probable exception of the *ts.red* redirection file - see *Customising the redirection file* below) to `\ts\sys`, in which case you will only need to add the `\ts\sys` directory to the MS-DOS path.

Subdirectories under `\sibosdk\` include:

<i>lib</i>	the location of standard libraries, startup objects, and some loadable device drivers (dynamic extensions to the SIBO operating system)
<i>include</i>	the location of header files
<i>sys</i>	the location of the SIBO Debugger and other miscellaneous programming tools
<i>pr</i>	the location of some standard project files
<i>demo</i>	the location of various standard demo programs
<i>s3atool</i>	Series 3a versions of an icon editor and the Spy application
<i>src</i>	(not present in all versions) the location of the source code for the Clib (standard C) library
<i>hwdemo</i>	(optionally) the location of programs demonstrating use of the HWIF library
<i>hwifsrc</i>	(optionally) the entire, buildable, source of the HWIF library, for interest and/or to allow the writing of extensions to HWIF
<i>wd</i>	(optionally) the source printer scripts of some WDR printer driver files
<i>ldd</i>	(optionally) the source code of some example device devices
<i>hcmast</i>	(optionally) software allowing alternative versions of the HC rom to be built
<i>wkdemo</i>	(optionally) source code of example software for the <i>Workabout</i>
<i>fconv</i>	(optionally) software allowing the creation of file format conversion DYLS for Word
<i>oopdemo</i>	(optionally) the location of programs demonstrating the basic use of the Object Oriented Programming system (other directories contain more advanced examples)
<i>record</i>	(optionally) the source of the Object Oriented Series 3a Record application. This is buildable, provided that the <code>\sibosdk\oop</code> directory has also been installed

The SIBO SDK requires between 3 and 6 Megabytes of disk space, depending on how much of it is installed.

Reconfiguring the TopSpeed project system

The Sibosdk software contains files, including *tsprj.txt* and *tsmain.txt*, that are replacements for files of the same name released by TopSpeed. The Sibosdk versions of the files allow the TopSpeed project system to handle the EPOC IMG system type, and the Sibosdk tools generally, in addition to the systems normally supported by TopSpeed. (See the following chapter for more details.)

The Sibosdk versions should be copied, from `\sibosdk\sys`, over the TopSpeed versions in `\ts\sys` (as guided by *read.me*). Copies of the original TopSpeed files should be kept, with their extensions changed to *.old* (say).

In case any specialised changes have already been applied to, say, your *tsprj.txt* and *tsmain.txt* files, you should re-apply these changes to the Sibosdk versions.

In order for all these changes to have any effect, the TopSpeed tool *tscfg* has to be run:

type `cd \ts\sys` to enter the appropriate directory

type `tscfg` to invoke *tscfg*.

Note that the replacement files are not the same as those supplied with versions of the SDK earlier than 2.0. *You must therefore repeat the reconfiguration process, even if you are upgrading from a previous SDK version.*

Customising the redirection file

The `\sibosdk\sys` directory contains the redirection file *ts.red* that allows the TopSpeed project system to find the SIBO SDK header files, library files, and other required files. When you invoke the project system to build an Epoc application it must be able to find this file. If there is no such file in the current directory, the TopSpeed system looks for this file in the directory containing the TopSpeed software itself (usually `\ts\sys`). So there are two options:

- overwrite the TopSpeed *ts.red* file (in the `\ts\sys` directory) with the Psion *ts.red* file (in the `\sibosdk\sys` directory), and place a copy of the original TopSpeed *ts.red* file into the directory in which you are writing PC C applications (if any)
- copy the Psion *ts.red* file into the directory in which you are writing a SIBO application.

The first option is more appropriate if you are writing more SIBO applications than PC applications, the second if you are not.

The contents of the default *ts.red* file is basically as listed below. Again, note that the supplied file is different from the *ts.red* in earlier versions of the SDK.

```
*.PR = .; C:\SIBOSDK\PR;
*.H = .; C:\SIBOSDK\INCLUDE;
*.HPP = .; C:\SIBOSDK\INCLUDE;
*.RH = .; C:\SIBOSDK\INCLUDE;
*.G = .; C:\SIBOSDK\INCLUDE;
*.RG = .; C:\SIBOSDK\INCLUDE;
*.XG = .; C:\SIBOSDK\INCLUDE;
*.RSG = .; C:\SIBOSDK\INCLUDE;
*.INC = .; C:\SIBOSDK\INCLUDE;
*.C = .; C:\SIBOSDK\SRC;
*.CPP = .; C:\SIBOSDK\SRC;
*.CAT = .; C:\SIBOSDK\SRC;
*.A = .; C:\SIBOSDK\SRC;
*.OBJ = .; C:\SIBOSDK\LIB;
*.LIB = .; C:\SIBOSDK\LIB;
*.HLP = C:\TS\DOC;
```

For example, the meaning of the line

```
*.H = .; C:\SIBOSDK\INCLUDE;
```

is that any *.h* file referenced in the course of building an application should be searched for first in the current directory, then in the `c:\sibosdk\include` directory.

To increase the search path for *.h* files, so that private *.h* files can be located from some other directory, just edit the *ts.red* file so that the line becomes, for example:

```
*.H = .; ..\INCLUDE; C:\SIBOSDK\INCLUDE
```

Note that the contents of this file assume that you have installed the Sibosdk system on your *c:* drive. If you have installed it on another drive you will have to edit *ts.red* to refer to the appropriate drive.

CHAPTER 2

A BRIEF OVERVIEW OF THE SIBO SDK

The SIBO SDK consists of a set of floppy disks containing the SDK software and a number of manuals describing the software and the Psion SIBO family of mobile and hand-held computers. The floppy disks contain the various tools, header files, libraries etc that are required to produce an image (.img) file that will run on Psion SIBO machines.

Manuals

The SDK manuals supplement the TopSpeed manuals. In particular you are referred to the *TopSpeed C Library reference* for the description of standard C (CLIB) library functions.

The SDK contains the following manuals:

<i>General Programming Manual</i>	This manual. It describes how to install the SDK, and how to build an application that will run on the SIBO machines. It also contains some notes on using the <i>TopSpeed C Library reference</i> manual, some discussion on how to copy protect software for SIBO computers, and a potentially very important chapter entitled <i>Fundamental Programming Guidelines</i> .
<i>HC Programming Guide</i>	Some documentation particularly oriented around the HC computer range.
<i>Series 3/3a Programming Guide</i>	Some documentation particularly oriented around the Series 3 range of machines.
<i>Workabout Programming Guide</i>	Some documentation particularly oriented around the <i>Workabout</i> computer.
<i>EPOC O/S System Services</i>	Describes the software interrupt interface to the ROM-based system services. An appendix contains a complete listing of the EPOC service numbers. This will be of particular use to OPL and assembly language programmers who wish to make direct access to EPOC services.
<i>Additional System Information</i>	Describes the MCLink communications software, resource files, WDR printing, various file formats, how to write device drivers, and other miscellaneous topics.
<i>PLIB Reference</i>	Describes the Psion PLIB library (this library provides the most direct access to the ROM based system services).
<i>Window Server Reference</i>	Describes the window server library WLIB which implements a range of sophisticated graphics operations.
<i>I/O Devices Reference</i>	Describes the interface to some of the Epoc device drivers, including those which access the serial and parallel ports.
<i>The SIBO Debugger</i>	Describes how to use the SIBO Debugger to debug applications being developed for SIBO computers.
<i>Hardware Reference</i>	Describes the basic hardware of SIBO computers.
<i>Programming in HWIF</i>	Describes the HWIF library (which contains support - mainly for the Series3 - for menus, dialogs, editors, and printing).

<i>Object Oriented Programming Guide</i>	Describes the development of applications using Object Oriented techniques. Its introduction provides an overview of Psion's Object Oriented programming system and of the OLIB, HWIM, FORM and XADD object libraries.
<i>ISAM Reference</i>	Describes the ISAM dynamic library which provides Indexed Sequential Access management for large database files.
<i>OLIB Reference</i>	Describes the OLIB object library.
<i>FORM Reference</i>	Describes the FORM object library.
<i>HWIM Reference</i>	Describes the HWIM object library.
<i>XADD Reference</i>	Describes the XADD object library.

Where to start

There is a very large amount of information in these manuals - perhaps too much for any one person to keep it all in their head. However, few developers will need to make detailed reference to more than around half of the manuals.

The manuals contain many cross references to sections where various topics are discussed in more detail. Use these cross references to help you find your way around the SDK documentation. The chapter immediately after this one is probably the best place to start. Alternatively, start reading the *HC Programming Guide*, the *Series 3/3a Programming Guide*, the *Object Oriented Programming Guide*, the *Plib Reference* manual - or anything that catches your eye as you flick through the pages.

The terms SIBO and EPOC explained

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

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

The hardware architecture is primarily implemented in custom ICs called ASICs. The SIBO architecture uses surface-mounted static CMOS ICs throughout. For further information see the *Hardware Reference* manual.

The EPOC operating system, designed for the SIBO architecture, has the following features:

- preemptive multi-tasking
- MS-DOS-compatible file systems
- installable file systems, including remote file access
- asynchronous services
- support for client-server architectures (used to implement system components such as the file server and window server)

- a comprehensive I/O system with many built-in I/O devices
- dynamically loadable device drivers
- reentrant function library
- multiple processes of the same program share a single copy of the code
- support for object-oriented programming
- code-shared dynamic link libraries.

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

See the *Plib Reference* manual for more details.

Fatal programming errors (Panics)

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

As well as protecting the system from defective applications, the panic system enforces a greater discipline on application code by terminating a process as soon as the condition is detected.

Panic numbers

Programming errors detected within different areas of the system code give rise to different panic numbers. The following table lists the possible panic numbers and the corresponding system code that can give rise to them.

0 to 80, and 255	The PLIB library
81 to 129	The Window Server library
130 to 160	The OLIB object library

These panic numbers are described in more detail in the appropriate manuals.

Panic numbers in the range 160 to 254 are used by code that is not resident in the ROM (such as the ISAM library). A given panic number may be used by more than one piece of code; such a panic may therefore have one of a number of causes. The only definitive way to discover the origin of such a panic is to make use of the SIBO Debugger to catch the panic and then trace it back to its source.

C++ and Object Oriented programming

This version of the SDK is compatible with TopSpeed C++ although, at the time of writing, the TopSpeed C++ package is not supplied by Psion as part of any variant of the SDK.

Note that the kinds of classes, and the means of creating and accessing them, in C++ are quite different from those in the Psion Object Oriented programming system.

Regardless of whether you use C or C++ to develop Object Oriented applications, if you wish to use Psion objects you must create instances of them and send messages to them by the mechanisms that are described in the *Object Oriented Programming* chapter of the *PLIB Reference* manual and the *Object Oriented Programming Guide*.

If you intend to develop a SIBO application that uses *only* Psion objects, you may still use C++ as a "better" version of C, without making use of its Object Oriented aspects.

You may, if you wish, use a mix of Psion and C++ classes, provided you make sure that you create and use instances of classes of each kind by the appropriate means. It is quite acceptable, for example, to use Psion classes for the application manager and user interface, but use C++ classes in the 'engine' of an application.

CHAPTER 3

BUILDING AN APPLICATION

Introduction

The end result of developing a program to run on a SIBO computer is normally an *image* program, with characteristic extension *.img*. Essentially, *.img* files are to Epoc what *.exe* files are to MS-DOS. (Sometimes, the image file is given the extension *.app* instead. See later for the distinction between *.img* and *.app* forms of images.)

Image files are actually produced via an intermediary *.exe* file by the operation of a tool *emake.exe*, though in practice this process is automated on behalf of the developer. The point is that the development cycle for *.img* files is basically the same as for *.exe* files in more traditional programming environments. The same compile-link-debug cycle exists in both cases.

The program is written, compiled, and linked on the PC. These steps involve the TopSpeed compiler and linker, and TopSpeed *project* (*.pr*) files. These steps may also involve the full TopSpeed *ts* development environment, which is an Integrated Development Environment (IDE). Alternatively, developers may prefer a more traditional approach, involving their own favoured stand-alone text editor, and a batch-file mechanism for invoking the TopSpeed compiler and linker.

Once linked, the program is transferred to the SIBO computer in one of three ways:

- under the control of the SIBO Debugger
- via a SSD written to by a PC SSD drive and then placed into the SIBO computer
- via a serial link, using Comms software such as MCLink.

A program running on a SIBO computer (though not one in its rom) can be debugged under the control of the SIBO Debugger running on a PC. If the program has been built under special conditions, source level debugging will be available; otherwise just machine-code level debugging.

The remainder of this chapter gives more details on all the above points:

- it introduces some particularly relevant aspects of the TopSpeed project file system
- it describes some batch files that may be found useful when building applications
- several example applications demonstrate the points made (all the example applications and associated files referred to in this chapter are optionally copied into `\sibosdk\demo` when the SDK is installed)

Equipment required

The following equipment is required in order to create an application that runs on a SIBO computer:

- One target SIBO computer (HC, MC, or Series3)
- One PC
- The TopSpeed C development system on the PC
- The Psion SIBO SDK software on the PC
- One or other form of communication between the HC and the PC.

Around 5 to 7 Mbytes of PC hard disk is required to install the TopSpeed C development system and the Psion SIBO SDK software. There is no particular requirement for the PC to have extended memory. Development can take place on an XT, but a 486 machine with around 4 Mbytes of extended memory machine will obviously produce results more quickly.

Which files are needed

To build an application, the following files are required:

- source files, such as *.c* and *.h* files
- a project file, with extension *.pr*
- a redirection file, with name *ts.red*
- libraries, with extension *.lib*.

At the same time, various batch files (extension *.bat*) may be found useful.

Additional files may be required for the development of applications with the aid of the Object Oriented system - see the *Object Oriented Programming Guide* for further details.

There is no need to have a separate project file or redirection file for every single application. For example, ordinarily there will only be one redirection file on any one PC, shared between *all* applications written on that PC. One possible exception is in the case of developers who use the TopSpeed system to write PC programs as well as SIBO programs.

A first example application

The following discussion centres around a very simple "Hello World" program.

The source C code is as follows:

```
/*
HELLO.C

CLIB Hello World application
*/

#include <stdio.h>

int main(VOID)
{
    printf("Hello World");
    getchar();
    return(0);
}
```

Evidently, the program prints the message *Hello World* onto the screen, waits for the ENTER key to be pressed, and then terminates.

A first look at project files

Before *hello.c* can be compiled and linked, a *project file* needs to be specified. In this case, the following (*hello.pr*) suffices:

```
#system epoc img
#model small jpi

#compile hello
#link hello
```

The meanings of the last two lines of this file are obvious enough: the file *hello.c* should be compiled, and then the resultant object file linked to create *hello.img*. Note that in any case of ambiguity, the extension *.c* should be added to the name specified in the *#compile* statement.

The first two lines in *hello.pr* are less obvious. Their meanings are as follows:

<code>#system epoc img</code>	The end outcome of the build is a <i>.img</i> file, as defined in the Epoc-customised part of the TopSpeed configuration (alternative <i>#systems</i> include <i>dos</i> and <i>win</i>)
<code>#model small jpi</code>	The code is to be compiled in <i>small</i> model (code and data segments each restricted to 64K), with the <i>jpi</i> (TopSpeed C) convention of using registers to pass parameters to subroutines.

These two lines *must* be present in *all* project files used to build applications for SIBO computers.

Other possible contents of *.pr* project files are discussed later in this chapter.

Creating the *.img* file

There are two ways a *.pr* file can be used to create a *.img* program:

- *inside* the TopSpeed *ts* programming environment (see TopSpeed documentation for full details)
- *outside* the *ts* environment.

For example, one reason for working *outside* the *ts* environment would be to allow the use of another text editor, such as *Brief*.

The remainder of this programming manual describes use of *.pr* files outside of the *ts* environment.

For the moment, simply type:

```
tsc /m hello
```

to have *hello.c* compiled and linked, with the end result (among other files) being *hello.img*.

The significance of the */m* parameter is that the project file is executed in "make" mode, with files not being recompiled or relinked needlessly.

Copying the program to the target machine

If your PC has a set of SSD drives attached, simply copy *hello.img* onto an SSD in one of these drives, and then insert the SSD into the target machine (the SIBO computer). Otherwise, you may wish to use MCLink as follows:

- connect the SIBO computer to your PC using a suitable cable
- run the Link application on the target machine (type `link` at the HC command line, click on the Link icon on an MC, or set Remote Link on in the System Screen of a Series3)
- run MCLink on the PC (eg by typing `MCLINK` if `lsibosdk\sys` is on your path)
- adjust the MCLink serial port and baud rate parameters if required
- type `copy hello.img rem::m:\hello.img` to transfer the program to the *m:* drive of the remote machine (or copy the file to `rem::m:\img\hello.img` on a Series3)

For more details about the MCLink program, see the chapter *Mclink, Mcprint, and Slink* in the *Additional System Information* manual. (For example, that chapter explains how the whole process of running MCLink can be handled via time-saving batch files at the PC end of the connection.) Finally, another way the program can be transferred from the PC to the target computer is by using the SIBO Debugger, as discussed below.

Running the program on the target computer

The way the program is run on the target computer varies from computer to computer:

- on an HC, simply type `hello` at the `$` prompt of the Command Shell
- on an MC, use the Run menu command of the System application, and select *hello.img*
- on a Series 3 or Series 3a, the entry *Hello* will appear in the file list of the RunImg application when this list is next updated (assuming that *hello.img* has been copied into a *\img* top-level directory), so that *hello.img* can be run simply by positioning the highlight over *Hello* and pressing ENTER.

Stepping through the program with the SIBO Debugger

In order to debug a program, there is no special need to copy it "by hand" onto the target machine.

Just type

```
\sibosdk\sys\sdbg hello
```

(or equivalent) at the PC end, and ensure Link is running on the target computer. (These instructions assume that *hello.img* is in the current directory on the PC.) In due course, the debugging screen will appear on the PC. (When debugging a program running on an MC, the baud rate for the Debugger to use may have to be given explicitly - eg `\sibosdk\sys\sdbg -b19200 hello`.)

Step through the program (use F8 or ALT+S) until it hangs (waiting for a key to be pressed on the target computer). Or simply run the program to completion (use F9 or ALT+R).

In fact, if *hello.img* has been built as specified above, the debugging screen will come up in *machine code level*. In order to debug at the source code level, a slight change has to be made in the way *hello.img* is built:

- either a line `#pragma debug(vid=>full)` has to be inserted in the project file (before any instruction to `#compile`)
- alternatively, a parameter `/v2` can be added to the command line invoking the project file.

Thus typing

```
tsc /m hello /v2
```

at the PC command line builds a version of *hello.img* suitable for source level debugging.

As before, just type

```
\sibosdk\sys\sdbg hello
```

but this time, the debugging screen by default starts in source level mode, and supports inspection of variables, etc. See the *SIBO Debugger* manual for more details.

A PLIB version of Hello World

PLIB and CLIB contrasted

The above "Hello World" program uses the so-called *CLIB* library.

CLIB is a version of the TopSpeed C library for the EPOC operating system. As such, it is a version of the standard ANSI C library. The functions in CLIB are described in the *TopSpeed C Library Reference* manual. Additional notes, including a list of the TopSpeed C library functions that are not implemented, may be found in the following chapter, *Notes on Clib*.

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

The benefits of using CLIB are:

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

However, for many programs, developers are strongly urged to consider using not *CLIB* but *PLIB* - Psion's proprietary C library. Whilst PLIB differs from the ANSI standard in many places, there are good reasons for all these differences, so as to best take advantage of the EPOC architecture. In particular:

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

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

See the *Introduction* chapter of the *PLIB Reference* manual for more details.

In fact, unless you are using the in-built user interface object dynamic libraries (accessed using object-oriented programming), you can freely mix PLIB calls with CLIB. Experienced C programmers can, if they wish, initially use the more familiar CLIB functions and regard PLIB and WLIB (the window server library) as they would regard non-portable components of any C library.

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

The code for `p_hello.c`

The code for a PLIB version of the above program *hello.c* is contained in the file *p_hello.c*:

```
/*
P_HELLO.C

PLIB Hello World application
*/

#include <plib.h>

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

and a corresponding project file *p_hello.pr* would be

```
#system epoc img
#set epocinit=iplib
#model small jpi

#compile p_hello
#link p_hello
```

The following differences will be noticed between *hello* and *p_hello*:

- the PLIB program uses a Psion-proprietary header file (*plib.h* in this case)
- the PLIB program uses Psion-proprietary function calls (*p_*xxx functions)
- the PLIB program links with a different library (this is one effect of the *epocinit* line in the project file - discussed further below).

To build *p_hello.img*, just type either `tsc /m p_hello` or `tsc /m p_hello /v2` (the latter producing a version supporting source-code debugging).

As a result of the differences between *hello* and *p_hello*, a substantially smaller *.img* file is produced (try it and see).

The reason for the remarkable codesize improvement of the PLIB program is that, as mentioned earlier, the functions in the PLIB library provide only *very thin shells* over functionality that is present in the ROM of the SIBO computer. PLIB programs make better use of the SIBO ROM software than do CLIB programs. Being tailored to the particular needs of SIBO computers, PLIB evolved with very different constraints and objectives from standard C libraries. In many cases, PLIB functions can be claimed to "improve" upon the specification of their nearest CLIB equivalents.

The difference in size between CLIB and PLIB programs is not always so remarkable as in the above example - it depends on the number and types of library function calls made. Indeed, it is perfectly possible to write some parts of an application using CLIB, and others in PLIB. This fact considerably simplifies any process of converting a previous large programming project from one computer system to the SIBO SDK system.

Whilst it is possible to avoid PLIB entirely, this is *not* recommended. Time spent gaining familiarity with the functions in the PLIB library should prove an excellent investment, aiding the production of leaner and more powerful applications. In any case, familiarity with PLIB is a pre-requisite for accessing many other parts of the SIBO ROM software - such as the enhanced graphics facilities of the Window Server.

The epocinit statement in .pr files

In *p_hello.pr*, the command

```
#set epocinit=iplib
```

sets the value of the project macro %epocinit. Note that this command must precede the #model command in any .pr file.

This command serves two purposes: it specifies whether you are using the CLIB or PLIB startup object files, and it specifies the stack size for the .img file produced. The allowed values of %epocinit are:

iclib	CLIB startup, 8k stack (recommended size when using CLIB startup)
iclib4	CLIB startup, 4k stack
iclib2	CLIB startup, 2k stack
iplib	PLIB startup, 4k stack (recommended size when using PLIB startup)
iplib8	PLIB startup, 8k stack
iplib2	PLIB startup, 2k stack

If %epocinit is not set then it defaults to iclib (as in *hello.pr*).

You must use the CLIB startup object files if you are writing a program that includes any CLIB library I/O functions. It is, however, possible to use many CLIB library functions (for example, the memory allocation functions) in conjunction with the PLIB startup. If you do not use any CLIB library functions then you should always use the PLIB startup.

See the *Introduction* chapter in the *PLIB Reference* manual for more about startup object files.

Housekeeping batch files and re-using project files

A general project file: unnamed.pr

Clearly, a project file such as *p_hello.pr* can be used, with only nominal changes, for a wide range of other similar programs.

Consider the related project file, *unnamed.pr*:

```
#system epoc img
#set epocinit=iplib
#model small jpi
#compile %main
#link %main
```

in which the only difference from *p_hello.pr* is that references to *p_hello* have changed into %main.

Any batch file that invokes *unnamed.pr* has to set the value of %main as a parameter to tsc. For example,

```
tsc /m unnamed.pr /smain=%1
```

with the TopSpeed /s construct being used to set the value of %main to the variable passed into the batch file.

A batch file for test compilation

A programmers' text editor usually has some means to compile or "test compile" a source file, from inside the editor. Many programmers find this a considerable boost to productivity.

For example, *Brief* supports compilation on the ALT-F10 hot key, with the way the compilation is done being determined by an MS-DOS environment variable:

- during *autoexec.bat* (or a batch file called therein), set the value of bcc, eg to !"cc.bat %s"
- the effect of ALT-F10 while editing a .c file would then be to run the batch file *cc.bat*, passing the basic name of the file (ie less the path and extension) into the batch file
- the leading exclamation mark specifies that compiler *warnings* should be reported, as well as *errors*.

The TopSpeed *ts* integrated development environment naturally possesses an equivalent mechanism, but some users may prefer to use an independent text editor.

Accordingly, one suggestion is that there should be a file *cc.bat* in the local directory (or in the path), with the following contents (or equivalent):

```
tsc %1.c /fpunnamed
```

The meaning of the */fp* construct is that the specified project file should be used (in this case, *unnamed.pr*).

Given that there is no */m* in this command (nor any */l*), the specified project file is invoked in so-called "compile" mode: nominated files are compiled, without any files being linked. Further, the compilation *always* takes place, without any calculation of whether an object file is already "up-to-date".

Once all the required C source files in a project have been successfully compiled, the programmer can exit the editor, and then "make" the project in the normal way:

- no time will be wasted in recompiling files unnecessarily
- a *.img* file will be produced (if the make is successful).

The *cc.bat* file used for test compilation could be accompanied by a *make.bat* file that invokes the project file in "make" mode.

Some complications

There are a couple of shortcomings with the above batch file *cc.bat*:

- it takes no account of whether files should be compiled with full debug information
- it takes no account of a possible specialised *.pr* file: the project file *unnamed.pr* is hard-wired.

It does not take too much imagination to come up with a more general scheme - as is embodied in the batch files *cc.bat* and *make.bat* actually shipped with the demo files *p_hello.c* etc:

- a test should be made for the existence of a project file with name *%1.pr*
- attention should be paid to the value of an environment variable for whether to generate full debugging information.

The batch files supplied assume that the required debug status is stored in an environment variable *%jpivid%*. This should have one of the values *v2* (for full debug information) or *v0* (for no debug information). However, *cc.bat* and *make.bat* each call a subsidiary batch file, *checkvid.bat*, which ensures that *%jpivid%* does indeed exist and has one of these values.

The value of *%jpivid%* is itself expected to be set up by calling the final batch file in the suite: *vid.bat*. Typing

```
vid on
```

at the MS-DOS command line has the effect of setting *%jpivid%* to *v2*, whereas typing

```
vid off
```

sets *%jpivid%* to *v0*. Typing *vid* by itself echoes the current *vid* setting.

The contents of these four batch files are as follows:

```
(vid.bat)

@echo off
goto X%1X
:Xv0X
:XoffX
set jpivid=v0
echo VID is now OFF
goto :end
:Xv2X
:XonX
set jpivid=v2
echo VID is now ON
goto :end
:XX
call checkvid
goto %jpivid%
:v0
echo VID is OFF
goto end
:v2
echo VID is ON
:end

(checkvid.bat)

@if not "%jpivid%"=="v2" set jpivid=v0

(cc.bat:)

@echo off
call checkvid
if exist %1.pr goto custom
tsc %1.c /fpunnamed /%jpivid%
goto end
:custom
tsc %1.c /fp%1 /%jpivid%
:end

(make.bat:)

@echo off
call checkvid
if exist %1.pr goto custom
tsc /m unnamed.pr /smain=%1 /%jpivid%
goto end
:custom
tsc /m %1.pr /smain=%1 /%jpivid%
:end
```

Naturally, there is considerable scope for further personalisation and enhancement of these batch files, if desired.

One final refinement would be to use the MS-DOS prompt command to change the prompt to reflect the current value of *vid*. For example,

```
prompt $p_%jpivid%$g
```

(The reason it is generally important to keep track of whether full debugging information is being generated is that significantly larger *.img* programs can result in this case.)

TSC versus TSCX

All the examples of compiling and linking that have been given so far use the TopSpeed *tsc* command. This command does not make use of expanded memory and may cause problems, particularly when linking large applications.

Most of the batch files supplied with the SDK to compile, link or make executables use the *tsc* version. If this causes difficulties on your PC, you may find that replacing *tsc* with *tscx* (which uses expanded memory) in these batch files will cure the problem.

More complicated programming setups

The remainder of this chapter makes no mention of programming *technique* or programming *concepts* within the SIBO SDK system (see the later chapter *Fundamental Programming Guidelines* for that). Rather, it continues to explain more details of the mechanics of building applications of various sorts.

Simple PLIB examples

The following four programs all use the default project file, *unnamed.pr*, and each consist of only one source module:

<i>p_search</i>	searches a specified text file for a given piece of text
<i>p_prndir</i>	prints specified directory listings to the screen
<i>p_dlist</i>	lists all current "devices" (ie local and remote disk drives)
<i>p_comp</i>	compares two specified files, to see if they match.

For example, to build a version of *p_dlist.img* suitable for source-level debugging, just type

```
vid on
make p_dlist
```

To exit any of these programs which repeatedly request user input, simply press ENTER on an empty input line.

Note: these programs are all restricted to so-called *console i/o*:

- no attractive graphics
- limited support for the user editing data entered previously.

Additionally, they pay no attention to the actual size of the screen on individual SIBO computers. Whilst the data they display fits well enough on the large screen of MC computers, the display is less suited to the smaller screens of HC or Series3 computers. Simple modifications can make amends in this last regard. But for enhanced graphics output, use of Window Server functions is needed.

Finally, these programs are all *single-threaded*, ie each has only one event source (the keyboard). At the same time, they contain no asynchronous i/o. (The vital topics of *multi-threaded* programming and asynchronous i/o are two of the central themes of the chapter *Fundamental Programming Guidelines*.)

But despite their limitations, these four programs will hopefully be found useful for the purpose of acquiring familiarity with *.pr* project files and with the SIBO SDK system generally. There is no need to worry unduly over their detailed content; however, taking the time to build them and then improve them could well turn out a very rewarding exercise (eg in making the transition from CLIB to PLIB).

Multi-file programs

For a program with more than one source file, the corresponding *.pr* needs but a slight modification.

For example, suppose a program *triple* has three source files: *triple.c* itself, *utils1.c* and *utils2.c*. A suitable project file *triple.pr* would be

```
#system epoc img
#set epocinit=iplib
#model small jpi

#compile triple
#compile utils1
#compile utils2
#link triple
```

The effect of the final *#link* statement is actually as follows:

- link together all the files listed with *#compile* statements
- link also the relevant startup module and standard libraries

- link also any object files or libraries specified by any `#pragma link` statements (see below for examples)
- give the final executable the name specified in the `#link` statement.

Note in particular there is no need for the name specified by the `#link` statement to match the name of the `.pr` file, nor the name of any of the individual files linked together.

Graphics programs

Programs which interact directly with the Window Server can produce a large variety of impressive graphics effects - icons and bitmaps, shapes and areas, mixed fonts and styles, scrolling and animation, information messages and alerts, and so on.

These programs can operate with `.pr` files of exactly the same form as described earlier in this chapter. For example, the file *unnamed.pr* can continue to be used, unchanged, for any simple single-module Window Server program. Parts of the Window Server library, *wlib.lib*, are automatically linked in as required, without any change being required in the `.pr` file: there is no need to ask for *wlib.lib* explicitly.

The source modules will, of course, have to change in the following aspects:

- calls to Window Server functions `gxxx` or `wxxx` will be included
- the Window Server header file *wlib.h* will have to be `#included`.

For more details, including some introductory Window Server programs of the "Hello World" variety, see the *Window Server Reference* manual.

HWIF programs

Developers writing for the Series3 can take advantage of the additional menu and dialog functionality (amongst other features) of the HWIF library, to create applications very similar to those built into the rom of the Series3.

See the *Programming in HWIF* manual for full details, including a suite of example programs.

Project files for these programs need to include the line

```
#pragma link(hwif.lib)
```

since the HWIF library is not one of those that are *automatically* searched at link time.

Customised libraries

Developers may wish to collect various utility routines, or other subsets of code, into their own libraries, which can in due course be linked into different programs. Possibly, developers may wish to distribute their libraries in object form (*.lib* files), and not in source form. In such a case, there is likely to be at least two different `.pr` files: one controlling the creation of the *.lib* file, and one that, later, joins the *.lib* file into a required application program.

For example, suppose that modules *utils1.c* and *utils2.c* are to be compiled and linked into a library called *utils.lib*. This can be accomplished by means of the following project file:

```
#system epoc img
#set epocinit=iplib
#model small jpi
#compile utils1
#compile utils2
#dolink utils.lib
```

Note the following points:

- the command `#dolink` is used rather than `#link`, to stop the TopSpeed system attempting to link in a startup object too (not to mention other standard libraries)
- the extension `.lib` explicitly given overrides the default `.img` that would otherwise be assumed on account of the statement `#system epoc img`.

A project file to produce an application *tutils*, say, that made use of functionality in *utils.lib*, could then be as follows:

```
#system epoc img
#set epocinit=iplib
#model small jpi
#compile tutils
#pragma link(utils.lib)
#link tutils
```

with *utils.lib* being searched for along the path specified in the redirection file *ts.red*.

Applications containing assembler as well as C

In principle, it is perfectly possible for an application to include assembler source modules, as well as modules written in C.

For example, if an application contains two source files, *cfile.c* written in C and *afile.a* written in assembler, the following could appear in the project file:

```
#compile cfile
#compile afile
```

The TopSpeed system will automatically run the appropriate "compiler" for each specified type of file - ie compiling the *.c* file and assembling the *.a* file.

However, programmers should note that there are various rules that must be adhered to in writing assembler modules for Epoc programs. See the *Introduction* chapter of the *PLIB Reference* manual in the first instance.

Note: the TopSpeed system will give an error message, and terminate, if there are two possible files each candidates as the source for of a *#compile* statement - for example, if the files *cfile.a* and *cfile.c* both exist in a directory. As another example, if a directory contains files *query.c* and *query.rc*, the statement *#compile query* will again result in an error - since the TopSpeed system regards the *.rc* file as a possible source file too. All these cases can be circumvented by giving the extension explicitly in the *#command* statement - eg *#compile query.c*.

Other advanced use of the project file system

The TopSpeed documentation describes many possible ways to exercise further control over the process of building program files.

One general piece of advice should, however, be borne in mind: while learning to program within the SIBO SDK system, please accept the default configuration proposed by Psion. Only attempt to refine this configuration once your program is already clearly working. Otherwise, it may prove difficult to determine whether some unexpected program behaviour is due to a coding mistake, or to some unexpected side-effect of a proposed "optimisation" of the build configuration.

In any case, optimisations resulting from careful choices of PLIB or WLIB (etc) functions, are likely to prove more significant than any that can easily be achieved by tweaking the SIBO version of the TopSpeed build configuration.

It is also generally a bad idea to ignore warnings from the compiler and linker (except where explicitly mentioned in the SDK manuals). Rather than discounting these warnings as "quirks" of the system, they should all be analysed and dealt with. In particular, don't be too hasty to disable "inconvenient" compiler warnings.

Note that the SIBO SDK build configuration is actually defined in two different parts:

- in the file *tsprj.txt* which has to be "compiled" (using *tscfg*) before being used
- in the file *stdepoc.h* which is always the first include file in any source module.

As mentioned in the chapter on *Installation*, the file *tsprj.txt* released as part of the SIBO SDK modifies and extends the one released by Clarion themselves, by adding in details specific to the SIBO SDK system.

Greater control over the image file created

This section explains some of the SIBO add-ons to the TopSpeed project file build system. It also covers standalone use of the tools *edump.exe*, *emake.exe* and *eremake.exe*:

- *edump.exe* provides key information about the contents of an image file
- *emake.exe* is the underlying tool which creates image files
- *eremake.exe* can be used to alter some of the "additional" contents of image files.

Priority, minimum heap, and version number

Three project file variables can be used to override various defaults otherwise used when a *.img* file is created:

%version	sets the version number of the <i>.img</i> file, which otherwise defaults to 0x100f
%priority	sets the initial priority of the program, which otherwise defaults to 0x80
%heapsize	sets the initial and minimum heap of the application, which otherwise defaults to 0x80.

The application version number can be read by various pieces of software, for example the Application Info command of the System application on MC computers. For more about legal version numbers, see the section on *p_version* in the *PLIB Reference* manual.

The initial process priority may occasionally need to be specified explicitly, eg for an application that is part of a suite of cooperating applications. See the section on *p_setpri* in the *PLIB Reference* manual and also the section *Priority changing* in the chapter *General Window Server Functions* in the *Window Server Reference* manual.

The value of %heapsize is the one that applications are most likely to wish to alter, since this has significance for error handling (see the chapter *Fundamental Programming Guidelines* later in this manual):

- the value of %heapsize is in paragraphs, eg 0x80 means 0x800 bytes ie 2 Kbytes
- the operating system will refuse to start an instance of the application if this amount of free heap space cannot be found for it
- once started, the application will never have its heap shrunk below this value.

The default values for these variables can be overridden in either of two ways:

- a line such as *#set heapsize=0x180* can be added into the project file
- the parameter */sheapsize=0x180* can be added to the end of the *tsc* command invoking the project file (this is evidently the same syntax as in the */smain=%1* in the supplied batch file *make.bat*).

Use of edump.exe

The tool *edump.exe* can be used to verify the values of the above three variables (amongst others) for a specified *.img* file. For example, typing *edump p_hello* results in output such as

```
EDump V2.02F (03/10/90) Copyright (C) Psion PLC 1989
LOC::D:\SIBOSDK\DEMO\P_HELLO.IMG IMAGE file data
Image version      = 200F
Code Segment       = 01D0 (bytes)
Initial IP         = 0000
Stack              = 1000 (bytes)
Data               = 0040 (bytes)
Heap               = 0800 (bytes)
Data Segment       = 1840 (bytes)
Initialized data    = 0030 (bytes)
Code checksum       = DA14
Data checksum       = 0BF7
Code Version       = 100F
Priority            = 0080
Header size        = 0040 (bytes)
Dyl count          = 0000
Dyl table offset    = 00000000
Image file size     = 00000240 (bytes)
```

where the default values of "Heap", "Code version", and "Priority" can all be seen.

Deleting *p_hello.img* and rebuilding it via the command

```
tsc /m p_hello /sversion=0x110b
```

before running *edump* again yields identical output, except that the "Code version" line changes.

Differences between .app files and .img files

Strictly speaking, there is no real difference between image files with extension *.img* and those with extension *.app*. For example, although the System applications on the MC and on the Series3 usually expect to install *.app* files, they will also, if requested, install suitable *.img* files.

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

- a *.pic* file providing the *icon* for the application
- a *.rsc* or *.rzc* file providing the *resource file* for the application
- a *.shd* file providing the *shell data* for the application (only for Series3 applications).

As such, a *.app* file is simply a *.img* file which has some associated files conveniently built into it. A significant advantage of a *.app* file is that a user cannot inadvertently sabotage the operation of the program by copying the *.img* file itself from one drive to another, but neglecting to copy one of the associated files.

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

Add-file lists

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

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

When any *.pr* project file is invoked that leads to the building of *tele.img*, the existence of a file *tele.afl* is checked for. If such a file is found, the files listed therein are combined with the core *.img* file to form a larger *.img* file as output. By convention, *.img* files that contain embedded add-files are renamed to *.app* files (though no such renaming takes place automatically).

The inclusion of these embedded files can be confirmed by running the tool *edump.exe* as follows. Typing *edump tele.app* might yield

```
EDump V2.02F (03/10/90) Copyright (C) Psion PLC 1989
LOC::E:\SIBOSDK\HWDEMO\TELE.APP IMAGE file data
Image version      = 200F
Code Segment       = 1ED0 (bytes)
Initial IP         = 0000
Stack              = 1000 (bytes)
Data               = 0690 (bytes)
Heap               = 0800 (bytes)
Data Segment       = 1E90 (bytes)
Initialized data    = 03C0 (bytes)
Code checksum       = 33DF
Data checksum       = 3033
Code Version        = 100F
Priority            = 0080
Header size         = 00F0 (bytes)
Add 1 offset, len   = 0040 (bytes), 0074 (bytes)
Add 2 offset, len   = 00C0 (bytes), 0000 (bytes)
Add 3 offset, len   = 00C0 (bytes), 002E (bytes)
Dyl count           = 0000
Dyl table offset    = 00000000
Image file size     = 00002380 (bytes)
```

For the moment, the interesting data here is contained in the three lines of the form:

```
Add n offset, len
```

These give the offsets within the combined *.app* file to the embedded add-files. In this case, three of the add-file slots are used; in general, any number from zero to four could be used.

For more discussion about various possible add-files, see the *Series 3/3a Programming Guide*.

A similar technique, using a DYL file list in a file with a *.dfl* extension, may be used to build any number of dynamic library (DYL) files into an application. This topic is described in more detail in the *Object Oriented Programming Guide*.

Changing the set of add-files in an image

The tool *eremake.exe* can be used to change the set of add-files built into an image file, without needing to run the TopSpeed project system again, and without needing to have any files to hand apart from the original image file. That is, image files can be *remade* with new add-file contents, without the earlier *.exe*, *.obj*, or *.c* files being present.

One common use of *eremake* is to convert eg an English language version of an application into a specified alternative language version. See the chapter *Resource Files* in the *Additional System Information* manual for a general discussion of applications that can run in more than one language.

For example, suppose that an application *Query* has all its language text isolated in a resource file *query.rzc*, which is one of the original add-files for the application. More precisely, suppose that the original contents of *query.afl* are

```
query.pic
query.rzc
query.shd
```

Suppose further that a French version of the resource file is produced: *frquery.rzc*, say. Then a new *.afl* list should be created, *frquery.afl* say:

```
query.pic
frquery.rzc
query.shd
```

and *eremake.exe* should be invoked as follows:

```
eremake -afrquery -o..\french\query.app query.app
```

For a full list of possible parameters to *eremake.exe*, simply type *eremake* by itself. Note that *eremake.exe* can be used to alter the priority, minimum heap, or version number of an image file.

In the above case, the "output" file *..\french\query.app* is created by "remaking" *query.app* with add-files listed in *frquery.afl*.

In this example, the files *query.pic* and *query.shd* from the original version are also used for the French version. Clearly, these could be changed too, if required.

Further uses of emake.exe

The actual conversion from *.exe* form into *.img* form is handled by the Psion-proprietary tool *emake.exe*, according to the command

```
#run "emake -b %afl% -o%name% -s -v%version% -p%priority% -h%heapsize% -%epoctype%
%name%.exe"
```

in the file *tsprj.txt*.

For a full list of possible parameters to *emake.exe*, just type *emake* by itself.

Note in particular that *emake* can produce other forms of final output, apart from *.img* files. This is determined by the *%epoctype* variable in the above command, which by default takes the value *t1*. Other possible values are *t2* through *t4*:

- t1 makes an *image file*, with characteristic extension *.img*
- t2 makes a *logical device driver*, with characteristic extension *.ldd*
- t3 makes a *physical device driver*, with characteristic extension *.pdd*
- t4 makes a *dynamic library*, with characteristic extension *.dyl*.

Logical and physical device drivers are further discussed in the *Writing Device Drivers* chapter in the *Additional System Information* manual. Dynamic libraries are discussed in the *Object Oriented Programming* chapter of the *PLIB Reference* manual, and in the *Object Oriented Programming Guide*.

CHAPTER 4

NOTES ON CLIB

This chapter provides some additional information about the implementation of CLIB, the version of the TopSpeed C library for the Epoc operating system. It need be read only by developers who wish to use CLIB (for example, to port existing code from another program). Developers who stick to PLIB can skip this chapter entirely.

The difference between PLIB and CLIB is explained in the previous chapter.

Functions missing from the MS-DOS Jpi C library

The following functions are not included in the EPOC version of CLIB because they rely on the IBM PC hardware, far or huge pointers, or direct MS-DOS functions calls. For many of the missing function calls there is an equivalent function or set of functions which may be called from either PLIB or WLIB.

absread	abswrite	allocmem
_arc	at	Awaited
bdos	bdosptr	bioscom
biosdisk	biosequip	bioskey
biosmemory	biosprint	biostime
_bios_disk	_bios_equiplist	_bios_keybrd
_bios_memsizes	_bios_printer	_bios_serialcom
_bios_timeofday	_chain_intr	change
_clearscreen	convertcoords	country
ctrlbrk	_cube	Delay
delay	_directwrite	_displaycursor
_dosbeginthread	_dosendthread	dosexterr
_dosfreestack	_dos_allocmem	_dos_close
_dos_creat	_dos_creatnew	_dos_findfirst
_dos_findnext	_dos_freemem	_dos_getdate
_dos_getdiskfree	_dos_getdrive	_dos_getfileattr
_dos_getftime	_dos_gettime	_dos_getvect
_dos_keep	_dos_open	_dos_read
_dos_setblock	_dos_setdate	_dos_setdrive
_dos_setfileattr	_dos_setftime	_dos_settime
_dos_setvect	_dos_write	_ellipse
_expand	execle	execlpe
execve	execvpe	farcalloc
farcoreleft	farfree	farmalloc
farrealloc	fcalloc	_fexpand
_ffree	_fheapchk	_fheapset
_fheapwalk	_floodfill	_fmalloc
_fmsize	_FP_OFF	_FP_SEG
_frealloc	freemem	geninterrupt
_getbkcolor	getcbrk	_getcolor
getcurdir	_getcurrentposition	getdfree
getdta	getfat	getfatd
_getfillmask	_getimage	_getlinestyle
_getlogcoord	_getphyscoord	_getpixel
getpsp	gettime	gettext

<code>_gettextcolor</code>	<code>gettextinfo</code>	<code>_gettextposition</code>
<code>getvect</code>	<code>getverify</code>	<code>_getvideoconfig</code>
<code>halloc</code>	<code>_harderr</code>	<code>_hardresume</code>
<code>_hardretn</code>	<code>hfree</code>	<code>hide</code>
<code>highvideo</code>	<code>hmemccpy</code>	<code>hmemchr</code>
<code>hmemcmp</code>	<code>hmemcpy</code>	<code>hmemicmp</code>
<code>hmemset</code>	<code>hrealloc</code>	<code>info</code>
<code>Init</code>	<code>intdos</code>	<code>intdosx</code>
<code>ioctl</code>	<code>keep</code>	<code>_lineto</code>
<code>Lock</code>	<code>locking</code>	<code>lowvideo</code>
<code>mktemp</code>	<code>MK_FP</code>	<code>movedata</code>
<code>movetext</code>	<code>_moveto</code>	<code>ncalloc_</code>
<code>_ms_cursor</code>	<code>_ms_driversize</code>	<code>_ms_getmotion</code>
<code>_ms_getpage</code>	<code>_ms_getpress</code>	<code>_ms_getrelease</code>
<code>_ms_getsensitivity</code>	<code>_ms_getstatus</code>	<code>_ms_lightpen</code>
<code>_ms_reset</code>	<code>_ms_restoredriver</code>	<code>_ms_savedriver</code>
<code>_ms_setdouble</code>	<code>_ms_setgraphcursor</code>	<code>_ms_setinterrupt</code>
<code>_ms_setmickeys</code>	<code>_ms_setpage</code>	<code>_ms_setposition</code>
<code>_ms_setrange</code>	<code>_ms_setsensitivity</code>	<code>_ms_settextcursor</code>
<code>_ms_swapinterrupt</code>	<code>_ms_updatescreen</code>	<code>normvideo</code>
<code>nosound</code>	<code>Notify</code>	<code>obsucredat</code>
<code>_outtext</code>	<code>palettecolor</code>	<code>palettecolorused</code>
<code>paletteopen</code>	<code>parsfnm</code>	<code>peek</code>
<code>peekb</code>	<code>_pie</code>	<code>poke</code>
<code>pokeb</code>	<code>_polygon</code>	<code>putbeneath</code>
<code>_putimage</code>	<code>putontop</code>	<code>puttext</code>
<code>randbrd</code>	<code>randbwr</code>	<code>readbufferln</code>
<code>_rectangle</code>	<code>_remappallpalette</code>	<code>_remappalette</code>
<code>_selectpalette</code>	<code>SEND</code>	<code>_setactivepage</code>
<code>_stbkcolor</code>	<code>setblock</code>	<code>setcbkr</code>
<code>_setcliprgn</code>	<code>_setcolor</code>	<code>setdta</code>
<code>setftime</code>	<code>_setfillmask</code>	<code>setframe</code>
<code>_setlinestyle</code>	<code>setlogorg</code>	<code>setpalette</code>
<code>setpalettecolor</code>	<code>_setpixel</code>	<code>_settextcolor</code>
<code>_settextposition</code>	<code>_settextwindow</code>	<code>settitle</code>
<code>setvect</code>	<code>setverify</code>	<code>_setvideomode</code>
<code>_setviewport</code>	<code>_setvsualpage</code>	<code>snapshot</code>
<code>sound</code>	<code>spawnle</code>	<code>spawnlpe</code>
<code>spawnve</code>	<code>spawnvpe</code>	<code>StartProcess</code>
<code>StartScheduler</code>	<code>StopProcess</code>	<code>StopScheduler</code>
<code>tempnam</code>	<code>textattr</code>	<code>textbackground</code>
<code>textcolor</code>	<code>textmode</code>	<code>tmpnam</code>
<code>top</code>	<code>Unlock</code>	<code>use</code>
<code>used</code>	<code>WAIT</code>	<code>window</code>
<code>windowclose</code>	<code>windowopen</code>	<code>_wrapon</code>
<code>_wrbufferln</code>		

int86x

int86x is implemented by exactly the same code as int86, ie the values in the segment registers struct are ignored. This is because in the pure small model DS=ES=SS and hence there is never any need to change the values in the segment registers.

intr

As for int86x above, the values in the segment registers are ignored.

File handle conversion routine

```
void *getRealHandle(int handle);
```

Returns a PLIB file handle, given a CLIB file handle. Returns NULL if the handle is not currently allocated.

The console channel

Programs built with the CLIB start-up module automatically open a console channel. The channel is also automatically assigned to `stdin`, `stdout` and `stderr`, unless they have been redirected.

It may prove useful to get the actual PLIB handle for the console channel. This can be achieved by either calling `getRealHandle(fileno(stderr))` or by referencing the global `_winHandle` as follows:

```
extern void *winHandle;
```

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

```
extern void *winHandle;

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

int main(void)
{
    ...

    return(0);
}
```

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

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

MS-DOS file names

Applications written for the Epoc O/S should in general avoid making any assumptions about file names (see the *Files* chapter in the *Plib Reference* manual) and should use `p_fparse` and `p_chdir` to manipulate file names and navigate directory paths. However the `Loc::` file system on all SIBO machines is totally MS-DOS compatible. Thus CLIB supports the TopSpeed STD C library functions `findfirst`, `findnext`, `getcwd`, etc which rely on MS-DOS naming conventions. However if an attempt is made to use these functions on other filing systems, such as `REM::`, the routines will return an error.

Floating point emulator

As the SIBO architecture does not allow for an 8087 maths coprocessor, all floating point is performed by software emulation of the 8087.

Normally the emulator code would be linked in with your program for MS-DOS exe's, but under Epoc O/S the emulator is provided by an LDD called `SY$8087.LDD` in `\sibosdk\lib\`. Any programs requiring the emulator will automatically load the LDD and free it again when it is no longer required. This has the advantage of saving about 8K of code from your program and allows the LDD to be shared by multiple processes.

The C startup module (see `r_emul.a`) will look for the LDD in the directory in which your program was executed. If it is not found the it will use the environment variable `EM$` (remember to use capitals for the name as environment variables in Epoc O/S are case sensitive).

Panic 80

An application which terminates with a "panic 80" before it even starts has almost certainly failed to locate `SY$8087.LDD`. There are three steps that can be taken to avoid the panic:

- move a copy of the LDD into the directory where the program will execute from
- set the environment variable `EM$` appropriately (eg using a short program)
- rewrite the program so that it does not require the use of the LDD.

See the *Floating Point* chapter of the *Plib Reference* manual for some more details.

Note that floating point instructions can easily be generated unexpectedly (eg to push or pop floating point registers) if the recommended build configuration pragmas are "improved" in any way - resulting in panic 80s "out of the blue". If in doubt, inspect the object code using the SIBO Debugger (or use the TopSpeed disassembler, *tsda*).

Building the CLIB library and header objects

To build the CLIB library you must have purchased the Professional variant of the SIBO SDK. This version includes the C Library Source Kit and the corresponding EPOC CLIB Library Source. Since some of the EPOC CLIB source modules are written in C++, you will also need to have separately purchased TopSpeed C++.

Simply startup *TS* in the *SRC* directory of *SIBOSDK*. Select CLIB as the project file and then MAKE the project. The libraries and objects will be generated in the *SRC* directory and need to be copied into the *LIB* directory. There is a batch file *INST.BAT* which will do this for you.

Note that, in addition to the CLIB library, there is also an RLIB component. This contains elements that are common between CLIB and PLIB.

The source code modules for PLIB and WLIB are currently in Turbo Assembler format and are not buildable using the TopSpeed assembler. They have not been included in this release of the SDK. The vast majority in any case are simple shells which just juggle registers.

Cautionary note

It is advisable to tread cautiously when changing the library, and under no circumstances should the pragmas be changed in either *TSPRJ.TXT* or *STDEPOC.H*.

CHAPTER 5

FUNDAMENTAL PROGRAMMING GUIDE-LINES

Introduction

In some ways, this chapter may be viewed as being among the most important in the whole of the SDK.

Follow the Guide-lines here and your programs have a good chance of possessing the following qualities:

<i>user responsiveness</i>	users will not be kept waiting impatiently if they want to interact with an application whilst it is busy - eg to cancel some operation part-way completed
<i>error robustness</i>	data will not suddenly be lost when run-time errors occur such as shortage of system memory (bear in mind that such errors are almost inevitable on a multi-tasking computer, when the system memory can become unexpectedly used up by other applications)
<i>architectural robustness</i>	changes in user requirements or in implementation tactics should not lead to the whole code becoming unmaintainable.

Of course, practice of *standard* general programming principles - such as modular programming, data hiding, egoless programming, designing prior to coding (not to mention adequate requirements specification prior to design), and a structured approach to validation and testing - all have important roles to play in the production of quality SIBO applications. But there are *additional* programming principles that have particular importance within the SIBO environment, and it is these that this chapter addresses. These principles should complement the ones good programmers from other backgrounds already practice.

Incidentally, just as the merits of the above-mentioned "standard" programming principles are not always immediately obvious (data hiding is a good example), but rather have to be learned, so it is with some of the principles outlined in this chapter. Their importance has become clear to the programming team at Psion only gradually, over several years' experience. It is understandable that experienced programmers from other backgrounds may wish to rush over this chapter, believing its contents to be inapplicable to them, but that would almost certainly be a mistake.

Again, there is of course no substitute for a wide-ranging knowledge of which library functions are available. Developers wishing to produce quality applications will naturally have to spend some considerable time familiarising themselves with the contents of the reference manuals within the SDK, so as to be able to spot the right function to use in any particular coding situation. This knowledge cannot be acquired simply by assenting to the set of programming principles covered in this chapter. But conversely, wide knowledge of the set of available function calls is insufficient, by itself, to produce programs with the traits listed at the start of this chapter.

Other related documentation

Applications which are *multi-lingual* (eg which present English language messages on an English language computer, French language messages on a French language computer, and so on) pose their own set of programming problems. These are discussed in the course of the *Resource Files* chapter of the *Additional System Information* manual. (Resource files are a tool of particular importance for multi-lingual applications.) Note that these problems are shared between applications which are *actually* multi-lingual and those that are *potentially* multi-lingual: it is better to design support for multi-linguality in from the start, than trying to add it on afterwards.

Graphics programming has its own particular set of do's and don'ts, in order that (for example) flicker-free redrawing and automated screen update take place. These are discussed at various places in the SDK, for example, in the *Window Server Reference* manual, and also in the *Programming in HWIF* manual.

Avoidance of excessive RAM usage, and also of "stack windup", are also of considerable importance within the SIBO architecture. Specific advice on these regards are scattered throughout the SDK; the present brief note simply has the purpose of drawing attention to the topic.

Finally, the task of *designing the user interface* (menus, dialogs, and so on) is another that poses its own special problems - programmers who labour under the misapprehension that user interfaces are "easy" to design almost invariably produce poor user interfaces. Guide-lines on these matters may be found within the object oriented documentation parts of the SDK.

The source code for the examples

The bulk of this chapter consists of a lengthy analysis of a suite of example programs. These build in four stages - *Events*, *Events2*, *Events3*, and *Events4* - to a program that can simultaneously process:

- keyboard input
- the expiry of a timer
- reports of the completion of a sub-process
- data received from a serial port

whilst all the time carrying on some compute-bound activity "in background"

At any given moment, the program cannot predict which of its five possible "event sources" will be the next to require CPU. As such, the program amply demonstrates the three vital themes of *multi-threadedness*, *asynchronous i/o*, and *yielding CPU*:

- *multi-threadedness* means that more than one set of activity takes place simultaneously within the program, with each different flow of activity being handled by its own "thread" of code
- *asynchronous i/o* means that the *queuing of a read request* (or write request) on an i/o channel is separated in code from the *completion* of that request
- *yielding CPU* means that lengthy calculations are broken down into subcomponents that are executed separately, with gaps in between so that some more urgent event source can be attended to, if necessary.

Although the program itself has limited practical use, it demonstrates the *basic architectural principles* that all sophisticated Epoc programs are bound to have to consider.

The source code for these four stages of *Events* can be installed from disc (SIBOSDK\DEMO), together with that of an associated "sub-process" application, *Subproc*. Readers are urged to take the time to build these applications and to experiment with them, in particular trying out some of the suggestions made in this chapter for how these programs could be modified.

User interface

The user interface of these example programs has deliberately been kept spartan. All screen drawing is to a "console terminal" that is 25 characters wide and 9 characters deep.

These decisions have the advantage that:

- the program works equally well on all different SIBO computers (even on the HC, where the screen is smallest)
- considerations about interfacing with the Window Server - necessary in order to achieve more graphically appealing displays - can be postponed while focusing instead on multi-threadedness, asynchronous i/o, and yielding CPU.

Various examples of Window Server programs demonstrating multi-threadedness (et al) can be found within the SDK. For example, the *Writing Software for the HC* chapter of the *HC Programming Guide* discusses an application called *Gauge*, that will in fact run happily (with only minor adjustments) on a Series 3. Again, the *Programming in HWIF* manual contains a large set of example programs, all of which interface with the Window Server.

A first look at multiple event sources

The main routine in *Events* is as follows (the individual calls made, including the Plib calls, are discussed below):

```
GLDEF_C VOID main(VOID)
{
    OpenConsole();
    DrawBorder();
    OpenTimer();
    timint=10;
    QueueTimer();
    QueueKey();
    FOREVER
    {
        p_iowait();
        if (keystat!=E_FILE_PENDING)
        {
            if (key.keycode==W_KEY_ESCAPE)
                p_exit(0);
            if (key.keycode>='1' && key.keycode<='9')
                SetTimInt(2*(key.keycode-'0'));
            QueueKey();
        }
        else if (timstat!=E_FILE_PENDING)
        {
            if (counter++==MAX_COUNT)
                counter=1;
            DisplayCount();
            QueueTimer();
        }
    }
}
```

Schematically, the code is as follows:

```
GLDEF_C VOID main(VOID)
{
    INITIALISE();
    QueueTimer();
    QueueKey();
    FOREVER
    {
        p_iowait();
        if (keystat!=E_FILE_PENDING)
        {
            PROCESS_KEY();
            QueueKey();
        }
        else if (timstat!=E_FILE_PENDING)
        {
            PROCESS_TIMER();
            QueueTimer();
        }
    }
}
```

in which it is clear that the program has two event sources - keypresses and the expiry of a timer.

What the program actually does is to update a numeric count (displayed on the screen) regularly, on a timer. The rate at which the timer fires is determined by which keys the user presses:

- it starts off firing once every second
- if the user presses the 2 key, the timer changes to firing once every 2/5 of a second (so that the numbers tick over more rapidly)
- if the user presses the 9 key, the timer changes to firing once every 9/5 of a second (so that the numbers tick over more slowly)

and so on. Further, every time a numeric key is pressed, the existing timer request is cancelled, and the timer reset - so that pressing repeatedly on keys such as 8 and 9 can have the effect of "stalling" the counter altogether.

The program exits in response to the ESC key being pressed. All other keys are ignored.

Remarks on timers

One simple approach to programming with delays is to call a function such as `p_sleep`, which effectively suspends the application for a specified amount of time.

This approach could be adopted in *Events*, were it not for the fact that, when the application is suspended, it cannot respond to a keypress. The keypress will only be received when the application "wakens up" again.

Now this might not be too much of a loss for very small time delays, but it is of course unacceptable for longer delays. For example, a program might wish to perform some housekeeping or maintenance once every twenty four hours - or simply update the display in a dialog once every two seconds, whilst allowing the user to cancel out of the dialog at any time. That is, whilst routines such as `p_sleep` certainly have a role to play, they cannot handle *all* timer requirements in programs.

A next possible approach would be to design a routine which, when called, suspended the application until the specified time elapsed *or* a keypress is received - whichever happens first. Indeed, there is a call with just this specification in the Opl programming language (pause when used with a *negative* time delay).

Actually, this routine would satisfy the requirements of *Events* perfectly. However, it has the severe drawback of *lack of architectural openness*. That is, suppose the application has to be modified at a later date, to be able to respond to another sort of event source - eg the arrival of data at a serial port, or an interprocess message from another application. Alternatively, the application may need to carry on some continuous activity, whilst waiting for the timer to expire. In either case, a more general approach is required.

This more general approach is the mechanism by which the *queuing* of a timer is separated, in code, from the *completion* of the timer. In the above code, the timer is queued by the call `QueueTimer`, whereas the completion of the timer occurs within the `p_iowait` call. (See later for the details.)

This kind of code separation between the queuing of a request and the completion of the request is known as *asynchronous* - because there is no automatic synchronisation of the two phases (as occurs, for example, in a call such as `p_sleep`).

Remarks on keypresses

A routine such as `p_getch` is to keypresses what `p_sleep` is to timers - in both cases, the program is effectively suspended until the request is completely satisfied.

Just as there is a place for `p_sleep` in programming, so also there is a place for `p_getch`. However, each is generally inappropriate when there is more than one event source current - as here.

So the call `p_getch` is split into two parts: the *request* for a keypress to be delivered, inside the call `QueueKey`, and the delivery of the keypress - inside `p_iowait`.

More on p_iowait

The call `p_iowait` is where the application gets suspended, while it waits for the completion of some or other event. But whilst `p_sleep` only returns when the associated timer expires, and `p_getch` only returns when a keypress is delivered, `p_iowait` returns when *any* known event has completed.

Of course, sometimes the application won't get suspended at all when it calls `p_iowait` - on account of a queued request already having completed. For example, during the time *Events* is inside the code `PROCESS_TIMER`, the user may have pressed a key, in which case the subsequent call to `p_iowait` will return immediately (not that the application needs to worry about this, however).

The way `p_iowait` works is by consulting the value of the *i/o semaphore* for the process. Each process has an i/o semaphore assigned to it. These semaphores are maintained in the private data space of the operating system.

An i/o semaphore commonly has its value changed in either of two ways:

- it is *decremented* whenever a call to `p_iowait` is made
- it is *incremented* whenever some piece of software "signals" the application.

For example, whenever a timer expires, the operating system timer device signals the application which owns the timer. Again, whenever the Window Server delivers a keypress to an application, it signals that application. In general, the completion of any i/o request always involves the application being signalled.

The i/o semaphore of an application initially has the value zero. A call to `p_iowait` only returns when the i/o semaphore is non-negative. Given that the call to `p_iowait` starts by decrementing the semaphore, the call will return only when "something has happened".

For more details about `p_iowait` and i/o semaphores in general, see the *Plib Reference* manual.

Note that the SIBO Debugger has a menu command (Process Status) which displays the value of the i/o semaphore of a process (amongst other information). The Spy application discussed in the *Series 3 Programming Guide* has a similar feature, which allows the i/o semaphores of all different running processes to be viewed simultaneously.

Status words: the other side of `p_iowait`

When a program returns from a call to `p_sleep`, it knows that it is a timer that has expired. Likewise, when a program returns from a call to `p_getch`, it knows that a keypress has been delivered. However, when a call to `p_iowait` returns, no such information is immediately available. All the application knows at this point is that *some* event has completed - not *which* event.

In general, the task of determining which events are indeed ready to be processed involves two factors:

- knowing which event sources are *active* - having had requests made on them
- knowing which of this subset have completed their requests.

For the moment, the first of these points can be ignored (but see later), with attention given to the notion of the *status word* of an asynchronous request:

- when the request is made, an address of a status word is specified (recall that a "word" is two adjacent bytes)
- when the request is made, the value `E_FILE_PENDING` gets written to this word
- when the request completes, a value other than `E_FILE_PENDING` is written to this word (the actual value varying depending on the type of asynchronous request)

Writing to the status word takes place just before the i/o semaphore for the application is signalled. Both these steps are integral to the SIBO mechanism of `p_iowait`.

All this explains the code in *Events* immediately after the call to `p_iowait`: the different status words are *polled* in turn, in order to find one whose value is no longer `E_FILE_PENDING`.

Prioritisation of event sources

Note that the order in which the status words are polled implicitly *prioritises* the different event sources. For it is possible for more than one asynchronous event to have completed; in this case, the first one of the two polled will be the one which gets the first response.

Note in particular that the order in which events are processed need bear no direct relation to the order in which the events actually completed. An event source lower down the priority listing can be "locked out" by rapidly firing event sources higher in the listing. In fact, code can often be written which *depends* on this prioritisation - so that a lower priority event source is only serviced when all higher priority event sources have quietened down.

I/O devices in general

In this example, keypresses are delivered by the *console device*. Likewise, timer expiry is handled by the *timer device*. The console device and the timer device are both instances of general so-called *i/o devices*, all of which possess a common interface:

- before using an i/o device, an application has to *open a channel* to it
- once opened, various services can be requested via the channel
- these services can all in principle be requested either *synchronously* or *asynchronously*
- if an i/o device channel is no longer needed, the resources it consumes (eg memory) can be released by *closing* the channel.

Channels to i/o devices are opened by means of the `p_open` call, in which the i/o device has to be specified by name. Another parameter to the `p_open` call also defines where the *handle* of the channel will be written.

The handle returned by `p_open` can then be used to request other services from the i/o device. It can also be passed as a parameter to `p_close`, to close the channel down again.

The way *synchronous* requests are made, via an i/o channel, is to use a `p_iow` call (or a convenience routine that layers over this). *Asynchronous* requests are made using a `p_ioa` (or `p_ioc`) call.

For example, the call to position the console cursor, inside the routine `write` in `events.c`, is as follows:

```
P_POINT pos;
WORD func;

pos.x=x;
pos.y=y;
func=P_SCR_POSA;
p_iow4(conH,P_FSET,&func,&pos);
```

This is a synchronous call because there is no point in calling it asynchronously: the effect of this call (setting the cursor position) is always (virtually) immediate: there is no scope for any extended delay as it is carried out.

On the other hand, the code for `QueueKey` is

```
LOCAL_C VOID QueueKey(VOID)
{
    p_ioa4(conH,P_FREAD,&keystat,&key);
}
```

which is clearly asynchronous. (The 'a' in "`p_ioa`" stands for *asynchronous*.)

By chance, it turns out that these two calls to `p_ioa` and `p_iow` have the same number of parameters. This, however, is only the case because the `P_FREAD` request requires one less parameter than the `P_FSET` request. In general, an asynchronous request always has one more parameter than the corresponding synchronous request - namely the status word, whose address has to be passed in the asynchronous case.

For more on the `p_io?` functions (including the significance of the numeric suffices to the function names), see the *Plib Reference* manual.

The console device

The *Events* application uses the console device - as opposed to just making calls like `p_printf` - for two reasons:

- as discussed already, the console device supports an asynchronous version of `p_getch`
- the console device supports repositioning of the cursor position - which fact is relied upon heavily in the application.

For a full specification of the console device, see the corresponding chapter in the *I/O Devices Reference* manual.

How to cancel a timer

The code inside `CancelTimer` merits some attention:

```
LOCAL_C VOID CancelTimer(VOID)
{
    p_iow2(timH,P_FCANCEL);
    p_waitstat(&timstat);
}
```

First, the `P_FCANCEL` service is requested. This is itself a synchronous request, completing at once.

However, the way the generic service `P_FCANCEL` works in device drivers is never to *retract* the earlier request, but rather to *precipitate its completion*. It cannot retract it in general because it may already have completed by the time the `P_FCANCEL` request is made:

- if the request has already completed, the `P_FCANCEL` does precisely nothing
- if the request is still outstanding, it is completed forthwith, with the value `E_FILE_CANCEL` being written to the status word, and with the application being signalled (on account of the fact that the earlier request has now completed, albeit precipitously).

In either case, the application will be signalled - either before the `P_FCANCEL`, or after it. Accordingly, this signal has to be "processed" (or "used up"). This is the purpose of the subsequent call to `p_waitstat`. For more details about `p_waitstat`, see the *Plib Reference* manual.

Omitting to call `p_waitstat` after making a `P_FCANCEL` request is a common error. The result of this error is that the next call to `p_iowait` will return at once, even though none of the remaining active event sources is ready to deliver an event.

For more details about the timer device in general, see the chapter *Time, Timers and Dates* in the *Plib Reference* manual.

Where to declare status words

On the subject of common programming errors in conjunction with asynchronous i/o, perhaps the most common one has yet to be mentioned. This is the mistake of declaring status words on the stack of some routine which will have returned long before the asynchronous request completes. The status word will now be a piece of random data - perhaps on the stack of another routine - and random damage can ensue when it is in due course written to.

Status words should always be declared either in static data (as in *Events*), or in a control block allocated from the heap.

A first look at error handling

Every time a function call is written into a SIBO program, the programmer should consider the question: could a run-time error occur in the middle of this function? And if so, what would happen?

Chief amongst these possible errors is lack of memory. This can occur in *three* different ways:

- the application has reached the limit of its 64k data segment
- the application is using less than 64k itself, but there is no system memory available for it to be given more heap space
- another program with which the application is cooperating runs out of memory.

Other errors that need to be considered include:

- resources not being available because another application is already using them (eg serial port or sound device driver, or even a file that is currently open by another application)
- disk-based errors such as disk full, disk corrupt, or disk removed
- the unexpected disappearance of the remote filing system (*rem::*)
- comms failures such as serial overrun, parity error, or line failure.

These errors cannot be dismissed with the philosophy that, in an ideal world, they will not happen. Instead, they can and will happen, despite the best endeavours of the programmer. SSDs becoming full up, comms cables being removed, or a file already being open by another program, are all problems that arise naturally in the operation of a SIBO computer.

Whatever the cause of a run-time error, applications should take every care that no data entered by the user is lost. Another requirement - to avoid parts of memory being permanently tied up for no purpose - is that partly assembled data constructs which cannot be completely assembled, owing to a run-time error at a later stage, should be carefully disassembled again.

Error handling in Events

Looking at the source code in *events.c*, it is clear that three different types of error are considered:

- failure to open the timer device
- failure to open the console device
- failure to set the size of the console screen.

None of the other function calls have any possibility of run-time error, as can be verified by considering them all individually. (In fact, considering every call individually for possibilities of run-time error has to be the *norm* when developing applications.)

It is worth considering the above three possible errors in a little more detail. For example, opening a timer can fail for two reasons:

- lack of memory for the timer control block in the application data space
- lack of memory in the operating system dataspace for the real timer entry.

Now whilst an application may be able to ensure that the first possibility never arises - by means of setting its minimum heap appropriately - it can never be sure of preventing the second case. The number of timers allocated in operating system dataspace depends not on circumstances *within* the original application, but rather on what *other* applications have done. If other applications running simultaneously happen to make heavy use of timer resources, the operating system may not be able to set aside the one timer channel requested by *Events*.

Should this occur, the code in `openTimer` and `check` ensure that a suitable error message is passed back to the user. The user can then take action to shut down some of the other applications running simultaneously on the computer, before trying to start *Events* again.

The case of the channel to the console device is similar. This time, it is resources of (for example) the Window Server which may be unable to meet the request made. Likewise when the console window is sized (at which time various arrays or back-up bitmaps need to be allocated).

Fatal errors and non-fatal errors

The three possible errors in *Events* are all treated as being *fatal*: the application initialisation fails, so the application terminates.

This kind of action makes good sense for errors during the initialisation of an application, but cannot on the whole be tolerated for errors during the main phase of an application (ie after the initialisation is complete). By this time, the user may well have committed some data into the application, which would be lost if the application suddenly terminated.

In these cases, a *retry* philosophy is much more appropriate: the application cleans up any interim semi-allocated resources, "rolls back" to its previous good state, and presents an error message informing the user what has happened. It is up to the user to try to correct the error condition, and then re-initiate the previous action.

When resources need to be tidied explicitly

It may be noted that there are no calls to `p_close` in *events.c*. These calls are unnecessary in this program, because the channels are automatically closed, by the operating system, when the application exits.

This situation must be contrasted carefully with that when a resource (such as a channel to an i/o device) is used *transiently* by an application. In this case, it *must* be freed as soon as it is no longer needed.

Inter-Process Communication Events2 and Subproc

Events2 extends the functionality of *Events* by supporting the facility to launch a sub-process. A sub-process is automatically started during program initialisation, and once it has terminated, the user can press ENTER to re-launch it again.

The sub-process mimics the carrying out of some extended activity. When it finishes, it signals the fact of its completion to *Events2*, together with an "answer", which *Events2* displays on the screen. This answer has in fact been calculated from a parameter passed by *Events2* to *Subproc* on the command line.

Subproc and *Events2* therefore carry out a restricted form of inter-process communication, and thereby illustrate some of the *multi-tasking* possibilities in writing software for SIBO computers.

A third event source

Events2 remains alert to the possibility of a signal from *Subproc*, even though it is busy responding to timer events and keyboard events. *Events2* manages this via a straightforward extension of the architecture in *Events*: one more "event source" is added into the picture.

Thus there is one more status word - *substat* - and one more test in the main loop in *main*:

```
else if (substat!=E_FILE_PENDING)
    ReportSub();
```

That is, whereas *main* used to poll up to *two* status words, on returning from a call to *p_iowait*, it now polls up to *three* status words: *substat* in addition to *keystat* and *timstat*.

However, this new event source differs from the previous two in that it is not an i/o device. There is no *p_ioa* call to request notification from *Subproc*; rather, the corresponding "queue" call is *p_logona*.

In the line of code

```
p_logona(pid,&substat);
```

the calling application (*Events2*) is requesting asynchronous notification of the eventual termination of the process identified by *pid*. This notification consists of two parts (now familiar):

- a value other than *E_FILE_PENDING* is written into *substat*
- the application is signalled (so that its i/o semaphore increments).

How *Events2* passes data to *Subproc*

Events2 passes its original data to *Subproc* via the command line.

In any case like this, there has to be an agreement between the two processes as to the format of the command line. Here, the protocol is established in the shared header file *subproc.h*, which defines the following struct:

```
typedef struct
{
    HANDLE pid;
    VOID *poff;
    ULONG data;
} SUBPROC_CL;
```

The members of this struct serve the following purposes:

- pid* identifies the process which launched *Subproc* (ie *Events2*)
- poff* an address within the dataspace of *Events2* where *Subproc* should in due course write back the "answer" it discovers
- data* the original seed data for *Subproc* to operate with.

The following code sets up this struct and creates and runs *Subproc*:

```
LOCAL_C VOID LaunchSub(VOID)
{
    HANDLE pid;
    SUBPROC_CL cl;
    TEXT subname[P_FNAME_SIZE];

    p_fparse("subproc.img", DatCommandPtr, &subname[0], 0);
    cl.pid=p_getpid();
    cl.poff=&answer;
    cl.data=p_date();
    if ((pid=p_execl(&subname[0], &cl, sizeof(cl)))<0)
    {
        p_notifyerr(pid, "Failed to launch subprocess", 0, 0, 0);
        return;
    }
    p_logona(pid, &substat);
    p_presume(pid);
}
```

The variable *answer* is a static. Clearly, it would be a significant error to include it on the stack of *LaunchSub*, since this stack will have unwound before *answer* gets written to (by *Subproc*).

As can be seen, in this example, the data value passed to *Subproc* is just the current time/date, as returned by *p_date*.

Note the call to `p_fparse`, which calculates the presumed full pathname of *subproc.img*, under the assumption that this is in the same directory as *events2.img*. (The full path of *events2.img* is written to `DatCommandPtr` by the operating system, when *Events2* starts running.)

For more on the operation of `p_logona`, see the *Plib Reference* manual.

The code in Subproc

The code in *subproc.c* is very straightforward:

```
#include <p_std.h>
#include <p_gen.h>
#include <p_math.h>
#include <epoc.h>
#include "subproc.h"

GLREF_C TEXT *DatCommandPtr;

GLDEF_C INT main(VOID)
{
    TEXT *pb;
    SUBPROC_CL *pcl;
    UWORD answer;
    WORD logstat;

    pb=DatCommandPtr+p_slen(DatCommandPtr)+1;
    if (*pb!=sizeof(SUBPROC_CL))
        return(E_GEN_ARG);
    pcl=(SUBPROC_CL *) (pb+1);
    if (!p_logona(pcl->pid,&logstat))
    {
        answer=(UWORD)(p_randl(&pcl->data)%(2*60*32));
        p_sleept(answer);
        if (logstat<0)
            p_pcpyto(pcl->pid,pcl->poff,&answer,sizeof(UWORD));
    }
    return(0);
}
```

The way *Subproc* mimics performing a lengthy calculation is simply to call `p_sleept`, for some random period up to two minutes in duration.

Note that *Subproc* exits at once if the command line passed to it is not of the expected type. On most SIBO computers, the result of an application calling `p_exit` with a negative value is a Notifier reporting the abnormal exit.

Note too that just as *Events2* asynchronously logs onto *Subproc*, *Subproc* logs onto *Events2*. However, this is for a different reason. Namely, *Events2* might terminate in the meantime, while *Subproc* is busy "calculating". In this case, a non-negative number will be written into `logstat`. If *Subproc* ignored this fact and proceeded regardless to `p_pcpyto` data into the process currently having identifier `pcl->pid`, in all probability the call would fail - process identifiers take a long time to get re-used, and the operating system would ignore the `p_pcpyto` on account of a non-existent `pid` being specified. But there is always an outside chance that the `pid` will indeed be re-used by the time *Subproc* has completed - in which case the `p_pcpyto` could cause random damage to a blameless application.

The reason there is a test on the result of the `p_logona` call in *Subproc* is to guard against the rare case of *Events2* exiting even before *Subproc* reaches the `p_logona` call.

Mechanisms for inter-process communication

The above mechanism is a very simple illustration of a form of IPC (*inter-process communication*). Epoc supports a considerable range of IPC services, chief amongst them being *IPC messaging*.

Messaging also involves the same principles of status words, synchronous or asynchronous requests, and signalling an application. In some cases, one process will signal another one explicitly, using the function `p_iosignalbypid`; in other cases, the signalling is performed by the operating system, say in response to a `p_mfree` call in an application (indicating that the message sent has been processed).

See the *Plib Reference* manual for more details of these different forms of IPC.

Debugging cooperating applications

One method of *debugging* the interaction between *Events2* and *Subproc* is as follows:

- leave both image files in the same directory on the PC
- in that directory, type `sdbg events2` to start debugging *Events2*
- place a breakpoint on the `p_logona` call in *events2.c*
- run *Events2* until it reaches this breakpoint
- note incidentally that the `p_fparse` call in `LaunchSub` automatically deduces that since *Events2* has been launched from a remote directory (ie on the PC), *Subproc* should also be launched (if possible) from this same remote directory
- when the Debugger breaks at the `p_logona` call, the code and data segments for *Subproc* will already be created (by virtue of the prior `p_execc` call)
- refresh the main list of remote processes; select *Subproc* and break into it (use the Break Into menu command)
- scan to the beginning of `main` in *subproc.c* and set a breakpoint there
- issue the Apply Breakpoints menu command (whence the windows will go blank)
- switch back to the window debugging *Events2* and step over the `p_presume` call
- this will have the effect of starting the execution in *Subproc* - in which case the breakpoint in `main` will be reached.

Debugging now continues, with two different debugging windows - one for *Events2*, and one for *Subproc*.

Error handling in Events2

The call to `p_execc` in *Events2* can fail for a variety of reasons - chief amongst them lack of system memory. Note that *Events2* does not treat this as a fatal error; rather, the error is reported to the user, and the program carries on (taking care to leave the internal variable `subexist` set correctly).

The user has the opportunity to respond to the message by freeing up some system memory, and then pressing ENTER to make another attempt to launch *Subproc*.

Socially responsible programming in Events2

There is one more change between *events.c* and *events2.c*, which actually corrects a bug deliberately left in *Events*. This change is the addition of a call to `p_unmarka` at the start of `main`.

If a SIBO computer is left switched on while *Events* is running, it will fail to auto-switch-off subsequently. This is because of the regular timer activity in the program, which keeps resetting the inactivity counter of the computer.

Suppose that the user turns the computer off explicitly. However, if an alarm rings at some stage, *Events* will start running again, and if the user is not at hand to notice the alarm, the result will soon be flattened batteries - regardless of the auto-switch-off setting. (Try it and see.)

However, the addition of the call to `p_unmarka` prevents any activity within *Events2* from resetting the inactivity counter. As a result, *Events2* is much more socially responsible than its precursor.

See the *Plib Reference* manual for more details on `p_unmarka`.

Data received from a serial port

Events3 extends the functionality of *Events2* by including yet another event source: data received from a serial port. Characters are read one at a time from the serial port and, if they are printable, they are echoed onto the screen.

For background information on the serial port, see the *Serial Port* chapter of the *I/O Devices Reference* manual.

Opening the serial port

The serial port is an i/o device: before services such as P_FREAD can be requested from it, a suitable channel has to be opened.

The contents of the `openSer` routine in *events3.c* are as follows:

```
LOCAL_C VOID OpenSer()
{
    INT ret;

    ret=p_open(&serH,"TTY:B",-1);
    if (ret<0)
        ret=p_open(&serH,"TTY:A",-1);
    if (ret<0)
    {
        p_notifyerr(ret,"Opening serial port",0,0,0);
        serH=0;
    }
}
```

If the attempt to open "TTY:B" fails, an attempt is made to open "TTY:A" instead; only if both attempts fail is this fact reported to the user.

Typical reasons for it being impossible to open a serial port are that port already being in use (eg by Link software) or the port not being physically present.

In contrast to the retry mechanism for launching *Subproc* which is built into *Events3*, there is no corresponding retry mechanism for opening the serial port. A trivial amendment could be made to allow this.

Active and inactive event sources

Events3 adds this fourth event source to the bottom of the prioritised list in `main`, which now has the following schematic form:

```
GLDEF_C VOID main(VOID)
{
    INITIALISE();
    QueueTimer();
    QueueKey();
    LaunchSub();
    QueueSer();
    FOREVER
    {
        p_iowait();
        if (keystat!=E_FILE_PENDING)
        {
            PROCESS_KEY();
            QueueKey();
        }
        else if (timstat!=E_FILE_PENDING)
        {
            PROCESS_TIMER();
            QueueTimer();
        }
        else if (subexist && substat!=E_FILE_PENDING)
            ReportSub();
        else if (serH && serstat!=E_FILE_PENDING)
        {
            p_tickle();
            DisplaySerChar();
            QueueSer();
        }
    }
}
```

Note however that the test for whether to call `ReportSub` has changed its form slightly. No longer is it sufficient just to poll `substat`; it is also necessary to check on the current value of `subexist`:

- `subexist` is set to `TRUE` when *Subproc* is successfully launched
- `subexist` is set back to `FALSE` whenever it is discovered that *Subproc* has completed
- thus the only time the value of `substat` should be polled is when `subexist` is `TRUE`.

Without this double check, a variety of bugs can be demonstrated - all of which stem from serial port events being misinterpreted as reports of *Subevent* completing.

Similarly, for the sake of correctness, the value of `serH` should be tested before going on to poll `serstat`: `serH` is 0 if it has proved impossible to open a serial port. (Actually, no immediate harm will ensue if the check on `serH` is omitted, given that this is the last of the event sources in the priority list.)

The reason no corresponding checks are required for the keyboard and timer event sources is that these event sources are always active:

- the program terminates at once if it proves impossible to open a channel to these devices
- every time an event is delivered from these sources, another request (for yet another event) is issued straightaway.

The variables `serH` and `subexist` in *Events3* play the role of what are sometimes called *active words* for their event sources - complementing the roles of their status words:

- active words have to be set and cleared by the application itself
- status words are written to by system software; generally, application software only *reads* the values of status words
- active words are set when a request *is made*; status words are set when a request *completes*.

One other possible advantage in maintaining active words for event sources is to avoid making the mistake of sending a `P_FCANCEL` request to an i/o channel which has not had a read request made on it. Some i/o devices panic the application if this happens.

Debugging applications with serial comms

Programs involving serial comms are inevitably harder to debug than without serial comms.

This is because the Debugger itself uses up one serial port on the computer. In general, *Events3* will be unable to open any serial port currently being used by Link software (such as the SIBO Debugger).

One possibility, however, is to use say port A to debug the application, and send serial data to the application via port B.

The role of `p_tickle`

Whenever serial port data is received by *Events3*, a call is made to `p_tickle` to reset the inactivity counter. This prevents the computer from auto-switching off part way through processing an incoming stream of serial data.

Note that there is no requirement to make a corresponding call whenever a keypress is received, since the operating system does this automatically.

Finally, it would of course be an error to make this call whenever the timer expires - since this would keep the computer permanently switched on.

Note that some early versions of the Epoc operating system may fail to support `p_tickle` (versions prior to 2.11).

Yielding CPU in compute-intensive programs

Events4 adds in yet another type of event source - one that is subtly different from all the others so far introduced. This is an event source which is always ready to run immediately. However, it is deliberately located in a low position in the priority listing, to ensure that other event sources are processed preferentially.

A real-life example of such an event source would be a recalculation computation in a spreadsheet, or a reformatting calculation in a word processor. Again, a long file operation - such as building the index of a DBF file - should also be broken up into chunks, so as to let the application to respond to other event sources in the meantime. Finally, a game may think indefinitely until such time as it is told to stop.

In *Events4*, this continual "thinking" is simulated by continually adjusting the display of part of the boundary of the console window. The constant visible change is meant to reflect constant internal activity.

Idle objects

Event sources such as just discussed are sometimes referred to as *idle objects*. The meaning of the name is that they only get a chance to run when the application is *otherwise* idle. For example, if an application is busy responding to keypresses, it is certainly not idle, and so any idle objects have no opportunity to run.

On the other hand, this name is of course potentially misleading, since the activity represented by the idle object is anything but idle.

The meaning of calling `p_iosignal`

The contents of the "queue" routine for the idle object in *Events4* is just the single line

```
p_iosignal();
```

Since there is no genuine i/o connected with an idle object, there is no system software that automatically signals the application when the i/o has completed. That is why the "queue" routine itself calls `p_iosignal`.

At the same time, it might be thought that this routine should write to a status word. Actually, however, there is no point in doing so: what matters is simply that the event source is active; in that case, it is automatically ready to deliver an "event" (ie to take more CPU).

Just as some event sources have no need to maintain an active word (since they are always active), others (ie idle objects) have no need to maintain a status word, since whenever they are active, they are ready to deliver an event.

One drawback of continuous activity

Interestingly, *Events4* has re-introduced the "stay awake" bug that *Events2* managed to fix from *Events*. This fact can easily be verified by user experimentation.

The call made to `p_unmarka` is now ineffective since, as is explained in the *Plib Reference* manual (section on `p_unmarka`), it is the `sys$null` process which administers the auto-switch-off, yet it will have no opportunity to run if any other applications are continually running. The point is that the priority of `sys$null` is lower than that of any other process.

One possible solution here is to call, say, `p_sleept(2)` every so often, to try to allow `sys$null` to run. However, this too will fail in the rare case when there are *two* similarly anti-social applications running simultaneously on the SIBO computer. Even if they both call `p_sleept` periodically, there is little chance of them both sleeping at the same time - which would be required in order for `sys$null` to run.

A better solution is to call `p_allowoff` from time to time. By doing so an application assumes responsibility for performing the auto-switch-off that would otherwise be done by `sys$null` if it had an opportunity to run.

Remarks on process priorities

The above problem of preventing the SIBO computer from auto-switching off is not the only potentially deleterious side-effect of continuous activity. Just as continuous activity within *Events4* prevents `sys$null` receiving any CPU, so too will any *other* lower priority processes be locked out.

As it happens, *Subproc* and *Events4* both run at the same process priority - the default of `0x80`. (See the chapter *Building an Application* for details on how process priorities are initialised.) However, suppose the process priority of *Subproc* were lowered to say `0x70`. In that case, continuous activity within *Events4* would mean that *Subproc*, although launched, would never receive any CPU. Thus *Events4* would perpetually display the message "Subp launched", but never the completion message "Subp slept xxx ticks".

The solution here is that any process about to become computationally intensive over a long period of time should consider lowering its process priority, say to `0x70` - using the call `p_setpri`. For some related discussion, see the section on `wStartCompute` in the *Window Server Reference* manual.

The call `p_ioyield`

One more potentially surprising fact about continuous activity within a program ought to be mentioned.

Clearly, not every program needs to be structured with an *asynchronous event processing loop* as in the *Events* programs. Whilst such a loop is undoubtedly the correct architecture for a larger program, other simple programs may have alternative structures.

For example, *Subproc* makes no call to `p_iowait` - nor does it need to, being purely single-threaded. Again, the example Plib programs discussed in the chapter *Building an Application* (eg *p_hello*, *p_comp* and *p_prndir*) likewise survive without an asynchronous event processing loop, since they are likewise single-threaded.

However, there may be a temptation for programs to use the following *semi-asynchronous* mechanism, in order to be able to respond to keypresses (say) whilst being mainly dedicated to some continuous activity:

- continue processing queue an asynchronous request to receive a keypress
- start the continuous activity
- every so often, test to see if the status word is still `E_FILE_PENDING`
- if it is, continue processing

In other words:

```
FOREVER
{
    QueueSer(&keystat);
    while (keystat==E_FILE_PENDING)
        MoreProcessing();
    ProcessKey();
}
```

However, this program may totally fail, with `keystat` never changing from `E_FILE_PENDING`.

The reason for this is rather complicated, but it is worth understanding. It has to do with what happens when an asynchronous event completes. This completion often involves *two distinct phases*:

- the completion itself happens on an interrupt
- an interrupt service routine runs, and the i/o device changes some of its internal variables to record this fact
- however, the i/o device generally cannot write to the associated status word during the interrupt service routine itself
- this is because there are strict rules on what can and cannot be done within an interrupt service routine (see the chapter *Writing Device Drivers* in the *Additional System Information* manual for more details)
- instead, the i/o device relies upon its so-called *wait handler* routine to be called, so that it can finish the job of completing the request - writing to the status word and signalling the application
- these wait handler routines only have an opportunity to run inside the call `p_iowait` (or equivalent).

In other words, the description given earlier in this chapter of the functioning of `p_iowait` was incomplete in one important respect: not only does this routine take note of the value of the i/o semaphore for the application, and suspend the application so long as this remains negative; it also gives all i/o device channels in the application an opportunity to run their wait handlers.

Accordingly, the earlier code has to be changed into

```
FOREVER
{
    QueueSer(&keystat);
    while (keystat==E_FILE_PENDING)
    {
        MoreProcessing();
        p_iosignal();
        p_iowait();
    }
    ProcessKey();
}
```

which is in fact much more in line with the main routine of the *Events4* application discussed.

For convenience, the single call `p_ioyield` can be used with the same effect as `p_iosignal` followed by `p_iowait`.

Note incidentally that it is a rare application that needs to actually *write* a wait handler; there is of course no need to write a wait handler routine just because an application involves asynchronous i/o.

General remarks

Multi-threadedness and multi-tasking

For the sake of clarity, the two different notions of *multi-tasking* and *multi-threadedness* ought to be compared and contrasted:

- multi-tasking involves two (or more) different processes; multi-threadedness involves two (or more) event sources *within one process*
- the operating system takes care of multi-tasking automatically, on behalf of applications; multi-threadedness requires conscious effort from an application
- multi-tasking is *pre-emptive* in that a process with a higher priority that is ready to run will always displace a lower priority process that is running
- multi-threadedness is *non* pre-emptive in that an event source with a higher priority has to wait until a lower priority event source voluntarily gives up CPU, before being able to run.

Subprocess or idle object?

In the design of large systems, there is considerable scope for decision-making on whether to assign compute-intensive tasks to idle objects within an application, or to a subprocess of the application proper. Some factors that may be considered in making such a decision are:

- having two different processes automatically allows a total combined dataspace greater than 64k
- a subprocess need not worry unduly about creating "holes" ("fragmentation") in its allocator heap, since these will all vanish when the subprocess terminates (when its entire heap vanishes); however, if an idle object is transient and terminates well before the overall application finishes, any fragmentation it creates in the allocator heap may have a more damaging long-term effect (see the chapter *Memory Allocation* in the *Plib Reference* manual)
- an idle object is much more tightly bound to the main program than is a subprocess
- communications between a subprocess and the main program have to be much more formalised than is the case with an idle object
- an idle object can draw to the same windows on the screen as the main application, but not so for a subprocess (except in the case of the MC - see the function `wAttachToClient` in the *Window Server Reference* manual).

Window redraws as an event source

Up till now, one very important event source has not been mentioned in this chapter. This is the event of a window requesting itself to be redrawn. (In fact, the request originates from the Window Server.)

These events do not occur when using console i/o, nor when using windows with back-up bitmaps: windows are automatically redrawn by the Window Server in these cases. However, high quality applications undertake window redrawing by themselves, so as to avoid the large ram overheads of the back-up bitmaps. (See the *Introduction* chapter of the *Window Server Reference* manual.)

Such applications *must* remain responsive to redraw events at all times - otherwise their screen will remain blank if they are suddenly task switched into foreground, or if an overlapping menu or dialog window is removed. This is all the more reason for these applications to adopt the sort of general event processing architecture outlined in this chapter.

The APPMAN and ACTIVE classes in OLIB

The APPMAN and ACTIVE classes in *olib.dyl* provide system support for object-oriented applications

- representing their event sources (each event source is represented by an "active object")
- prioritising these event sources
- maintaining active and status words for each event source
- calling `p_iowait` at appropriate times
- ensuring that all event sources are properly polled.

At the same time, the APPMAN class ("appman" is short for "application manager") provides an automated mechanism for error handling, as well as handling the interface to resource files (see the *Resource Files* chapter in the *Additional System Information* manual).

These topics are further discussed in the Object Oriented Programming Guide. In addition, the APPMAN and ACTIVE classes are documented in the *OLIB Reference* manual.

CHAPTER 6

COPY-PROTECTING SOFTWARE

Introduction

This chapter considers various mechanisms to "copy-protect" software written for SIBO computers.

Possible *goals* of copy-protection include:

- software should be able to run, only from the original SSD it is supplied in
- software should be able to run, only on one particular computer
- first generation copies should be allowed, but not second generation copies (ie direct copies made from the original SSD would run, but not copies of these copies)
- possibly, the number of first generation copies allowed could be limited.

These different goals vary in their applicability to the different computers in the SIBO range. For example, software that disallowed even first generation copies would be very unwelcome on the Series 3, because it prevents *consolidation* - the process whereby users copy more than one different application onto the same SSD. It is unlikely that such software would sell at all well. On the other hand, this kind of *strict* copy protection makes more sense for the HC, and for corporate ("vertical") programs generally.

At the same time, software developers may wish to apply differing amounts of sophistication in their copy protection schemes - some being willing merely to frustrate the casual would-be copier, and some being determined not to allow copying at all (if possible).

This chapter does not favour one copy protection method over all others. Rather, it simply hints at various different possibilities. Software developers may find it convenient to mix and match the different proposals made, as well as others (along the same lines) that they can think of themselves.

By the very nature of the subject, too much documentation would be self-defeating. Once any algorithm to implement copy protection becomes widely known, methods to defeat this algorithm may be developed and circulated.

The free space method

This method aims at *strict* copy protection: the software will only run if it is on the original SSD.

This simple yet effective method turns one of the commonly debated 'disadvantages' of using Flash storage into an advantage.

The method is based around the fact that when a file is deleted from a Flash SSD it remains on the pack taking up space but is no longer practically accessible. Assuming that an application master pack contains a deleted file, if someone were to copy this pack using the usual copy from the root including subdirectories, the resulting copy of the master pack will differ fundamentally: the deleted file no longer exists. This in turn means, rather conveniently, that the amount of free space on the pack will be different.

Given the above you can then hard-code into your application a check to see if the figure for space on the pack matches the figure you expect: if it does then all well and good; else, do not allow the software to run.

The way to find the amount of free space on an SSD is to call `p_dinfo`. For example, a program that prints the amount of free space on the disk the program was launched from:

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

GLREF_D TEXT *DatCommandPtr;

GLDEF_C INT main(VOID)
{
    P_DINFO dinfo;

    p_dinfo(DatCommandPtr,&dinfo);
    p_printf("Free space is %lu",dinfo.free);
    p_getch();
    return(0);
}
```

First generation copying methods

Methods that allow first generation copies to run, but not second generation copies, rely on a separate "AppCopy" program being provided on the master SSD, in addition to the program itself.

First generation copies can only be made using the AppCopy program. Copies made using ordinary Copy commands will fail. Further, AppCopy will refuse to copy first generation copies into second generation copies.

Essentially, AppCopy does not make an exact copy, but changes some of the bytes in the application program. This can be done, even on Flash SSDs, provided:

- the original values of the bytes are all 0xff's
- the bytes are adjusted *one at a time* (assuming that a complete copy has already been made, and that this complete copy has to be altered).

The last point is most important; the Flash filing system tests for the special case of only one byte being written, and in this case, alters the physical byte on the SSD. In other cases, the physical structure of the file is significantly changed.

Preserving checksums

Note that in adjusting bytes in a program file, care has to be taken not to disturb the code or data checksums (as reported eg by the tool *edump.exe*). Otherwise, the operating system will refuse to run the program, believing it to be corrupt.

The checksums can be preserved in either of two ways:

- provided enough bytes are changed, the checksum can be left the same as it was originally
- rather than changing the program part of the image file, one of the *add-files* inside the image file should be changed (see the chapter *Building an Application* for details of add-files).

In general, the second method is preferable.

What kind of change should be made

The change made to the program file has two purposes:

- the file is now recognisably a first generation copy
- the change contains data somehow allowing this first generation copy to run, in a way preventing a straight copy of this file from running on another computer.

Possible ideas on this second point include:

- information from an environment variable specially created on the target computer (secretly and with a random value), by the AppCopy program
- the date the copy was made (so that the copy will "expire" after a certain length of time)
- details about the low-level structure of the SSD (see below).

To make the mechanism less obvious to a casual browser, any information from say an environment variable ought to be stored in the file *encrypted* in some way.

Restricting the number of copies made

In order to restrict the number of first generation copies ever made to eight, say (which would not be unreasonable), certain bytes in the *master* copy of the program could be changed. As above, these bytes would start with the value 0xff, and would have to be written one at a time.

The documentation for the product would have to state clearly that AppCopy could only be used eight times. This would have the effect of making the owner of the software most wary against making cavalier bootleg copies.

This method will of course only work if the SSD containing the original copy can be written to. This will be impossible for OTP (One Time Programmable) or masked ROM SSDs. In this case, information about the number of first generation copies made could be stored in another environment variable, though of course this method would be easier to subvert.

Low level SSD information

Each SSD contains a so-called "unique ID" which can be used to identify it.

Another potentially very useful piece of information would be the *physical pack offset* of the start of a nominated file on an SSD. An AppCopy program could create a small file on the target SSD, and then delete it, before copying the program file across. The physical pack offset of the start of the program file could then be written into the program file (possibly in encrypted form), for the program to check when it starts to run.

Alternatively, information could be read *from the deleted file*.

These types of information can be read by use of the `p_locreadpdd` function, described in the *Files* chapter of the *PLIB Reference* manual.

Copy-protection by changing the ROM

In the case of the HC, it is possible to use the tool *romwrite.exe* to overwrite portions of a file *custom\$.dat* in the ROM of the HC. See the chapter *Introduction to the HC* in the *HC Programming Guide* for details of the operation of *romwrite.exe*.

Programs can then check that they are running on a given specified HC.

To read the contents of *custom\$.dat*, just open the file as normal, specifying the full path name *rom::custom\$.dat*.

One other possibility is to change the contents of the ROM more radically, eg placing certain software into the ROM, with a program on an SSD refusing to run unless this software is present in the ROM. For more details, again see the *HC Programming Guide*.

CHAPTER 7

COMPATIBILITY

Introduction

The vast majority of existing software will, in principle, run on all machines in the SIBO range, provided that the display can accommodate itself to the different screen sizes.

For details of the differences between SIBO machines see *Writing Software for the HC* in the *HC Programming Guide* and the *Series 3 family compatibility* section of the *Series 3 Programming Overview* chapter in the *Series 3/3a Programming Guide* manual.

What machine am I running on?

Most SIBO machines can be distinguished by their screen sizes, as determined by the return value from a call to `p_getlcd`. The possible return values for existing SIBO machines are as follows:

<code>E_LCD_640_400</code>	(0)	a 640x400 pixel display as on the MC 400
<code>E_LCD_640_200_SMALL</code>	(1)	a 640x200 pixel display as on the MC 200
<code>E_LCD_160_80</code>	(4)	a 160x80 pixel display as on the HC
<code>E_LCD_240_80</code>	(5)	a 240x80 pixel display as on the Series 3
<code>E_LCD_480_160</code>	(11)	a 480x160 pixel display as on the Series 3a and Series 3c
<code>E_LCD_240_100</code>	(12)	a 240x100 pixel display as on the <i>Workabout</i>
<code>E_LCD_240_160</code>	(14)	a 240x160 pixel display, as on the Siena

This will distinguish all machines except the Series 3a and Series 3c, which have screens of the same size. These two machines can be distinguished by means of a call to the function `p_returnexpansionportinfo`, present in machines with EPOC version 3.90F or later, as in the following example code fragment.

```
UINT lcdtype;
UINT version;
UINT port;

lcdtype=p_getlcd();
if (lcdtype==E_LCD_480_160)
{ /* S3a or S3c */
version=p_version();
if (version>=0x390F)
{ /* safe to call p_returnexpansionportinfo */
port=p_returnexpansionportinfo();
if ((port & 0x0700)==0x0300)
{
/* must be S3c */
}
else
{
/* must be S3a */
}
}
else
{
/* EPOC version less than 3.90, so must be S3a */
}
}
else
{
/* machine determined by LCD type */
}
```

INDEX

- .afl files
 - add file lists, 3-13
 - add file lists - changing, 3-14
- .app files
 - versus .img files, 3-13
- .dfl files
 - add file lists for DYLS, 3-14
- .dyl files
 - building with emake.exe, 3-14
- .img files
 - applications, 3-1
 - control over, 3-12
 - creating, 3-3
 - versus .app files, 3-13
- .ldd files
 - building with emake.exe, 3-14
- .pdd files
 - building with emake.exe, 3-14
- .pic files
 - application icons, 3-13
- .pr files
 - application project files, 3-1
 - project files - a first look, 3-2
- .rsc files
 - resource files - application, 3-13
- .rzc files
 - resource files compressed, 3-13
- .shd files
 - shell data files, 3-13
- ACTIVE class
 - remarks on OLIB, 5-16
- add file list
 - application, 3-13
 - application - changing, 3-14
 - application DYLS, 3-14
- application
 - a first look at .img files, 3-3
 - add file lists, 3-13
 - add file lists - changing, 3-14
 - add file lists for DYLS, 3-14
 - app files vs img files, 3-13
 - batch files - housekeeping, 3-6
 - building, 3-1
 - building - introduction, 3-1
 - building - more complex example, 3-9
 - building - more complex example PLIB, 3-9
 - building - multi-file, 3-9
 - building - with assembler, 3-11
 - building emake.exe, 3-14
 - building eremake.exe, 3-14
 - copy protection see copy protection, 6-1
 - debugging on SIBO system, 3-4
 - file - control over, 3-12
 - file transfer to SIBO system, 3-1, 3-3
 - graphics and, 3-10
 - heap size - minimum, 3-12
 - Hwif - building, 3-10
 - icon files, 3-13
 - information from edump.exe, 3-12
 - multi-tasking - remarks on, 5-16
 - multi-threaded - remarks on, 5-16
 - priority - control over, 3-12
 - resource .rsc files, 3-13
 - resource files .rzc compressed, 3-13
 - running on SIBO system, 3-3
 - shell data files, 3-13
 - version number - control over, 3-12
- application development
 - C SDK equipment required, 3-1
 - C SDK files for oop, 3-2
 - C SDK files required, 3-2
 - first example, 3-2
 - fundamental guide-lines, 5-1
- applications
 - .img files, 3-1
- APPMAN class
 - remarks on OLIB, 5-16
- assembler
 - applications - containing, 3-11
- asynchronous I/O
 - programming, 5-2
- auto-switch-off
 - p_unmarka, 5-14
 - SYS\$NULL - chance to run, 5-14
- background processing
 - programming example, 5-2
- batch files
 - application housekeeping, 3-6
 - compilation, 3-6
 - compilation complications, 3-7
- build
 - C SDK configuration files, 3-11

- building
 - applications, 3-1
 - applications - control over, 3-12
 - applications - more complex example, 3-9
 - applications - more complex example PLIB, 3-9
 - applications - multi-file, 3-9
 - applications - with assembler, 3-11
 - applications introduction, 3-1
 - applications with graphics, 3-10
 - applications with Hwif, 3-10
- C
 - manuals - Topspeed C, 2-1
- C SDK
 - build configuration files, 3-11
 - configuration file tscfg compiler, 3-11
 - configuration file tsprj.txt, 3-11
 - configuring the redirection file, 1-3
 - configuring the Topspeed project system, 1-2
 - directory structure, 1-2
 - documentation version, 1-1
 - equipment required, 3-1
 - files required, 3-2
 - files required - oop, 3-2
 - installation, 1-1
 - installation phases, 1-1
 - manuals, 2-1
 - manuals - where to start, 2-2
 - overview SIBO system, 2-1
 - professional version, 1-1
 - SIBO software, 1-2
 - small model code, 1-1
 - standard version, 1-1
 - Topspeed C, 1-1
 - Topspeed software, 1-2
 - variants of, 1-1
- C++
 - object oriented programming and, 2-3
- cancelling timers
 - in general, 5-6
- Clarion Software
 - Topspeed C, 1-1
- CLIB
 - console device I/O, 4-3
 - DOS file names, 4-3
 - file handle conversion, 4-2
 - floating point emulator, 4-3
 - floating point emulator - panic 80, 4-3
 - int86x implementation, 4-2
 - intr implementation, 4-2
 - library - building, 4-4
 - missing DOS functions, 4-1
 - notes on, 4-1
- CLIB & PLIB
 - contrasted, 3-4
- code size
 - application CLIB, 3-5
 - application PLIB, 3-5
- compatibility
 - software on SIBO systems, 7-1
- compilation
 - batch file complications, 3-7
 - batch files, 3-6
 - optimisation warning, 3-11
 - TSC vs TSCX, 3-8
- compute intensive application
 - CPU yielding, 5-13
- configuration file
 - C SDK tscfg compiler, 3-11
 - C SDK tsprj.txt, 3-11
- configuration files
 - C SDK build, 3-11
- console device
 - implementation in CLIB, 4-3
 - in general, 5-6
- cooperating applications
 - debugging, 5-11
- copy protection
 - first generation copy method, 6-2
 - free space method, 6-1
 - introduction, 6-1
 - low level SSD method, 6-3
 - mechanisms, 6-1
 - ROM changing method, 6-3
- CPU intensive
 - yielding in applications, 5-13
- CPU yielding
 - compute intensive applications, 5-13
 - programming, 5-2
- customised
 - libraries, 3-10
- debugging
 - applications - on SIBO system, 3-1, 3-4
 - cooperating applications, 5-11
 - serial port comms applications, 5-13
- device driver
 - LDD - building with emake.exe, 3-14
 - PDD - building with emake.exe, 3-14
- devices
 - I/O in general, 5-5
- directory structure
 - C SDK, 1-2
- documentation
 - C SDK - where to start, 2-2
- documentation sources
 - programming, 5-1
- DOS functions
 - CLIB - missing functions, 4-1
- DYL
 - add file lists into application, 3-14
 - building with emake.exe, 3-14
- dynamic library
 - DYL - building with emake.exe, 3-14

- edump.exe
 - application information, 3-12
 - utility program, 3-12
- emake.exe
 - application building, 3-14
 - application conversion, 3-1
 - utility program, 3-14
- EPOC
 - explained, 2-2
- epocinit
 - project file statement, 3-6
- eremake.exe
 - application building, 3-14
 - utility program, 3-14
- error handling
 - a first look, 5-7
 - events and, 5-7
 - resource tidying, 5-8
- errors
 - panic number ranges, 2-3
 - panics explained, 2-3
- events
 - active and inactive sources, 5-12
 - ACTIVE class - remarks on, 5-16
 - APPMAN class - remarks on, 5-16
 - idle objects, 5-14
 - multiple source examples, 5-3
 - p_iosignal - function of, 5-14
 - prioritisation of sources, 5-5
 - window redraw - remarks on, 5-16
- example code
 - detecting which SIBO system, 7-1
- fatal program errors
 - panics explained, 2-3
- file handle
 - conversion in CLIB, 4-2
- file names
 - DOS implementation in CLIB, 4-3
- file transfer
 - applications - to SIBO system, 3-1
 - applications to SIBO system, 3-3
- floating point
 - emulator - panic 80 in CLIB, 4-3
 - emulator in CLIB, 4-3
- fundamental guide-lines
 - programming, 5-1
- graphics
 - applications and, 3-10
- heap size
 - minimum, 3-12
- hello world
 - application - PLIB first example, 3-4
 - application first example, 3-2
- Hwif
 - application building, 3-10
- I/O devices
 - in general, 5-5
- icon files
 - application, 3-13
- IDE
 - Topspeed Integrated Development Environment, 3-1
- idle object
 - contrasted with sub-process, 5-16
- idle objects
 - events, 5-14
 - p_iosignal - function of, 5-14
- image file
 - control over, 3-12
- include file
 - stdepoc.h, 3-11
- installation
 - C SDK, 1-1
- int86x
 - implementation in CLIB, 4-2
- inter-process communications
 - in Events2 - and Subproc, 5-8
 - mechanisms, 5-10
- intr
 - implementation in CLIB, 4-2
- IPCS
 - in Events2 - and Subproc, 5-8
 - mechanisms, 5-10
- key presses
 - remarks on, 5-4
- keyboard input
 - programming example, 5-2
- LDD
 - building with emake.exe, 3-14
- libraries
 - customised, 3-10
- library
 - building CLIB, 4-4
 - CLIB missing DOS functions, 4-1
 - CLIB notes on, 4-1
- machine
 - detecting which SIBO system, 7-1
- manuals
 - C SDK, 2-1
 - Topspeed, 2-1
- multi-tasking
 - applications - remarks on, 5-16
- multi-threaded
 - applications - remarks on, 5-16
 - programming, 5-2
- object oriented programming
 - C++ and, 2-3
 - Psion C approach, 2-3
- optimisation
 - compilation warning, 3-11
- p_hello.c
 - source code, 3-5
- p_iosignal
 - function of, 5-14

- p_iowait
 - remarks on, 5-4
 - status words - declaring, 5-7
 - status words - remarks on, 5-5
- p_ioyield
 - remarks on, 5-14
- p_tickle
 - function of, 5-13
- p_unmarka
 - auto-switch-off, 5-14
- panic 80
 - floating point emulator - CLIB, 4-3
- panics
 - explained, 2-3
 - number ranges, 2-3
- PC
 - C programs, 1-1
- PC based
 - development, 1-1
- PDD
 - building with emake.exe, 3-14
- PLIB
 - hello world application example, 3-4
- PLIB & CLIB
 - contrasted, 3-4
- priorities
 - processes - remarks on, 5-14
- priority
 - application - control over, 3-12
- process priorities
 - remarks on, 5-14
- program
 - file transfer to SIBO system, 3-3
- program errors
 - panics explained, 2-3
- program size
 - application CLIB, 3-5
 - application PLIB, 3-5
- programming
 - asynchronous I/O, 5-2
 - console device I/O in general, 5-6
 - CPU yielding, 5-2
 - documentation sources, 5-1
 - error handling - a first look, 5-7
 - error handling - in events, 5-7
 - error handling - resource tidying, 5-8
 - events - prioritisation of sources, 5-5
 - events multiple source examples, 5-3
 - example of background processing, 5-2
 - example of keyboard input, 5-2
 - example of serial port data input, 5-2
 - example of sub-process completion, 5-2
 - example of timer expiry, 5-2
 - fundamental guide-lines, 5-1
 - I/O devices in general, 5-5
 - introduction - example source, 5-2
 - introduction to, 5-1
 - introduction to events - example apps, 5-2
 - key presses - remarks on, 5-4
 - multi-threaded, 5-2
 - p_iowait - remarks on, 5-4
 - p_iowait remarks on, 5-4
 - p_iowait status words - declaring, 5-7
 - p_iowait status words - remarks on, 5-5
 - p_ioyield - remarks on, 5-14
 - PC based development, 1-1
 - semaphore I/O, 5-4
 - timers cancelling in general, 5-6
 - timers remarks on, 5-4
 - user interface - simple examples, 5-2
- programs
 - .img files, 3-1
- project file
 - a first look at .pr files, 3-2
 - advanced use, 3-11
 - application project .pr files, 3-1
 - epocinit statement, 3-6
 - reusing - general, 3-6
- resource files
 - application .rsc files, 3-13
 - application .rzc compressed, 3-13
- screen sizes
 - detecting which SIBO system, 7-1
 - table of for SIBO systems, 7-1
- segment registers
 - small code model, 4-2
- semaphore
 - I/O remarks on, 5-4
- serial port
 - comms debugging, 5-13
 - data reception of, 5-11
 - opening, 5-12
- serial port data input
 - programming example, 5-2
- shell data files
 - application, 3-13
- SIBO
 - explained, 2-2
 - systems family, 1-1
- SIBO SDK
 - see C SDK, 2-1
- SIBO systems
 - screen size table, 7-1
 - software compatibility, 7-1
 - what machine, 7-1
- small code model
 - segment registers, 4-2
- small model code
 - C SDK, 1-1
- software
 - compatibility across SIBO systems, 7-1
 - copy protection introduction, 6-1
 - copy protection mechanism, 6-1
- stack size
 - default, 3-6
 - specifying, 3-6

- status words
 - declaring - `p_iowait`, 5-7
- `stdepoc.h`
 - include file, 3-11
- sub-process
 - contrasted with idle object, 5-16
- sub-process completion
 - programming example, 5-2
- `SY$NULL`
 - chance to run auto-switch-off, 5-14
- timer expiry
 - programming example, 5-2
- timers
 - cancelling in general, 5-6
 - remarks on, 5-4
- Topspeed
 - Integrated Development Environment, 3-1
- Topspeed C
 - Clarion Software, 1-1
 - library reference, 2-1
 - package, 1-1
- `ts.red` file
 - C SDK redirection file, 1-3
- TSC vs TSCX
 - contrasted, 3-8
- `tscfg`
 - configuration file compiler, 3-11
- `tsprj.txt`
 - configuration file, 3-11
- user interface
 - programming examples, 5-2
- utility program
 - `edump.exe`, 3-12
 - `emake.exe`, 3-14
 - `eremake.exe`, 3-14
- version number
 - application - control over, 3-12
- versions
 - C SDK, 1-1
- what machine
 - detecting the SIBO system, 7-1
- window redraw
 - event source - remarks on, 5-16
- window server
 - WLIB library introduction, 3-5
- WLIB
 - introduction, 3-5
- yielding CPU
 - programming, 5-2