Pacific
C
HI-TECH
SOFTWARE

Registration Card

Complete this card and return it to HI-TECH Software to ensure that you will receive technical support and access to updates. Please use block letters!

Your name: _____

Company name (if applicable): _____

Your postal address: _____

_____ Country: _____ Post/Zip code: _____

Phone:_____ Fax:_____ E-mail:_____

Software package: Pacific C for DOS Version: _____

Serial no: _____ Supplied by: _____

I hereby agree to the terms and conditions of the licence agreement for this software

Signed: ✘ _____

Unsigned registrations will not be accepted.

Fold here

Please help us to help you by answering the following questions. We will use this information to improve our products.

Please tick the operating systems you use:

- ❑ MS-DOS
- ❑ Windows 3.1 or 3.11 or WfWg
- ❑ Windows '95
- ❑ Windows NT
- ❑ OS/2
- ❑ Unix (specify platform)

_____

- ❑ Other (please specify)

_____

Please indicate programming languages you use

- ❑ C
- ❑ C++
- ❑ Objective-C
- ❑ Eiffel
- ❑ Pascal
- ❑ Modula-2
- ❑ Modula-3
- ❑ Simula
- ❑ Smalltalk
- ❑ Forth
- ❑ Assembler
- ❑ Other (please specify)

_____

If you program embedded systems, please list the processors you use

_____
_____
_____
_____
_____
_____
_____
_____
_____
_____

Thank you!

HI-TECH SOFTWARE

Tape here

Please attach correct postage

HI-TECH Software
PO Box 103
ALDERLEY QLD 4051
AUSTRALIA

Introduction 1

Quick Start Guide 2

Using PPD 3

Runtime Organization 4

8086 Assembler Reference Manual 5

Lucifer- A Source Level Debugger 6

Rom Development with Pacific C 7

Linker Reference Manual 8

Librarian Reference Manual 9

Error Messages A

Library Functions B

YOU SHOULD CAREFULLY READ THE FOLLOWING BEFORE INSTALLING OR USING THIS SOFTWARE PACKAGE. IF YOU DO NOT ACCEPT THE TERMS AND CONDITIONS BELOW YOU SHOULD IMMEDIATELY RETURN THE ENTIRE PACKAGE TO YOUR SUPPLIER AND YOUR MONEY WILL BE REFUNDED. USE OF THE SOFTWARE INDICATES YOUR ACCEPTANCE OF THESE CONDITIONS.

To ensure that you receive the benefit of the warranty described below, you should complete and sign the accompanying registration card and return it to HI-TECH Software immediately.

SOFTWARE LICENCE AGREEMENT

HI-TECH Software, a division of Gretetoy Pty. Ltd., of 41 Yarrabee Rd., The Gap, QLD 4061 Australia, provides this software package for use on the following terms and conditions:

This software package is fully copyrighted by HI-TECH Software and remains the property of HI-TECH Software at all times.

You may:

- Use this software package on a single computer system. You may transfer this package from one computer system to another provided you only use it on one computer system at a time.
- Make copies of diskettes supplied with the software package for backup purposes provided all copies are labelled with the name of the software package and carry HI-TECH Software's copyright notice.
- Use the software package to create your own software programs. Provided such programs do not contain any part of this software package other than extracts from any object libraries included then these programs will remain your property and will not be covered by this agreement.
- Transfer the software package and this licence to a third party provided that the third party agrees to the terms and conditions of this licence, and that all copies of the software package are transferred to the third party or destroyed. The third party must advise HI-TECH Software that they have accepted the terms and conditions of this licence.

You may NOT:

- Sell, lend, give away or in any way transfer copies of this software package to any other person or entity except as provided above, nor allow any other person to make copies of this software package.
- Incorporate any portion of this software package in your own programs, except for the incorporation in executable form only of extracts from any object libraries.
- Use this package to develop life-support applications or any application where failure of the application could result in death or injury to any person. Should you use this software to develop any such application, you agree to take all responsibility for any such failures, and indemnify HI-TECH Software against any and all claims arising from any such failures.

TERM

This licence is effective until terminated. You may terminate it by returning to HI-TECH Software or destroying all copies of the software package. It will also terminate if you fail to comply with any of the above conditions.

WARRANTY

HI-TECH Software warrants that it has the right to grant you this licence and that the software package is not subject to copyright to which HI-TECH Software is not entitled. Certain State and Federal laws may provide for warranties additional to the above.

LIMITATION OF LIABILITY

This software package has been supplied in good faith and is believed to be of the highest quality. Due to the nature of the software development process, it is possible that there are hidden defects in the software which may affect its use, or the operation of any software or device developed with this package. You accept all responsibility for determining whether this package is suitable for your application, and for ensuring the correct operation of your application software and hardware. HI-TECH Software's sole and maximum liability for any defects in this package is limited to the amount you have paid for the licence to use this software. HI-TECH Software will not be liable for any consequential damages under any circumstances, unless such exclusion is forbidden by law.

Trade Marks

The following are trade marks of HI-TECH Software:

Pacific C; HI-TECH C; Lucifer; PPD; HPD

Other trade marks and registered trade marks used in this document are the property of their respective owners.

Technical Support

For technical support on this software product contact your reseller. HI-TECH Software may be contacted as follows, but note that HI-TECH Software will not necessarily provide technical support unless you have a current technical support agreement in place.

	Within Australia	***From elsewhere***
Fax	07 3354 2411	+61 7 3354 2422
Phone	07 3354 2411	+61 7 3354 2411
Electronic mail	support@htsoft.com	support@htsoft.com
World Wide Web	http://www.htsoft.com	http://www.htsoft.com

There is also a mailing list you may join which carries discussion of Pacific C. For information or to join this list, send electronic mail to pacific-request@htsoft.com. You will get an automatic reply with information on subscribing.

HI-TECH Software
a division of Gretetoy Pty. Ltd.
PO Box 103, Alderley
QLD 4051 Australia

Contents

1 - Introduction

1.1 The Pacific C Compiler 1
1.2 Installation 1
1.2.1 INSTALL Program 1
1.2.1.1 Installation Steps 1
1.2.1.2 Custom Installation 2
1.2.1.3 Serial Number and Installation Key 2
1.2.1.4 Accessing the Compiler 3

2 - Quick Start Guide

2.1 Getting started 5
2.2 A Sample Program 5
2.3 Using PPD 5
2.4 Using PACC 6

3 - Using PPD

3.1 Introduction 7
3.1.1 Typographic Conventions 7
3.1.2 Starting PPD 7
3.2 The HI-TECH Windows User Interface 8
3.2.1 Hardware Requirements 8
3.2.2 Pull-down Menus 9
3.2.2.1 Keyboard Menu Selection 10
3.2.2.2 Mouse Menu Selection 10
3.2.2.3 Menu Hot Keys 10
3.2.3 Selecting Windows 11
3.2.4 Moving and Resizing Windows 11
3.2.5 Buttons 13
3.2.6 The Setup Menu 14
3.3 Tutorial: Creating and Compiling a C Program 14
3.4 The PPD Editor 20
3.4.1 Status Line 20
3.4.2 Keyboard Commands 21
3.4.3 Block Commands 21
3.4.4 Clipboard Editing 23

3.4.4.1 Selecting Text 23
3.4.4.2 Clipboard Commands 24
3.5 PPD Menus 24
3.5.1 <<>> Menu 26
3.5.2 File Menu 27
3.5.3 Edit Menu 28
3.5.4 Options Menu 30
3.5.5 Compile Menu 32
3.5.6 Make Menu 33
3.5.7 Run Menu 37
3.5.8 Utility Menu 39
3.5.9 Help Menu 42
4 - Runtime Organization
4.1 Memory Models 43
4.2 Runtime Startup 43
4.3 Stack Frame Organization 43
4.4 Prototyped Function Arguments 45
4.5 Stack and Heap Allocation 45
4.6 Linkage to Assembler 45
4.6.1 Assembler Interface Example 46
4.6.2 Signatures 47
4.7 Data Representation 48
4.7.1 Longs 48
4.7.2 Pointers 48
4.7.3 Floats 48
4.8 Linking 8086 Programs 48
4.8.1 Terms 49
4.8.2 Link and Load Addresses 49
4.8.3 Segmentation 49

4.8.4 Link Addresses 49
4.8.5 Load Addresses 50
4.8.6 Linking and the -P option 50
5 - 8086 Assembler Reference Manual
5.1 Introduction 55
5.2 Usage 55
5.2.1 -Q 55
5.2.2 -U 55
5.2.3 -O*file* 56
5.2.4 -L*list* 56
5.2.5 -W*width* 56
5.2.6 -X 56
5.2.7 -E 56
5.2.8 -M 56
5.2.9 -1,-2,-3,-4 56
5.2.10 -N 57
5.2.11 -I 57
5.2.12 -S 57
5.2.13 -C 57
5.3 The Assembly Language 57
5.3.1 Symbols 57
5.3.1.1 Temporary Labels 58
5.3.2 Constants 59
5.3.2.1 Character Constants 59
5.3.2.2 Floating Constants 59
5.3.3 Expressions 59
5.3.3.1 Operators 59
5.3.3.2 Relocatability 59
5.3.4 Pseudo-ops 61
5.3.4.1 .BYTE 61
5.3.4.2 .WORD 61
5.3.4.3 .DWORD 61

5.3.4.4 .FLOAT 62
5.3.4.5 .BLKB 62
5.3.4.6 EQU 62
5.3.4.7 SET 62
5.3.4.8 .END 62
5.3.4.9 IF 62
5.3.4.10 .ENDIF 63
5.3.4.11 .ENDM 63
5.3.4.12 .PSECT 63
5.3.4.13 .GLOBL 65
5.3.4.14 .LOC 65
5.3.4.15 .MACRO and .ENDM 65
5.3.4.16 .LOCAL 66
5.3.4.17 .REPT 67
5.3.4.18 .IRP and .IRPC 67
5.3.4.19 Macro Invocations 68
5.3.4.20 .TITLE 68
5.3.4.21 .SUBTTL 68
5.3.4.22 .PAGE 68
5.3.4.23 .INCLUDE 68
5.3.4.24 .LIST 69
5.3.5 Addressing Modes 69
5.3.5.1 Register 69
5.3.5.2 Immediate 69
5.3.5.3 Direct 70
5.3.5.4 Register indirect 70
5.3.5.5 Indexed 70
5.3.5.6 String 70

5.3.5.7 I/O 71
5.3.5.8 Segment prefixes 71
5.3.5.9 Jump 71
5.3.6 80386 Addressing Modes 71
5.3.6.1 Segment Registers 71
5.3.6.2 String 72
5.3.6.3 Single register 72
5.3.6.4 Double indexed 72
5.3.6.5 Scaled index 72
5.4 Diagnostics 72

6 - Lucifer - A Source Level Debugger

6.1 Introduction 77
6.2 Usage 77
6.3 Commands 78
6.3.1 The A Command: set command line arguments 78
6.3.2 The B Command: set or display breakpoints 79
6.3.3 The D Command: display memory contents 79
6.3.4 The E Command: examine C source code 79
6.3.5 The G Command: commence execution 80
6.3.6 The I Command: toggle instruction trace mode 80
6.3.7 The Q Command: exit to DOS 81
6.3.8 The R Command: remove breakpoints 81
6.3.9 The S Command: step one C line 81
6.3.10 The T Command: trace one instruction 82
6.3.11 The U Command: disassemble 82
6.3.12 The X Command: examine or change registers 83
6.3.13 The @ Command: display memory as a C type 83
6.3.14 The ^ Command: print a stack trace 84
6.3.15 The $ Command: reset to initial configuration 85
6.3.16 The . Command: set a breakpoint and go 85
6.3.17 The ; Command: display from a source line 85

6.3.18 The = Command: display next page of source 85
6.3.19 The - Command: display previous page of source 85
6.3.20 The / Command: search source file for a string 86
6.3.21 The ! Command: execute a DOS command 86
6.3.22 Other Commands 86
7 - ROM Development with Pacific C
7.1 Requirements for ROM Development 87
7.2 ROM Code vs. DOS .EXE Files 87
7.3 What You Need to Know 88
7.4 First Steps 90
7.4.1 Entering the Program 91
7.4.2 Selecting Processor Type 91
7.4.3 Choosing the File Type 93
7.4.4 Setting Memory Addresses 93
7.4.5 Selecting Optimizations 94
7.4.6 Entering Source Files 94
7.4.7 Making the Program 95
7.4.8 Running the Code 95
7.5 Going Further 96
7.6 Implementing Lucifer 96
7.7 Debugging with Lucifer 97
8 - Linker Reference Manual
8.1 Introduction 99
8.2 Relocation and Psects 99
8.3 Program Sections 100
8.4 Local Psects 100
8.5 Global Symbols 100
8.6 Link and load addresses 100
8.7 Operation 101

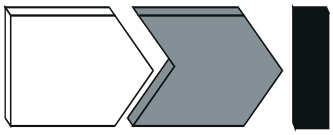
8.7.1 Numbers in linker options 101
8.7.2 -8 101
8.7.3 -A***class=low-high,...*** 101
8.7.4 -C***psect=class*** 102
8.7.5 -D*symfile* 102
8.7.6 -F 103
8.7.7 -G*spec* 103
8.7.8 -H*symfile* 103
8.7.9 -I 103
8.7.10 -L 104
8.7.11 -LM 104
8.7.12 -M*mapfile* 104
8.7.13 -N 104
8.7.14 -O*outfile* 104
8.7.15 -P*spec* 104
8.7.16 -S*psect=max* 107
8.7.17 -U*symbol* 107
8.7.18 -V*avmap* 107
8.7.19 -W*num* 107
8.7.20 -X 107
8.7.21 -Z 107
8.8 Invoking the Linker 107

9 - Librarian Reference Manual

9.1 Introduction 109
9.2 Usage 109
9.2.1 Creating a library 109
9.2.2 Updating a library 110
9.2.3 Deleting library modules 110
9.2.4 Extracting a module 110
9.2.5 Listing a library 110
9.3 Library ordering 110
9.4 Creating libraries from PPD 111

Appendix A - Error Messages 113
Appendix B - Library Functions 151

Introduction



1.1 The Pacific C Compiler

This manual covers the Pacific C compiler for MS-DOS from HI-TECH Software. In this manual you will find information on installing, using and customising the compiler.

Pacific C requires an 80286 processor with at least 512K of free conventional memory, and a hard disk, It will also run on 80386 and 80486 processors. MS-DOS 3.1 or later is required, we recommend MS-DOS 3.3 or later, or DRDOS 6.0 or later. We strongly recommend you have at least 1MB of free XMS memory (from HIMEM.SYS). A mouse is not required, but is strongly recommended

1.2 Installation

Pacific C is supplied on one or more 3.5" or 5.25" diskettes. The contents of the disks are listed in a file called **PACKING.LST** on disk 1. To install the compiler, you must use the **INSTALL** program on disk 1. This is located in the root directory of disk 1. Place disk 1 in either floppy drive, then type **A:INSTALL** or **B:INSTALL** as appropriate. The **INSTALL** program will then present a series of messages and ask for various information as it proceeds. You may need to check your environment variables (use the **SET** command) in case you have an environment variable called **TEMP** set. If this variable is set, its value should a directory path. The full path must exist and designate a directory in which temporary files can be created.

1.2.1 INSTALL Program

The **INSTALL** program uses several on-screen windows. There is a message window, in which it displays messages and prompts. There is also a one-line status window in which **INSTALL** says what it is currently doing. Other windows will pop up from time to time. A dialog or alert window will pop up when **INSTALL** wants user action, or when any kind of error occurs, and will offer the opportunity to continue, retry (if appropriate) or terminate. If you select **TERMINATE**, the installation will be incomplete. A sliding bar will indicate the approximate degree of completion of the installation.

1.2.1.1 Installation Steps

When **INSTALL** prompts for action, you may either press **ENTER** to continue, **ESCAPE** to terminate, or use a mouse if you have one to select the buttons displayed in the dialog window. To use a mouse, move the cursor into the desired button, then press and release the left mouse button.

Initially **INSTALL** will simply advise it is about to install a HI-TECH Software package. Select **CONTINUE** or press **ENTER**. You will then be asked to choose between a full, no questions asked installation, or a custom installation in which you may choose not to install optional parts of the compiler. The custom installation will also allow you to specify the directories in which the compiler is installed

1.2.1.2 Custom Installation

If you selected a custom installation, **INSTALL** then asks you a series of questions about directory paths for installation of the compiler. At each one it will display a default path and ask you to press **ENTER** to confirm that path, or enter a new path then press **ENTER**. Again you may select **TERMINATE** if you do not want to continue. Note that **INSTALL** will create any directory necessary, except that it will NOT create intermediate directories, e.g. it will create the directory **C:\COMPILE\HITECH** if it does not exist, but it will not create the **COMPILE** directory in this case. It must already exist.

INSTALL also asks for a temporary file directory. This is a directory in which the compiler will place temporary files, and should be a RAM disk if you have one (but ensure the RAM disk is reasonably large - at least several hundred Kbytes). Otherwise it may be a directory on your hard disk or simply blank. If it is blank the compiler will create temporary files in the working directory.

Next **INSTALL** asks a series of questions about installation of optional parts of the compiler. For each part you may answer yes (**ENTER**) or no (**F10**) or use the mouse to click the appropriate button.

1.2.1.3 Serial Number and Installation Key

After these questions have been answered, or immediately if you selected a full installation, **INSTALL** will ask you to enter the serial number and installation key. You will also be asked to enter your name and your company's name (the company name is optional). **INSTALL** will serialise the installed compiler with this information. The serial number and installation key are found on the reverse of the manual title page. The serial number and key must be entered correctly or the installation will not proceed.

After this **INSTALL** proceeds to copy (decompressing as it goes) the files that are contained in the basic compiler and the optional parts. Each file being copied will be displayed in the status window. If **INSTALL** discovers it is copying a file that already exists, i.e. it is going to overwrite a file, it will pop up a dialog window asking if you want to overwrite the file. If you answer **YES** the first time this occurs, you will be asked if you want to be prompted about any other overwritten files. Answering **NO** will cause install to silently overwrite any other files that already exist. This would be in order if you were reinstalling or installing an updated version.

During the copying process, **INSTALL** may bring up a window in the middle of the screen containing informative text from a file on the distribution disk. This will contain information about the compiler and other HI-TECH Software packages. While reading this information, you may use the up and down arrow keys and the Page-up and Page-down keys to scroll the text.

INSTALL will bring up a dialog window whenever you need to change disks. When this occurs, place the requested disk in the floppy drive and press **ENTER**. On completion of the installation, it will if necessary edit your **AUTOEXEC.BAT** and **CONFIG.SYS** files. When it does so, it will bring up two edit windows containing the new and old versions of the file. You may scroll the windows and compare to see what changes have been made, and edit the new version if you desire. When done, press **F1** or click in the **DONE** button in the status window. Pressing **ESC** or clicking in the **ABORT** window will prevent **INSTALL** from updating the file. This step will not occur if your **AUTOEXEC.BAT** does not need modification.

After this **INSTALL** will display some messages, then advise you to press **ENTER** to read the release notes for the compiler. This will load the file **READ.ME** (which will also be copied onto your hard disk) in the screen window and allow you to read it. Pressing **ENTER** again will exit to DOS.

At this stage you may need to reboot to allow the changes to **AUTOEXEC.BAT** to take effect. **INSTALL** will have told you to do this if it is necessary. The installation is now complete.

1.2.1.4 Accessing the Compiler

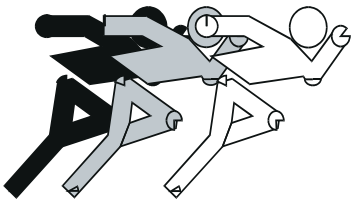
The installation process will include in your **PATH** environment variable the directory containing the compiler executable programs. However, it is possible that some other programs already installed on your system have the same name as one of the compiler programs. The most common of these is **LINK**. To overcome this you may need to re-organise your PATH environment variable.

The compiler drivers are **PACC.EXE** (command line version) and **PPD.EXE** (integrated version). These are basically the only commands you need to access the compiler, but you may also want to run other utilities directly.

Chapter 1 - Introduction

1

Quick Start Guide



2.1 Getting started

For new users of the Pacific C compiler, the following section gives a step-by-step guide to getting a first program running with Pacific C.

2.2 A Sample Program

```
/*
#include <stdio.h>

main()
{
        printf("Hello, world!\n");
}
```

2.3 Using PPD

You will find a complete guide to using PPD in the section "PPD User's Guide" (page). To enter this program, simply follow these steps:

- Start PPD by typing its name, then pressing ENTER. If you have installed PPD properly, it will be in your search path. You should have on screen a menu bar, a large Edit window, and a smaller Message window.
- Start typing the program text in the edit window. The editor command keys allow either the standard PC keys (arrow keys etc.) or WordStar-compatible keystrokes.
- After typing the complete program (with any modifications necessary for your hardware) press ALT-S. A dialog box will appear asking you to enter a name to save the file. Type the name HELLO.C and press RETURN. The file will be saved to disk.
- Press F3 to compile the program. PPD will compile the program. Any errors found will stop the compilation, and the errors will be listed in a window that appears at the bottom of the screen. The cursor in the edit window will be positioned on the error line. Correct the error, then press F3 again. You will not have to re-enter the memory addresses. On completion of compilation, an output file called HELLO.EXE will be left in the current directory.

- Press F7 to run the program. PPD will revert to a DOS text screen and run your compiled program. You will see the words “Hello, world!” appear, followed by “Press any key to return...”. If you then press a key, you will be returned to PPD. To exit PPD, press ALT-Q.

2.4 Using PACC

To use PACC to compile your sample program, you will first need to create a file containing the program. You should use whatever text editor you are used to using, as long as it can create a plain ASCII file. The DOS EDIT command is satisfactory. Call the file HELLO.C. Then run PACC thus

PACC HELLO.C

If you have correctly entered the sample program, no error messages should result. If you do get error messages, edit the program to correct them, and recompile with PACC as before. After a successful compilation, PACC will print a memory usage summary. Then run your program by typing the command

HELLO

This will load and run HELLO.EXE. You should see the words “Hello, world!” appear.

Using PPD

3.1 Introduction

This chapter covers PPD, the **P**acific C **P**rogrammer's **D**evelopment system integrated environment.It assumes that you already have an installed Pacific C compiler, if you haven't installed your compiler go to chapter 1, INTRODUCTION, and follow the installation instructions there.

3.1.1 Typographic Conventions

Throughout this chapter, we will adopt the convention that any text which you need to type will be printed in **bold type**. The computer's prompts and responses will be printed in `constant spaced type`. Particularly useful points and new terms will be illustrated using *italicised type*. With a window based program like PPD, some concepts are difficult to convey in print and will be introduced using short tutorials and sample screen displays.

3.1.2 Starting PPD

To start PPD, simply type PPD at the DOS prompt and, after a brief period of disk activity you will be presented with a screen similar to the one shown in figure 3-1.

The initial PPD screen is broken up into three windows which, from the top, contain the menu bar, the PPD text editor and the message window. Other windows may appear when certain menu items are selected, however the editing screen is what you will use most of the time.

PPD uses the HI-TECH Windows user interface to provide a text screen based user interface with multiple overlapping windows and pull down menus. Later in this chapter are described the user interface features which are common to all HI-TECH Windows applications.

PPD can optionally take a single command line argument which is either the name of a text file, or the name of a *project file*. (Project files are discussed in a later section of this chapter). If the argument has an extension .PRJ, PPD will attempt to load a project file of that name. File names with any other extension will be treated as text files and loaded by the editor. If an argument without an extension is given, PPD will first attempt to load a .PRJ file, then a .C file. For example, if the current directory contains a file called X.C and PPD is invoked with the command PPD X, it will first attempt to load X.PRJ and when that fails, will load X.C into the editor. If no source file is loaded into the editor, an empty file with name "untitled" will be started.

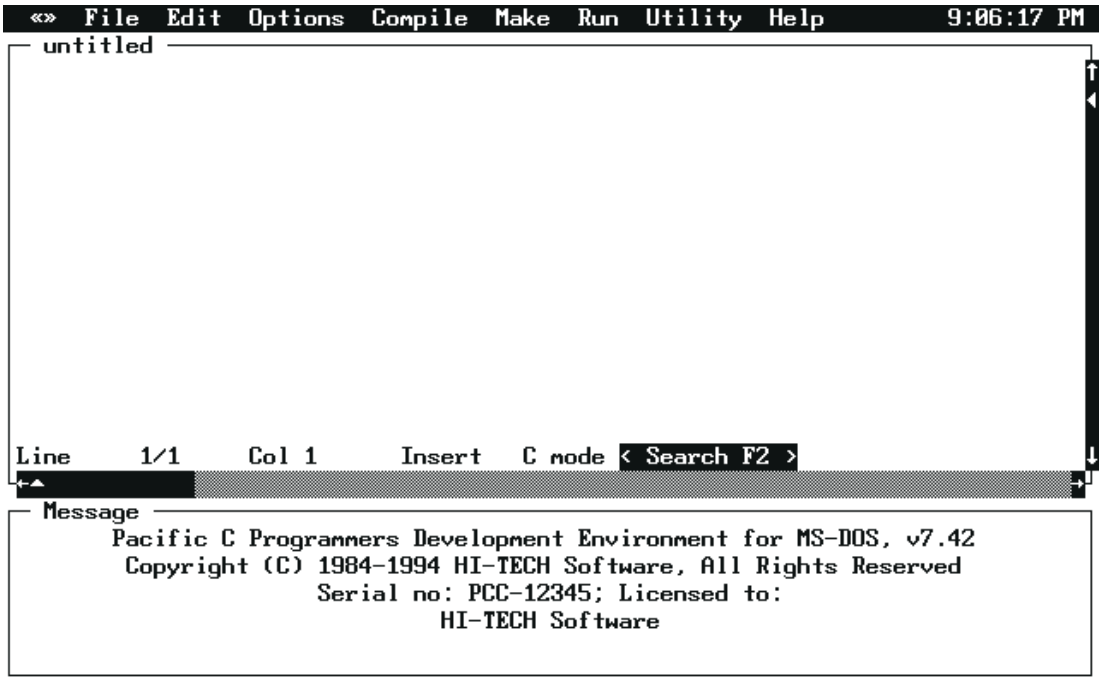


Figure 3-1; PPD Startup screen

3.2 The HI-TECH Windows User Interface

The HI-TECH Windows user interface used by PPD provides a powerful text screen based user interface which can be used either by keyboard alone, or with a combination of keyboard and mouse operations. For new users most operations will be simpler using the mouse, however as experience with the package is gained *hot-key* sequences for most commonly used functions will be learned.

3.2.1 Hardware Requirements

HI-TECH Windows based applications will run on any MS-DOS based machine with a standard display card capable of supporting text screens of 80 columns by 25 rows or more. Higher resolution text modes like the EGA 80 x 43 mode will be recognised and used if the mode has already been selected before

PPD is executed, or with a `/screen:xx` option as described below. Problems may be experienced with some poorly written VGA utilities which initialize the hardware to a higher resolution mode but leave the BIOS data area in low memory set to the values for an 80 x 25 display.

It is also possible to have PPD set the screen display mode on EGA and VGA displays to show more than 25 lines. The option **/SCREEN:nn** where **nn** is one of 25, 28, 43 or 50 will cause PPD to set the display to that number of lines, or as close as possible. EGA displays support only 25 and 43 line text screens, while VGA supports 28 and 50 lines as well.

The display will be restored to the previous mode after PPD exits. The selected number of lines will be saved in PPD.INI and used for subsequent invocations of PPD unless overridden by another **/SCREEN** option.

PPD will recognize and use any mouse driver which supports the standard INT 33H interface. Almost all modern mouse drivers support the standard device driver interface, however some older mouse drivers are missing a number of the driver status calls. If you are using such a mouse driver, PPD will still work with the mouse, but the **Mouse Setup** dialogue in the <<>> menu will not work.

3.2.2 Pull-down Menus

HI-TECH Windows includes a system of *pull-down menus* which are based around a *menu bar* across the top of the screen. The menu bar will be broken into a series of words or symbols, each of which is the title of a single pull-down menu.

The menu system can by used by keyboard, with the mouse, or with a combination of mouse and keyboard actions. The keyboard and mouse actions listed in table 3-1 are supported:

Table 3-1; PPD keyboard and mouse actions

Action	Key	Mouse
Open menu	alt-Space	Press left button in menu bar or press middle button anywhere on screen
Escape from menu	alt-Space or Escape	Press left button outside menu system
Select item	Enter	Release left or centre button on highlighted item
		Click left or centre button on an item
Next menu	right arrow	Drag to right
Previous menu	left arrow	Drag to left
Next item	down arrow	Drag downwards
Previous item	up arrow	Drag upwards

3.2.2.1 Keyboard Menu Selection

Selecting a menu item by keyboard is accomplished by pressing **alt-Space** to open the menu system, then using the arrow keys to move to the desired menu and highlight the item required. When the item required is highlighted it can be selected by pressing **Enter**. Some menu items will displayed with lower intensity or a different colour and are not be selectable. These items are disabled because their selection is not appropriate within the current context of the application. To give an example, the **Save project** item will not be selectable if no project has been loaded or defined.

3.2.2.2 Mouse Menu Selection

Selecting a menu item using the mouse appears to be somewhat awkward to new users, however it soon becomes second nature with experience. The menu system is opened by moving the pointer to the title of the menu which is desired and pressing the left button. It is possible to browse through the menu system by holding the left button down and dragging the mouse across the titles of several menus, opening each in turn. The menu system may also be operated with the middle button on three button mice. Pressing the middle button brings the menu bar to the front, making it selectable even if it is completely hidden by a zoomed window.

Once a menu has been opened, two styles of selection are possible. If the left or middle button is released while no menu item is highlighted, the menu will be left open and selection can be made using the keyboard or by moving the pointer to the desired menu item and clicking the left or middle mouse button. If the mouse button is left down after the menu is opened, a selection can be made by dragging the mouse to the desired item and releasing the button.

3.2.2.3 Menu Hot Keys

When browsing through the menu system you will notice that some menu items have *hot key* sequences displayed. For example, the PPD menu item **Save** has the key sequence **alt-S** displayed as part of the item. When a menu item has a key equivalent, it can be selected directly by pressing that key without opening the menu system. Key equivalents will be either **alt-***alphanumeric* keys or function keys. Where function keys are used, different but related menu items will commonly be grouped on the one key. For example, in PPD **F3** is assigned to **Compile and Link**, **shift-F3** is assigned to **Compile to .OBJ** and **ctrl-F3** is assigned to **Compile to .AS**.

Key equivalents are also assigned to entire menus, providing a convenient method of going to a particular menu with a single keystroke. The key assigned will usually be **alt** and the first letter of the menu name, for example **alt-E** for the **Edit** menu. The menu key equivalents are distinguished by being highlighted in a different colour (except on monochrome displays) and are highlighted with inverse video when the **alt** key is depressed.

A full list of PPD key equivalents is shown in table 3-2.

3.2.3 Selecting Windows

HI-TECH Windows allows windows to be overlapped or tiled under user control. If using the keyboard, you can bring a window to the front by pressing **ctrl-Enter** one or more times. Each time **ctrl-Enter** is pressed, the rearmost window is brought to the front and each other window on screen shuffles one level towards the back. A series of **ctrl-Enter** presses will cycle endlessly through the window hierarchy.

If you are using the mouse, you can bring any visible window to the front by pressing the left button in its content region[1]. A window can be made rearmost by holding the **alt** key down and pressing the left button in its content region. If a window is completely hidden by other windows, it can usually be located either by pressing **ctrl-Enter** a few times or by moving other windows to the back with **alt-left-button**.

Some windows will not come to the front when the left button is pressed in them. These windows have a special attribute set by the application and are usually made that way for a good reason. To give an example, the PPD compiler error window will not be made front most if it is clicked, instead it will accept the click as if it were already the front window. This allows the mouse to be used to select the compiler errors listed while leaving the editor window front most so the program text can be altered.

3.2.4 Moving and Resizing Windows

Most windows can be moved and resized by the user. There is nothing on screen to distinguish windows which cannot be moved or resized, if you attempt to move or resize and window and nothing happens, it indicates that the window cannot be resized. Some windows can be moved but not resized, usually because their contents are of a fixed size and resizing the window would not make sense. The PPD calculator is an example of a window which can be moved but not resized.

Windows can be moved and resized either using the keyboard or the mouse. When using the keyboard, move/resize mode can be entered by pressing **ctrl-alt-space**. The application will respond by replacing the menu bar with the move/resize menu ( ), allowing the front most window to be moved and resized.

The allowable actions in move/resize mode are as follows:

Move/resize mode can also be exited with any normal application action, like a mouse click, hot key or menu system activation with **alt-space**.

Windows can also be moved and resized using the mouse. Any visible window can be moved by pressing the left mouse button on its frame, dragging it to a new position and releasing the button. If a window is “grabbed” near one of its corners the pointer will change to a diamond and it will be possible to move the window in any direction, including diagonally. If a window is grabbed near the middle of the top or

1 * Pressing the left button in a window frame has a completely different effect, as discussed later in this chapter.

Table 3-2; PPD menu hot keys

Key	Meaning
alt-O	Open editor file
alt-N	Clear editor file
alt-S	Save editor file
alt-A	Save editor file with new name
alt-Q	Quit to DOS
alt-F	Open File menu
alt-E	Open Edit menu
alt-P	Open project file
alt-I	Open Compile menu
alt-M	Open Make menu
alt-R	Open Run menu
alt-H	Open Help menu
alt-U	Open Utility menu
alt-W	Warning level dialog
alt-Z	Optimisation menu
alt-D	DOS command
alt-J	Run COMMAND.COM
alt-T	Open Options menu
alt-L	Download code via Lucifer
alt-B	Run Lucifer debugger
F3	Compile and link single file
shift-F3	Compile to object file
ctrl-F3	Compile to assembler code
F5	Make target program
ctrl-F5	Re-make all objects and target program
shift-F5	Re-link target program
alt-P	Load project file
shift-F7	User defined command 1
shift-F8	User defined command 2
shift-F9	User defined command 3
shift-F10	User defined command 4
F2	Search in edit window

Key	Meaning
shift-F2	Search again, forward
ctrl-F2	Search again, backward
alt-X	Cut to clipboard
alt-C	Copy to clipboard
alt-V	Paste from clipboard
alt-G	Goto line number
alt-T	Set tab size

bottom edge the pointer will change to a vertical arrow and it will only be possible to move the window vertically. If a window is grabbed near the middle of the left or right edge the pointer will change to a horizontal arrow and it will only be possible to move the window horizontally.

If a window has a *scroll bar* in its frame, pressing the left mouse button in the scroll bar will not move the window, but will instead activate the scroll bar, sending scroll messages to the application. If you want to move a window which has a frame scroll bar, just select a different part of the frame.

Windows can be resized using the right mouse button. Any visible window can be resized by pressing the right mouse button on its bottom or left frame, dragging the frame to a new boundary and releasing the button. If a window is grabbed near its lower right corner the pointer will change to a diamond and it will be possible to resize the window in any direction. If the frame is grabbed anywhere else on the bottom edge, it will only be possible to resize vertically, likewise if the window is grabbed anywhere else on the right edge it will only be possible to resize horizontally. If the right button is pressed anywhere in the top or left edges nothing will happen.

It is also possible to *zoom* a window to its maximum size. The front most window can be zoomed by pressing **shift-(keypad)+**, if it is zoomed again it will revert to its former size. In either the zoomed or unzoomed state the window can be moved and resized, thus zoom effectively toggles between two user defined sizes. A window can also be zoomed by clicking the right mouse button in its content region.

3.2.5 Buttons

Some windows contain *buttons* which can be used to select particular actions immediately. Buttons are like menu items which are always visible and selectable. A button can be selected either by clicking the left mouse button on it or by using its key equivalent. The key equivalent to a button will either be displayed as part of the button, or as part of a help message somewhere else in the window. For example, the PPD error window ( ) contains a number of buttons, to select EXPLAIN you would either click the left mouse button on it or press **F1**.

Table 3-3; Resize keys

Key	Action
left arrow	Move window to left
right arrow	Move window to right
up arrow	Move window upwards
down arrow	Move window downards
shift-left arrow	Shrink window horizontally
shift-right arrow	Expand window horizontally
shift-up arrow	Shrink window vertically
shift-down arrow	Expand window vertically
Enter or Escape	Exit move/resize mode

3.2.6 The Setup Menu

If you open the system menu, identfified by the symbol <<>> on the menu bar, you will find two entries; an About entry which will display information about the version number of PPD, and a Setup entry. Selecting the Setup entry will open a dialogue box as shown in figure 3-6. This box both displays information about PPD's memory usave, and allows you to set the mouse sensitivity, whether the time of day is displayed in the menu bar, and whether sound is used. After changing mouse sensitivity values, you can test them by clicking on the Test button. This will change the mouse values so you can test the altered sensitivity. If you subsequently click Cancel, they will be restored to the previous values. Selecting OK will confirm the altered values, and save them in PPD's initialisation file, so they will be reloaded next time you run PPD.

Similarly, the sound and clock settings will be stored in the initialisation file if you select OK.

3.3 Tutorial: Creating and Compiling a C Program

This tutorial should be sufficient to get you started using PPD, it does not attempt to give you a comprehensive tour of PPD's features, that is left to the reference section of this chapter. If you have not used a HI-TECH Windows based application before, we strongly suggest that you complete this tutorial even if you are an experienced C programmer.

Before starting PPD, we first need to create a work directory. Make sure you are logged to the root directory on your hard disk and type the following commands:

```
C:\> md tutorial
C:\> cd tutorial
C:\> TUTORIAL> PPD
```

You will be presented with the PPD startup screen as discussed before. At this stage the editor is ready to accept whatever text you type. A flashing block cursor should be visible in the top left corner of the edit window. You are now ready to enter your first C program using PPD, which will naturally be the infamous “hello world” program.

Type the following text, pressing enter once at the end of each line. Blank lines may be added by pressing enter without typing any text.

```
#include <stdio.h>

/* infamous hello world program ! */

main()
{
        printf(“Hello, world”)
}
```

Note that a semicolon has been deliberately omitted from the end of the cputs() statement in order to demonstrate PPD’s error handling facilities. shows the screen as it should appear after entry of the “hello world” program.

We now have a C program (complete with one error!) entered and almost ready for compilation, all we need to do is save it to a disk file and then invoke the compiler. In order to save your source code to disk, you will need to select the **Save** item from the **File** menu (figure 3-3).

If you do not have a mouse, follow these steps: 1) Open the menu system by pressing **alt-Space**. 2) Move to the **Edit** menu using the **right arrow** key. 3) Move down to the **Save** item using the **down arrow** key. 4) When the **Save** item is highlighted, select it by pressing the **Enter** key.

If you are using the mouse, follow these steps: 1) Open the **File** menu by moving the pointer to the word **File** in the menu bar and pressing the left button. 2) Highlight the **Save** item by dragging the mouse downwards with the left button held down, until the **Save** item is highlighted. 3) When the **Save** item is highlighted, select it by releasing the left button.

When the **File** menu was open, you may have noticed that the **Save** item included the text **alt-S** (figure 3-3) at the right edge of the menu. This indicates that the save command can also be accessed directly using the *hot-key* command **alt-S**. A number of the most commonly used menu commands have hot-key equivalents which will either be **alt-***alphanumeric* sequences or function keys.

After **Save** has been selected, you should be presented with a *dialogue* (figure 3-4) prompting you for the file name. If PPD needs more information, such as a file name, before it is able to act on a command, it will always prompt you with a standard dialogue like the one below.

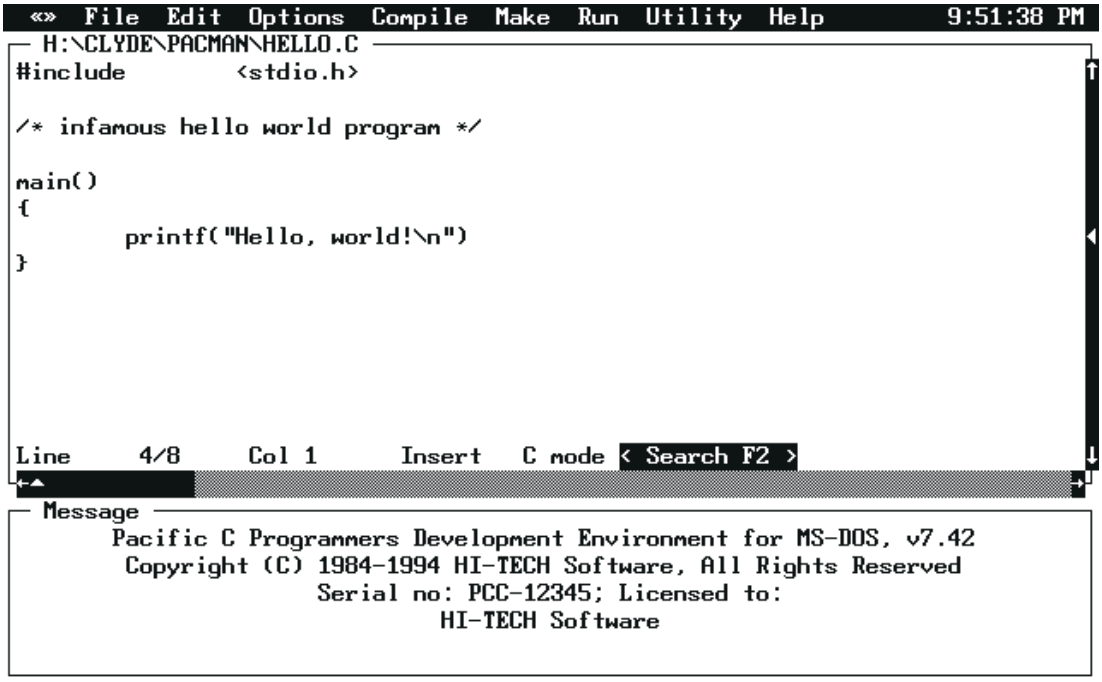


Figure 3-2; Hello proram in PPD

The dialogue is broken up into an *edit line* where you can enter the filename to be used, and a number of *buttons* which may be used to perform various actions within the dialogue. A button may be selected either by clicking the left mouse button with the pointer positioned on it, or by using its key equivalent. The text in the edit line may be edited using the standard editing keys: **left arrow**, **right arrow**, **backspace**,**del** and **ins**. **Ins** toggles the line editor between insert and overwrite mode.

In this case, we wish to save our C program to a file called “hello.c”. Type **hello.c** and then press **Enter**. There should be a brief period of disk activity and then PPD will respond with Saved hello.c in the message window.

We are now ready to actually compile the program. To compile and link in a single step, select the **Compile and link** item from the **Compile** menu, using the pull down menu system as before. Note that **Compile and link** has key **F3** assigned to it, in future you may wish to save time by using this key. After selecting **Compile and link**, you should see messages from the various compiler passes (starting with CPP) appearing in the message window.

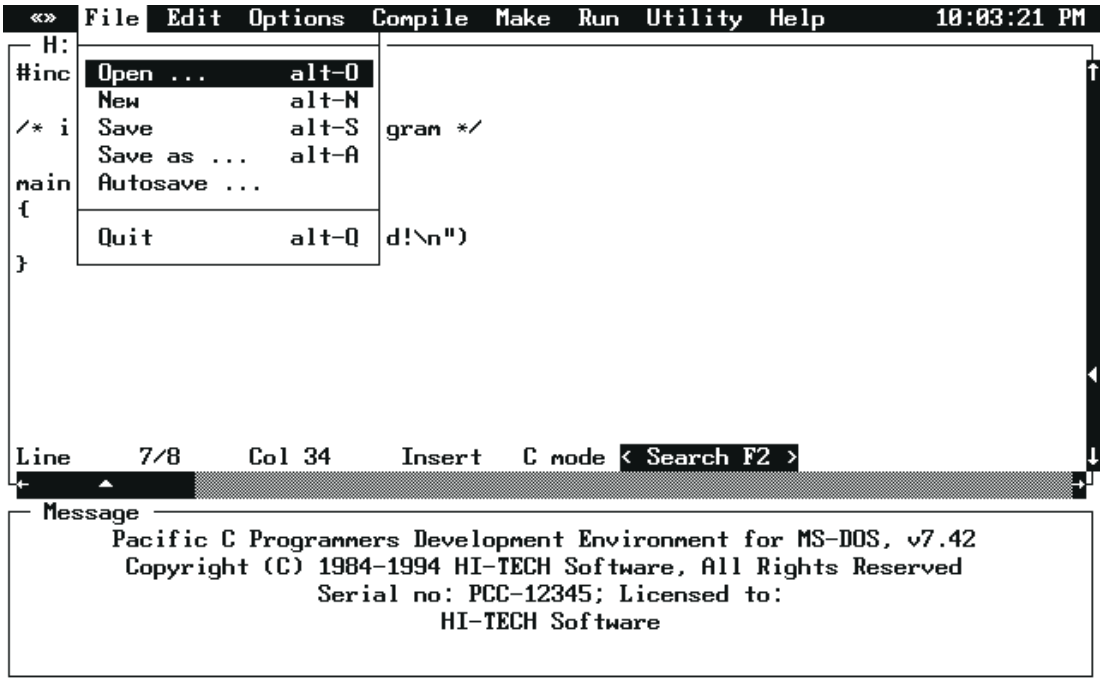


Figure 3-3; PPD File menu

This time, the compiler will not run to completion because we deliberately omitted a semicolon on the end of a line, in order to see how PPD handles compiler errors. After a couple of seconds of disk activity as the CPP and P1 phases of the compiler run, you should hear a “splat” noise and the message window will be replaced by a window containing a number of buttons and the message “; expected”, as shown in figure 3-5.

The text in the frame of the error window details the number of compiler errors generated, and which phase of the compiler generated them. Most errors will come from P1.EXE and CGEN.EXE*, CPP.EXE and LINK.EXE can also return errors. In this case, the error window frame contains the message 1 error, 0 warnings from p1.exe indicating that pass 1 of the compiler found 1 fatal error. It is possible to configure PPD so that non-fatal warnings will not stop compilation. If only warnings are returned, an additional button will appear, labelled CONTINUE. Selecting this button (or F4) will resume the compilation.

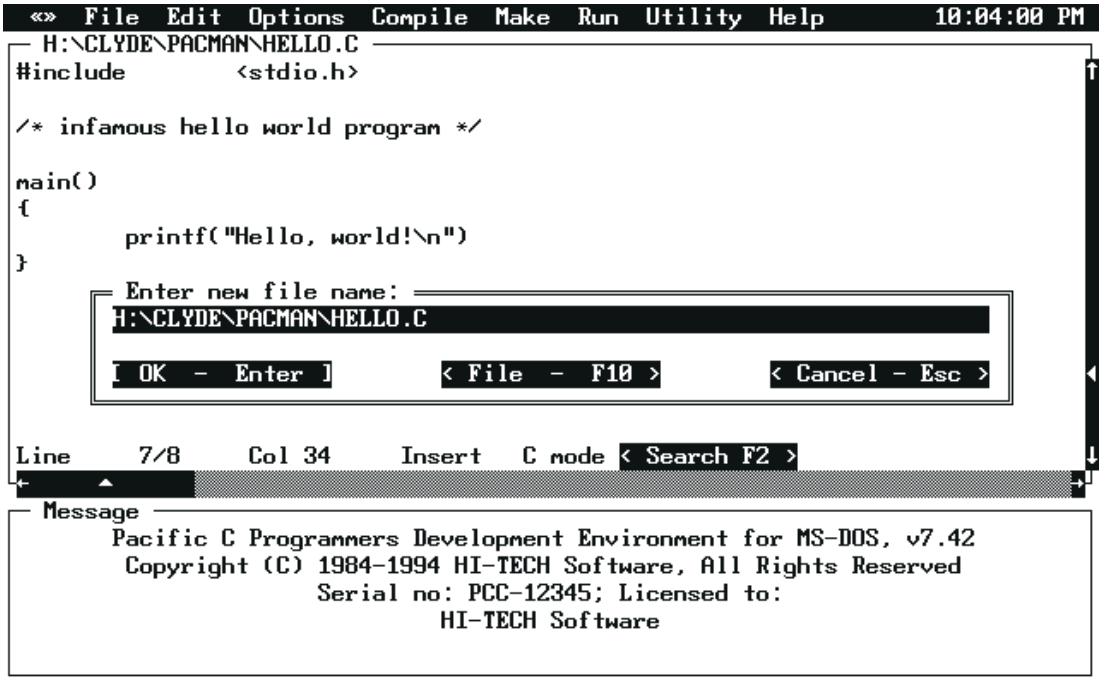


Figure 3-4; PPD File save dialogue

In this case, the error message **`;expected`** will be highlighted and the cursor will have been placed on the start of the line after the printf() statement, which is where the error was first detected. The error window contains a number of buttons, which allow you to select which error you wish to handle, clear the error status display, or obtain an explanation of the currently highlighted error. In order to obtain an explanation of the error message, either select the EXPLAIN button with a mouse click, or press **F1**.

The error explanation for the missing semicolon doesn't give much more information than we already had, however the explanations for some of the more unusual errors produced by the compiler can be very helpful. All errors produced by the pre-processor (CPP), pass 1 (P1), code generator (CGxx), assembler (ASxx) and linker (LINK) are handled. You may dismiss the error explanation by selecting the HIDE button (press **Escape** or use the mouse).

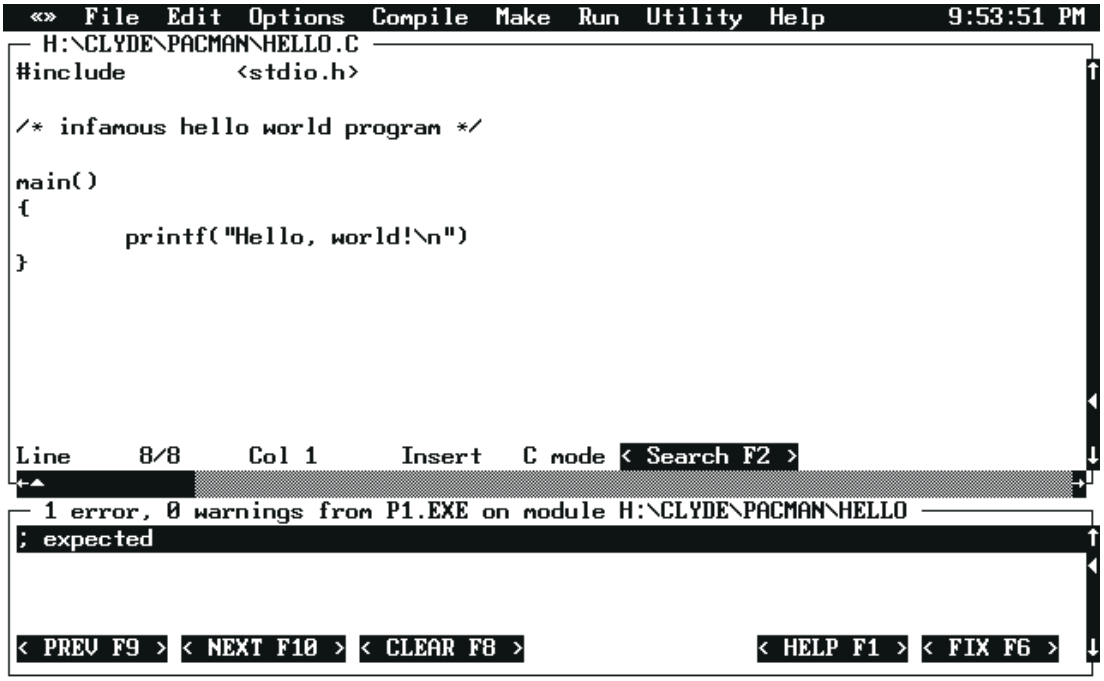


Figure 3-5; PPD Error window

In this instance PPD has analyzed the error, and is prepared to fix the error itself. This is indicated by the presence of the FIX button in the bottom right hand corner of the error window. If PPD is unable to analyze the error, it will not show the FIX button. Clicking on the FIX button, or pressing F6 will fix the error by adding a semicolon to the end of the previous line. A “bip-bip” sound will be generated, and if there was more than one error line in the error window, PPD would move to the next error.

To manually correct the error, move the cursor to the end of the printf() statement and add the missing semicolon. If you have a mouse, simply click the left button on the position to which you want to move the cursor. If you are using the keyboard, move the cursor with the arrow keys. Once the missing semicolon has been added, you are ready to attempt another compilation.

This time, we will “short circuit” the edit-save-compile cycle by pressing **F3** to invoke the “Compile and link” menu item. PPD will automatically save the modified file to a temporary file, then compile it. The message window will then display the commands issued to each compiler phase in turn. If all goes well, you will hear a tone and see the message **`Compilation successful`**.

To run the program, now open the Run menu, and select the item “Run HELLO.EXE”, or simply press F7. PPD will disappear, and the program will run in the DOS screen. The message “Hello, world” will appear, followed by “Press any key to continue”. If you press a key, you will be returned to PPD.

This tutorial has presented a simple overview of single file edit/compile/execute development. PPD is also capable of supporting multi-file projects (including mixed C and assembly language sources) using the project facility. The remainder of this chapter presents a detailed reference for the PPD menu system, editor and project facility.

3.4 The PPD Editor

PPD has a built in text editor designed for the creation and modification of program text. The editor is loosely based on *WordStar* with a few minor differences and some enhancements for mouse based operation. If you are familiar with WordStar or any similar editor you should be able to use the PPD editor without further instruction. PPD also supports the standard PC keys, and thus should be readily usable by anyone familiar with typical MS-DOS or Microsoft Windows editors.

The PPD editor is based in its own window, known as the *edit window*. The edit window is broken up into three areas, the *frame*, the *content region* and the *status line*.

The frame indicates the boundary between the edit window and the other windows on the desktop. The name of the current edit file is displayed in the top left corner of the frame. If a newly created file is being edited, the file name will be set to “untitled”. The frame can be manipulated using the mouse, allowing the window to be moved around the desktop and re-sized.

The content region, which forms the largest portion of the window, contains the text being edited. When the edit window is active, the content region will contain a cursor indicating the current insertion point. The text in the content region can be manipulated using keyboard commands alone, or a combination of keyboard commands and mouse actions. The mouse can be used to position the cursor, scroll the text and select blocks for clipboard operations.

3.4.1 Status Line

The bottom line of the edit window is the status line, containing the following information about the file being edited: **`Line n/m`** shows the current line number, counting from the start of the file, and the total number of lines in the file. **`Col n`** Shows the number of the column containing the cursor, counting from the left edge of the window. If the status line includes the text ^K after the Col entry, it indicates that the editor is waiting for the second character of a WordStar **ctrl-K** command. See the “Editor Keyboard Commands” section for a list of the valid **ctrl-K** commands. If the status line includes the text ^Q after the Col entry, the editor is waiting for the second character of a WordStar **ctrl-Q** command. See the “Editor Keyboard Commands” section for a list of the valid **ctrl-Q** commands. **`Insert`** Indicates whether text typed on the keyboard will be inserted at the cursor position. Using the *insert mode toggle*

command (the **Ins** key on the keypad, or **ctrl-V**), the mode can be toggled between **`Insert`** and **`Overwrite`**. In overwrite mode, text entered on the keyboard will overwrite characters under the cursor, instead of inserting them before the cursor.

`Indent` Indicates that the editor is in auto indent mode. Auto indent mode is toggled using the **ctrl-Q I** key sequence. By default, auto indent mode is enabled. When auto indent mode is enabled, every time a new line is added the cursor will be aligned under the first non-space character in the preceding line. If the file being edited is a C file, the editor will default to **`C mode`**. In this mode, when an opening brace ('{') is typed, the next line will be indented one tab stop. In addition, it will automatically align a closing brace ('}') with the first non-blank character on the line containing the corresponding opening brace. This makes the auto indent mode ideal for entering C code.

The **`SEARCH`** button may be used to initiate a search operation in the editor. To select SEARCH, click the left mouse button anywhere on the text of the button. The search facility may also be activated using the **F2** key and the WordStar **ctrl-Q F** sequence. The **`NEXT`** button is only present if there has already been a search operation. It searches forwards for the next occurrence of the search text. NEXT may also be selected using **shift-F2** or **ctrl-L**. The **`LAST`** button is used to search for the previous occurrence of the search text. This button is only present if there has already been a search operation. The key equivalents for LAST are **ctrl-F2** and **ctrl-P**.

3.4.2 Keyboard Commands

The editor accepts a number of keyboard commands, broken up into the following categories: *Cursor movement commands*, *Insert/delete commands*, *Search commands*, *Block and Clipboard operations* and *File commands*. Each of these categories contains a number of logically related commands. Some of the cursor movement commands and block selection operations can also be performed with the mouse.

Table 3-4 provides an overview of the available keyboard commands and their key mappings. A number of the commands have multiple key mappings, some also have an equivalent menu item.

The Zoom command, **ctrl-Q Z**, is used to toggle the editor between windowed and full-screen mode. In full screen mode, the PPD menu bar may still be accessed either by pressing the ALT key or by using the middle button on a three button mouse.

3.4.3 Block Commands

In addition to the movement and editing command listed in the "Editor Keyboard Commands" table, the PPD editor also supports WordStar style block operations and mouse driven cut/copy/paste clipboard operations. The clipboard is implemented as a secondary editor window, allowing text to be directly entered and edited in the clipboard. The WordStar style block operations may be freely mixed with mouse driven clipboard and cut/copy/paste operations.

The block operations are based on the **ctrl-K** and **ctrl-Q** key sequences which are familiar to anyone who has used a WordStar compatible editor.

Table 3-5 lists the WordStar compatible block operations which are available.

The block operations behave in the usual manner for WordStar type editors with a number of minor differences. “Backwards” blocks, with the block end before the block start, are supported and behave exactly like a normal block selection. If no block is selected, a single line block may be selected by keying block-start (**ctrl-K B**) or block-end (**ctrl-K K**). If a block is already present, any block start or end operation has the effect of changing the block bounds.

Begin Block **ctrl-K B**

The key sequence **ctrl-K B** selects the current line as the start of a block. If a block is already present, the block start marker will be shifted to the current line. If no block is present, a single line block will be selected at the current line.

End Block **ctrl-K K**

The key sequence **ctrl-K K** selects the current line as the end of a block. If a block is already present, the block end marker will be shifted to the current line. If no block is present, a single line block will be selected at the current line.

Go To Block Start **ctrl-Q B**

If a block is present, the key sequence **ctrl-Q B** moves the cursor to the line containing the block start marker.

Go To Block End **ctrl-Q K**

If a block is present, the key sequence **ctrl-Q K** moves the cursor to the line containing the block end marker.

Block Hide Toggle **ctrl-K H**

The block hide/display toggle, **ctrl-K H** is used to hide or display the current block selection. Blocks may only be manipulated with cut, copy, move and delete operations when displayed. The bounds of hidden blocks are maintained through all editing operations so a block may be selected, hidden and re-displayed after other editing operations have been performed. Note that some block and clipboard operations change the block selection, making it impossible to re display a previously hidden block.

Copy Block **ctrl-K C**

The **ctrl-K C** command inserts a copy of the current block selection before the line which contains the cursor. A copy of the block will also be placed in the clipboard. This operation is equivalent to a clipboard **Copy** operation followed by a clipboard **Paste** operation.

Move Block **ctrl-K V**

The **ctrl-K V** command inserts the current block before the line which contains the cursor, then deletes the original copy of the block. That is, the block is moved to a new position just before the current line. A copy of the block will also be placed in the clipboard. This operation is equivalent to a clipboard **Cut** operation followed by a clipboard **Paste** operation.

Delete Block **ctrl-K Y**

The **ctrl-K Y** command deletes the current block. A copy of the block will also be placed in the clipboard. This operation may be undone using the clipboard **Paste** command. This operation is equivalent to the clipboard **Cut** command.

Read block from file **ctrl-K R**

The **ctrl-K R** command prompts the user for the name of a text file which is to be read and inserted before the current line. The inserted text will be selected as the current block. This operation may be undone by deleting the current block.

Write block to file **ctrl-K W**

The **ctrl-K W** command prompts the user for the name of a text file to which the current block selection will be written. This command does not alter the block selection, editor text or clipboard in any way.

Indent

This operation is available only via the **Edit** menu. It will indent by one tab stop the current block, or the current line if no block is selected.

Outdent

This is the complement to the previous operation, i.e. it removes one tab from the beginning of each line in the selection, or the current line if there is no block selected. It is accessible only via the **Edit** menu.

Comment/Uncomment

Also available only in the **Edit** menu, this operation will insert or remove a C++ style comment leader (//) at the beginning of each line in the current block, or the current line if there is no block selected. If a line is currently uncommented, it will be commented, and if it is already commented, it will be uncommented. This is repeated for each line in the selection. This allows a quick way of commenting out a portion of code during debugging or testing.

3.4.4 Clipboard Editing

The PPD editor also supports mouse driven clipboard operations, similar to those supported by several well known graphical user interfaces.

Text may be selected using mouse click and drag operations, deleted, cut or copied to the clipboard, and pasted from the clipboard. The clipboard is based on a standard editor window and may be directly manipulated by the user. Clipboard operations may be freely mixed with WordStar style block operations.

3.4.4.1 Selecting Text

Blocks of text may be selected using left mouse button and click or drag operations. The following mouse operations may be used: **Mouse Click** A single click of the left mouse button will position the cursor and hide the current selection. The **Hide** menu item in the **Edit** menu, or the **ctrl-K H** command, may be used to re display a block selection which was cancelled by a mouse click. **Mouse Double Click** A double click of the left mouse button will position the cursor and select the line as a single line block. Any previous selection will be cancelled. **Mouse Click and Drag** If the left button is pressed and held, a multi line selection from the position of the mouse click may be made by dragging the mouse in the direction which you wish to select. If the mouse moves outside the top or bottom bounds of the editor window, the editor will scroll to allow a selection of more than one page to be made. The cursor will be moved to the position of the mouse when the left button is released. Any previous selection will be cancelled.

3.4.4.2 Clipboard Commands

The PPD editor supports a number of clipboard manipulation commands which may be used to cut text to the clipboard, copy text to the clipboard, paste text from the clipboard, delete the current selection and hide or display the current selection. The clipboard window may be displayed and used as a secondary editing window. A number of the clipboard operations have both menu items and *hot key* sequences. The following clipboard operations are available:

Cut **alt-X**

The **Cut** option copies the current selection to the clipboard and then deletes the selection. This operation may be undone using the **Paste** operation. The previous contents of the clipboard are lost.

Copy **alt-C**

The **Copy** option copies the current selection to the clipboard without altering or deleting the selection. The previous contents of the clipboard are lost.

Paste **alt-V**

The **Paste** option inserts the contents of the clipboard into the editor before the current line. The contents of the clipboard are not altered.

Hide **ctrl-K H**

The **Hide** option toggles the current selection between the hidden and displayed state. This option is equivalent to the WordStar **ctrl-K H** command.

Show clipboard

This menu options hides or displays the clipboard editor window. If the clipboard window is visible, it is hidden. If the clipboard window is hidden it will be displayed and selected as the current window. The clipboard window behaves like a normal editor window in most respects except that no block operations may be used. This option has no key equivalent.

Clear clipboard

This option clears the contents of the clipboard, and cannot be undone. If a large selection is placed in the clipboard, you should use this option to make extra memory available to the editor after you have completed your clipboard operations.

Delete selection

This menu option deletes the current selection without copying it to the clipboard. **Delete selection** should not be confused with **Cut** as it cannot be reversed and no copy of the deleted text is kept. Use this option if you wish to delete a block of text without altering the contents of the clipboard.

3.5 PPD Menus

This chapter presents a item-by-item description of each of the PPD menus. The description of each menu includes a screen dump showing the appearance of the menu within a typical PPD screen.

Table 3-4; Editor keys

Command	Key	WordStar Key
Character left	left arrow	ctrl-S
Character right	right arrow	ctrl-D
Word left	ctrl-left arrow	ctrl-A
Word right	ctrl-right arrow	ctrl-F
Line up	up arrow	ctrl-E
Line down	down arrow	ctrl-X
Page up	PgUp	ctrl-R
Page down	PgDn	ctrl-C
Start of line	Home	ctrl-Q S
End of line	End	ctrl-Q D
Top of window		ctrl-Q E
Bottom of window		ctrl-Q X
Start of file	ctrl-Home	ctrl-Q R
End of file	ctrl-End	ctrl-Q C
Goto previous line		ctrl-Q P
Insert mode toggle	Ins	ctrl-V
Insert CR at cursor		ctrl-N
Open new line below cursor		ctrl-O
Delete char under cursor	Del	ctrl-G
Delete char to left	Backspace	ctrl-H
Delete line		ctrl-Y
Delete to end of line		ctrl-Q Y
Search	F2	ctrl-Q F
Search forwards	shift-F2	ctrl-L
Search backwards	alt-F2	ctrl-P
Toggle auto indent mode		ctrl-Q I
Zoom or unzoom window		ctrl-Q Z
Open file	alt-O	
New file	alt-N	
Save file	alt-S	
Save file (new name)	alt-A	

Table 3-5; Block operation key sequences

Command	Key Sequence
Begin block	ctrl-K B
End block	ctrl-K K
Hide or show block	ctrl-K H
Go to block start	ctrl-Q B
Go to block end	ctrl-Q K
Copy block	ctrl-K C
Move block	ctrl-K V
Delete block	ctrl-K Y
Read block from file	ctrl-K R
Write block to file	ctrl-K W

3.5.1 <<>> Menu

The <<>> (system) menu is present in all *HI-TECH Windows* based applications. It contains any handy system configuration utilities and *desk accessories* which we considered to be worth making a standard part of the desktop.

Setup ...

This menu item selects the standard mouse firmware configuration menu, and is present in any *HI-TECH Windows* based application. The “mouse setup” dialogue allows you to adjust the horizontal and vertical sensitivity of the mouse, the *ballistic threshold*[2] of the mouse and the mouse button auto-repeat rate. This menu item will not be selectable if there is no mouse driver installed. With some early mouse drivers, this dialogue will not function correctly. Unfortunately there is no way to detect drivers which exhibit this behaviour because even the “mouse driver version info” call is missing from some of the older drivers!

This dialogue will also display information about what kind of video card and monitor you have, DOS version, and free DOS memory available. See figure 3-6.

About...

The About dialogue displays information on the version number of PPD, the licence details, and the authors.

2 The *ballistic threshold* of a mouse is the speed beyond which the response of the pointer to further movement becomes exponential. Some primitive mouse drivers do not support this feature.

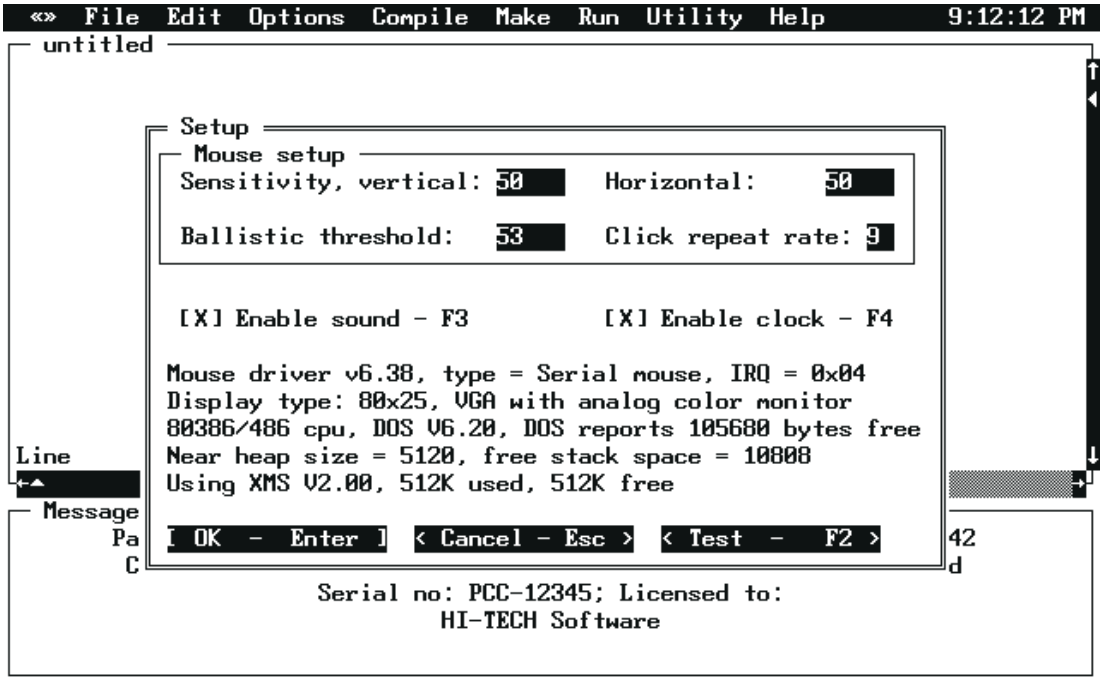


Figure 3-6; Setup dialogue

3.5.2 File Menu

The **Edit** menu contains editor file handling commands and the PPD **Quit** command. See figure 3-3.

Open ... **alt-O**

This command loads a file into the editor. You will be prompted for the file name and if a wildcard (e.g. "*.C") is entered, you will be presented with a file selector dialogue. If the previous edit file has been modified but not saved, you will be given an opportunity to save it or abort the command.

New **alt-N**

The **New** command clears the editor and creates a new edit file with default name "untitled". If the previous edit file has been modified but not saved, you will be given a chance to save it or abort the command.

Save **alt-S**

Save the current edit file. It the file is "untitled", you will be prompted for a new name, otherwise the current file name (displayed in the edit window's frame) will be used

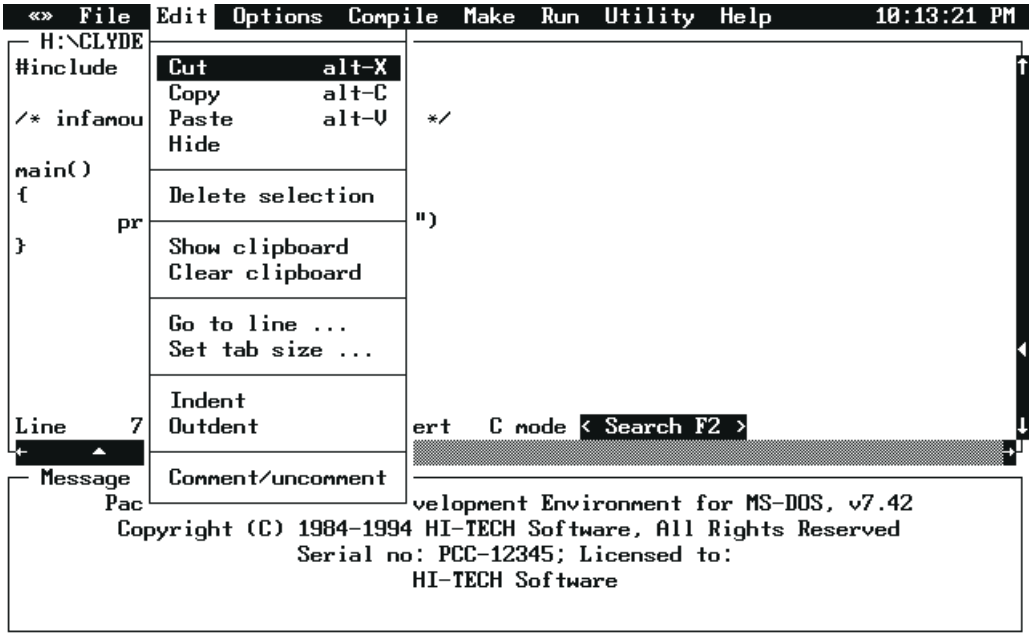


Figure 3-7; PPD edit menu

Save as ... **alt-A**

This command is similar to **Save**, except that a new file name is always requested.

Autosave...

This item will invoke a dialogue box allowing you to enter a time interval in minutes for auto saving of the edit file. If the value is non-zero, then the current edit file will automatically be saved to a temporary file at intervals. Should PPD not exit normally, e.g. if your computer suffers a power failure, then the next time you run PPD, it will automatically restore the saved version of the file.

Quit **alt-Q**

The **Quit** command is used to exit PPD to DOS. If the current edit file has been modified but not saved, you will be given an opportunity to save it or abort the command.

3.5.3 Edit Menu

The Edit menu contains items relating to the text editor and clipboard. The edit menu is shown in figure 3-7.

Cut **alt-X**

The **Cut** option copies the current selection to the clipboard and then deletes the selection. This operation may be undone using the **Paste** operation. The previous contents of the clipboard are lost.

Copy **alt-C**

The **Copy** option copies the current selection to the clipboard without altering or deleting the selection. The previous contents of the clipboard are lost.

Paste **alt-V**

The **Paste** option inserts the contents of the clipboard into the editor before the current line. The contents of the clipboard are not altered.

Hide

The **Hide** option toggles the current selection between the hidden and displayed state. This option is equivalent to the WordStar **ctrl-K H** command.

Delete selection

This menu option deletes the current selection without copying it to the clipboard. **Delete selection** should not be confused with **Cut** as it cannot be reversed and no copy of the deleted text is kept. Use this option if you wish to delete a block of text without altering the contents of the clipboard.

Show clipboard

This menu options hides or displays the clipboard editor window. If the clipboard window is visible, it is hidden. If the clipboard window is hidden it will be displayed and selected as the current window. The clipboard window behaves like a normal editor window in most respects except that no block operations may be used. This option has no key equivalent.

Clear clipboard

This option clears the contents of the clipboard, and cannot be undone. If a large selection is placed in the clipboard, you should use this option to make extra memory available to the editor after you have completed your clipboard operations.

Go to line ... **alt-G**

The **Go to line** command allows you to go directly to any line within the current edit file. You will be presented with a dialogue prompting you for the line number. The title of the dialogue will tell you the allowable range of line numbers in your source file.

Set tab size ... **alt-T**

This command is used to set the size of tab stops within the editor. The default tab size is 8, values from 1 to 16 may be used. For normal C source code 4 is another good value. The tab size will be stored as part of your project if you are using the **Make** facility.

Indent

Selecting this item will indent by one tab stop the currently highlighted block, or the current line if there is no block selected.

Outdent

This is the reverse operation to Indent. It removes one tab from the beginning of each line in the currently selected block, or current line if there is no block.

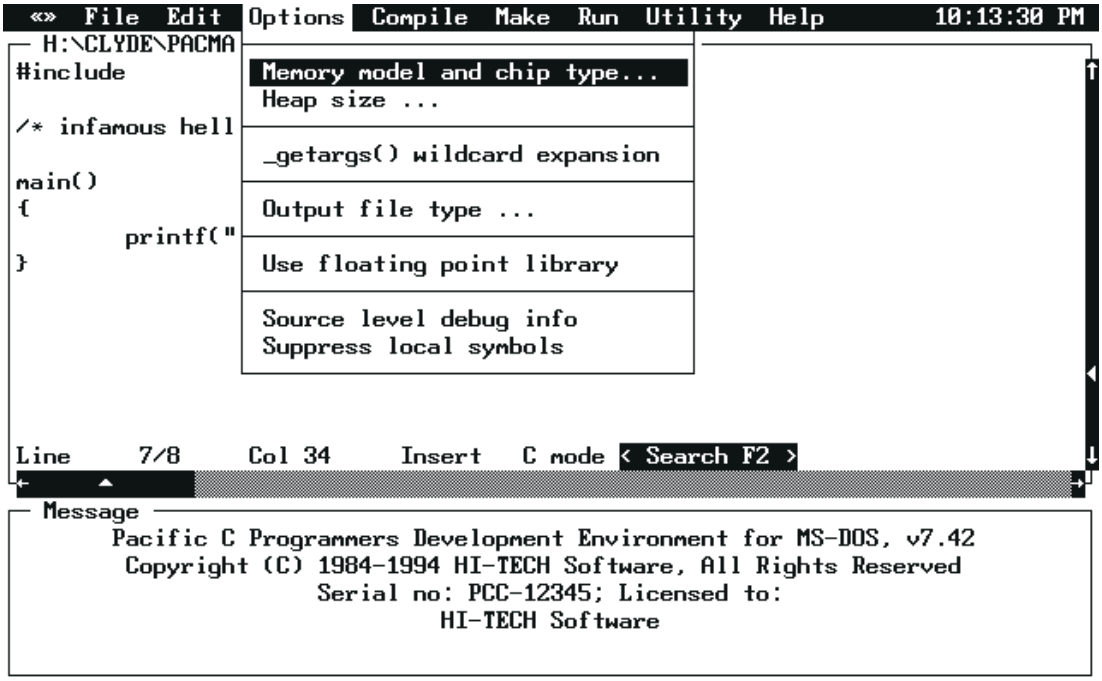


Figure 3-8; PPD options menu

Comment/Uncomment
This item will insert or remove C++ style comment leaders (//) from the beginning of each line in the current block, or the current line. This has the effect of commenting out those lines of code so that they will not be compiled. If a line is already commented in this manner, the comment leader will be removed.

C colour coding
If this option is selected, the editor will display C syntax elements in specified colours. This is especially useful for finding missing closing comment indicators, etc. Turn this option off when editing a non-C file that you do not want to be displayed in colour.

3.5.4 Options Menu

The **Options** menu contains commands which allow selection of compiler options, memory models, and target processor. Selections made in this menu will be stored in a project file, if one is being used. The Options menu is shown in figure 3-8.

Memory model and chip type...
This option activates a dialogue box which allows you to select which code generation memory model you wish to use. Available memory models are small (64K code plus 64K data) and large (large code and large data). You may also select the processor type, either 8086 or 80186/286. There is also a toggle for generation of 8087 (coprocessor) floating point instructions. By default a floating point library is linked that will work with or without a maths coprocessor.

Heap size...
This entry will open a dialogue box allowing you to set the heap size. The “heap size” value is placed in the .EXE file header to tell DOS how much heap and stack space should initially be allocated to the program. With Pacific C, it is used to limit the size of “near” data and heap, there is no “far” heap as such because the Pacific C memory allocation routines use MS-DOS system calls to obtain memory. The heap size should be entered as a hexadecimal value between 0 and 10000 (hex) inclusive. The default value is 0, which tells Pacific C to use a default heap size of 64K bytes. If a value larger than 64K is specified, it will not be accepted.

Output file type
The default output file type is a DOS .EXE file, i.e. an executable program. PPD will also create libraries that you can use to combine several modules you have written. Libraries are discussed in more detail in the Linker manual and the Utilities manua. This option will allow you to specifiy that you want to create a library rather than an .EXE file. A library can only be created from a project file.

Getargs wild card expansion
This option is a toggle, i.e. it is either on (indicated by a diamond shape next to it) or off (nothing next to it). When on, it will cause the linker to link into your program code to perform wild card expansion on command line arguments, i.e. command line arguments with * or ? characters will be expanded into matching file names.

Use floating point library
This option is also a toggle. It is used to tell the linker that you wish to use the *floating point printf()* support library. The *float* library includes a version of *printf()* which supports the floating point output formats %e, %f and %g. Use of this option will increase the size of the compiled application. You should only use this option if you wish to use floating point formats in *printf()*, it is not necessary if you only wish to perform floating point calculations.

Source level debug info
This menu item is used to enable or disable source level debug information in the current symbol file. If you are using a HI-TECH Software debugger like LUCIFER, you should enable this option.

Suppress local symbols
Prevents the inclusion of all local symbols in the symbol file. Even if this option is not active, the linker will filter irrelevant compiler generated symbols from the symbol file.

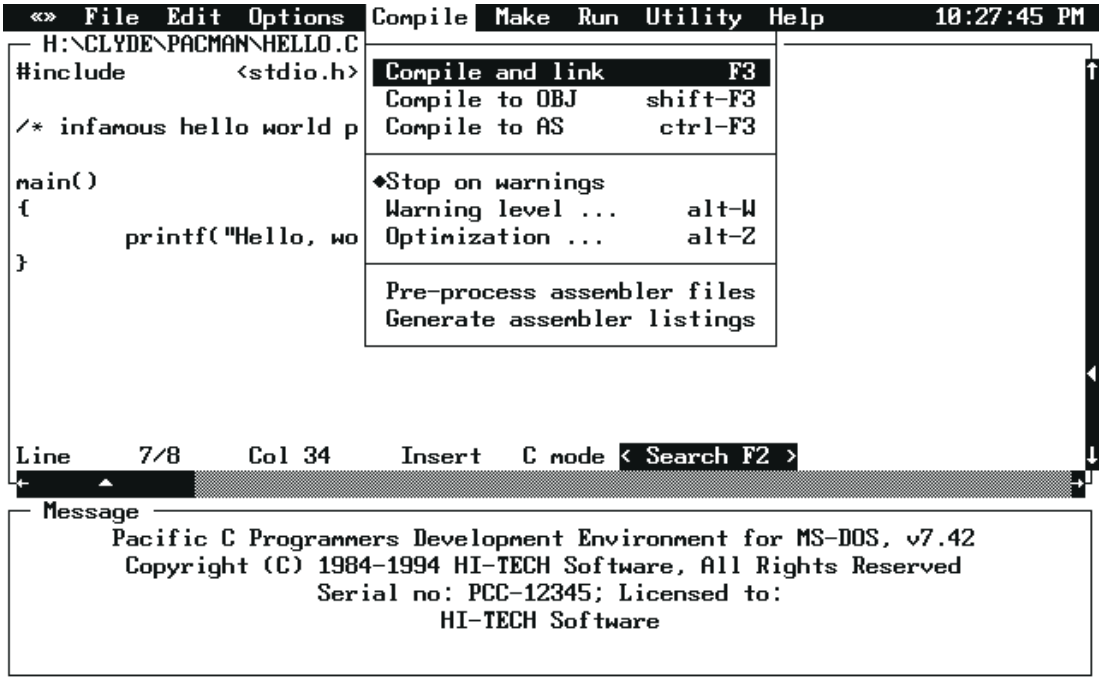


Figure 3-9; PPD Compile Menu

3.5.5 Compile Menu

The **Compile** menu, shown in figure 3-9, contains the various forms of the compile command along with several machine independent compiler configuration options.

Compile and link **F3**

This command will compile a single source file and then invoke the linker and *objtoexe* to produce an executable file. If the source file is an .AS file, it will be passed directly to the assembler. The output file will have the same base name as the source file, but a different extension. For example HELLO.C would be compiled to HELLO.EXE.

Compile to .OBJ **shift-F3**

Compiles a single source file to a .OBJ file only. The linker and objtoexe are not invoked. .AS files will be passed directly to the assembler. The object file produced will have the same base name as the source file and extension .OBJ.

Compile to .AS **ctrl-F3**
This menu item compiles a single source file to assembly language, producing an assembler file with the same base name as the source file and extension .AS This option is handy if you want to examine or modify the code generated by the compiler. If the current source file is an .AS file, nothing will happen.

Stop on Warnings
This toggle determines whether compilation will be halted when non-fatal errors are detected. A mark appears against this item when it is active.

Warning level ... **alt-W**
This command calls up a dialogue which allows you set the compiler warning level, i.e. it determines how picky the compiler is about legal but dubious code. The range of currently implemented warning levels is -3 to 9, lower warning levels are stricter. At level 9 all warnings (but not errors) are suppressed. Level 1 suppresses the "func() declared implicit int" message which is common when compiling Unix derived code. Level 3 is suggested for compiling code written with less strict (and K&R) compilers. Level 0 is the default. This command is equivalent to the -W option to the PACC command.

Optimization ... **alt-Z**
Selecting this item will open a dialogue allowing you to select different kinds and levels of optimization. The default is no optimization. Selections made in this dialogue will be saved in the project file if one is being used.

Pre-process assembler files
Selecting this item (which will be indicated by a diamond mark against it if the menu is re-opened) will make PPD pass assembler files through the pre-processor before assembling. This makes it possible to use C pre-processor macros and conditionals in assembler files.

Generate assembler listing
This menu option tells the assembler to generate a listing file for each C or assembler source file which is compiled. The name of the list file is determined from the name of the symbol file, for example TEST.C will produce a listing file called TEST.LST.

3.5.6 Make Menu

The **Make** menu (figure 3-10) contains all of the commands required to use the PPD *project* facility. The project facility allows creation of complex multiple source file applications with ease, as well as a high degree of control of some internal compiler functions and utilities.

To use the project facility, it is necessary to follow several steps.

- Create a new project file using the **Start new project ...** command. After selecting the project file name, PPD will present several dialogues to allow you to set up the memory model.
- Enter the list of source file names using the **Source file list ...** command.
- Set up any special libraries, pre-defined pre-processor symbols, object files or linker options using the other items in the **Make** menu.

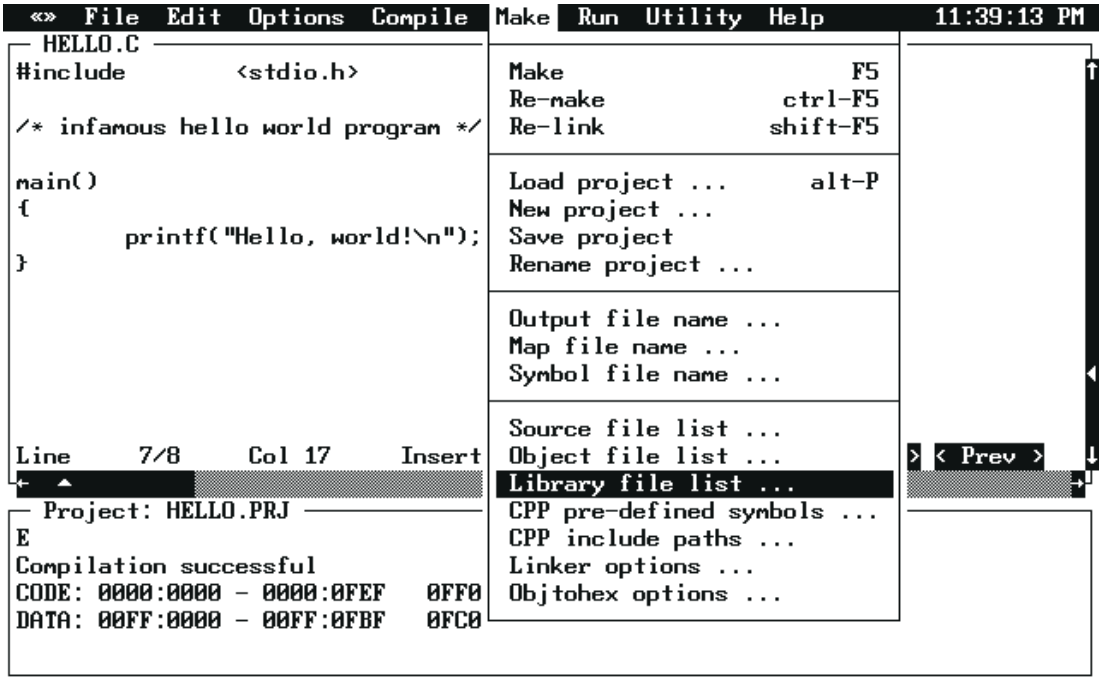


Figure 3-10; PPD make menu

- Save the project file using the **Save project** command.
- Compile your project using the **Make** or **Re-Make** command.

Make **F5**

The Make command re-compiles the current project. When Make is selected, PPD re-compiles any source files which have been modified since the last Make command was issued. PPD determines whether a source file should be recompiled by testing the modification time and date on the source file and corresponding object file. If the modification time and date on the source file is more recent than that of the object file, it will be re-compiled.

If all .OBJ files are current but the output file cannot be found, PPD will re-link using the object files already present. If all object files are current and the output file is present and up to date, PPD will print a message in the message window indicating that nothing was done.

PPD will also automatically check dependencies, i.e. it will scan source files to determine what files are #included, and will include those files in the test to determine if a file needs to be recompiled, i.e. if you modify a header file, any source files #including that header file will need to be recompiled.

Should PPD produce the error “Nothing to make” this indicates that you have not entered any files in the **Source file list...**

Re-make **ctrl-F5**

The Re-make command forces recompilation of all source files in the current project. This command is equivalent to deleting all .OBJ files and then selecting **Make**.

Re-link **shift-F5**

The Re-link command relinks the current project. Any .OBJ files which are missing or not up to date will be regenerated.

Load project file ... **alt-P**

This command loads a pre-defined project file. The user is presented with a file selection dialogue allowing a .PRJ file to be selected and loaded. If this command is selected when the current project has been modified but not saved, the user will be given a chance to save the project or abort the command. After loading a project file, the message window title will be changed to display the project file name.

Start new project ...

This command allows the user to start a new project. All current project information is cleared and all items in the Make menu are enabled. The user will be given a chance to save any current project and will then be prompted for the new project’s name.

Following entry of the new name PPD will present several dialogues to allow you to configure the project. These dialogues will allow you to select processor type and memory model, output file type, optimization settings and memory addresses. You will still need to enter source file names in the **Source file list**.

Save project

This item saves the current project to a file.

Rename project...

This will allow you to specify a new name for the project. The next time the project is saved it will be saved to the new file name. The existing project file will not be affected (if it has already been saved).

Output file name ...

This command allows the user to select the name of the compiler output file. This name is automatically setup when a project is created. For example if a project called PROG1 is created and a .EXE file is being generated, the output file name will be automatically set to PROG1.EXE.

Map file name ...

This command allows the user to enable generation of a symbol map for the current project, and specify the name of the map. If a mark character appears against this item, map file generation has been selected. The default name of the map file is generated from the project name, e.g. PROG1.MAP

Symbol file name ...
This command allows selection of generation of a symbol file, and specification of the symbol file name. The default name of the symbol file will be generated from the project name, e.g. PROG1.SYM. The symbol file produced is suitable for use with any HI-TECH Software debugger. If **Symbolic debug info** in the **Options** menu is also selected, the symbol file will also contain line-number information for use with source level debuggers like the HI-TECH Software LUCIFER debugger.

Source file list ...
This option displays a dialogue which allows a *list* of source files to be edited. The source files for the project should be entered into the list, one per line. When finished, the source file list can be exited by pressing escape, clicking the mouse on the "DONE" button, or clicking the mouse in the menu bar.

The source file list can contain any mix of C and assembly language source files. C source files should have suffix .C and assembly language files suffix .AS so that PPD can determine to which parts of the compiler files should be passed.

Object file list ...
This option allows any extra .OBJ files to be added to the project. One .OBJ file per line should be entered, operation of this dialogue is the same as for the source file list dialogue. This list will normally only contain one object file: the run-time startoff module for the current code generation model. For example, if a project is generating small model code, by default this list will contain the small model runtime startoff module RT86—DS.OBJ. Object files corresponding to files in the source file list SHOULD NOT be entered here as .OBJ files generated from source files are automatically used. This list should only be used for extra .OBJ files for which no source code is available, such as run-time startoff code or utility functions brought in from an outside source.

If a large number of .OBJ files need to be linked in, they should be condensed into a single .LIB file using the LIBR utility and then accessed using the **Library file list ...** command.

Library file list ...
This command allows any extra object code libraries to be searched when the project is linked. This list normally only contains the default libraries for the memory model being used, for example if the current project is generating small model code and floating point code is in use, this list will contain the libraries 86DSC.LIB and 86DSF.LIB. If an extra library, brought in from an external source is required, it should be entered here.

It is a good practice to enter any non-standard libraries before the standard C libraries, in case they reference extra standard library routines. The normal order of libraries should be: user libraries, floating point library, standard C library. The floating point library should be linked before the standard C library if floating point is being used.

CPP pre-defined symbols ...
This command allows any special pre-defined symbols to be defined. Each line in this list is equivalent to a -D option to the command line compiler PACC. For example, if a CPP macro called DEBUG with value 1, needs to be defined, add the line DEBUG=1 to this list. Some standard symbols will be pre-defined in this list, these should not be deleted as some of the standard header file rely on their presence.

CPP include paths ...
This option allows extra directories to be searched by the C pre-processor when looking for header files. When a header file enclosed in angle brackets, for example <stdio.h> is included, the compiler will search each directory in this list until it finds the file.

Linker options ...
This command allows the options passed to the linker by PPD to be modified. The default contents of the linker command line are generated by the compiler from information selected in the Options menu: memory model, etc. You should only use this command if you are sure you know what you are doing !

Objtohex options ...
This command allows the options passed to objtohex by PPD to be modified. Normally you will not need to change these options as generation of .EXE files can be selected in the Options menu. If you want to generate one of the "strange" output formats which objtohex can produce, like COFF files, you will need to change the objtohex options using this command.

3.5.7 Run Menu

The **Run** menu shown in figure 3-11 contains options allowing MS-DOS commands and user programs to be executed.

Table 3-6; Macros usable in user commands

Macro name	Meaning
$(LIB)	Expands to the name of the system library file directory, e.g. C:\PACIFIC\LIB\
$(CWD)	The current working directory.
$(INC)	The name of the system include directory.
$(EDIT)	The name of the file currently loaded into the editor. If the current file has been modified, this will be replaced by the name of the auto saved temporary file. On return this will be reloaded if it has changed.
$(OUTFILE)	The name of the current output file, i.e. the executable file
$(PROJ)	The base name of the current project, e.g. if the current project file is AUDIO.PRJ, this macro will expand to AUDIO with no dot or file type.

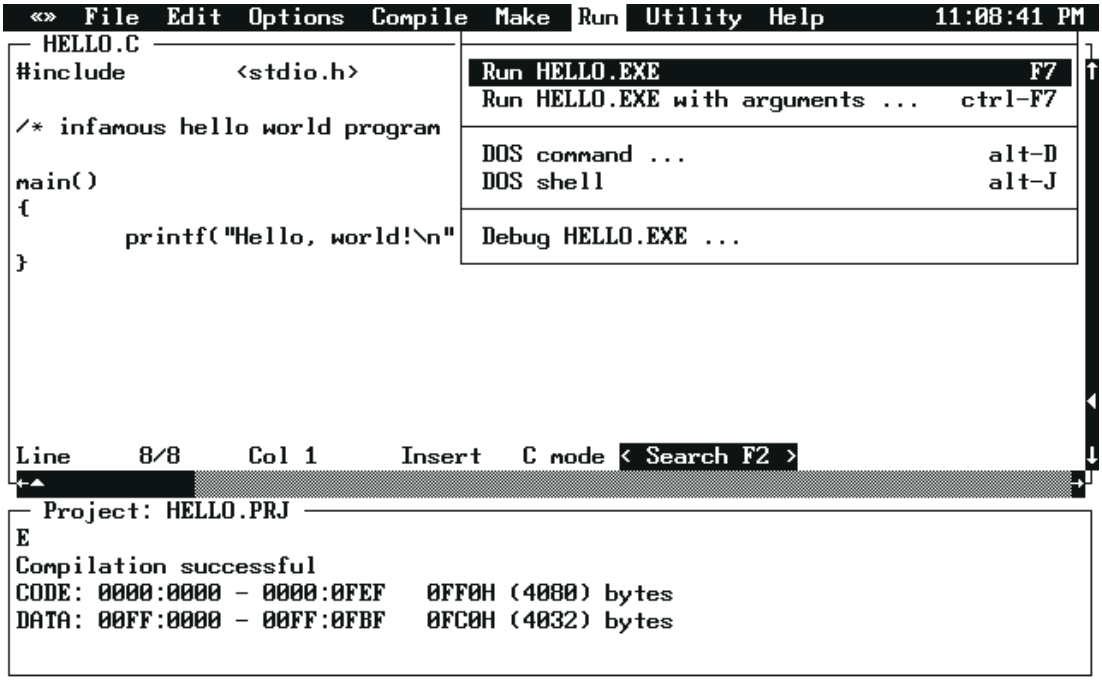


Figure 3-11; PPD run menu

Run PROG.EXE. **F7**

This option runs a compiled program. It is disabled until a program has been compiled, and is updated to reflect the program name. When selected, PPD will swap itself out of memory and run the program, with no command line arguments.

Run PROG.EXE with arguments... **ctrl-F7**

If the program requires arguments, use this option instead. It will prompt with a dialogue box to allow command line arguments to be supplied for the program.

DOS command ... **alt-D**

This option allows a DOS command to be executed exactly like it had been entered at the COMMAND.COM prompt. This command could be an internal DOS command like DIR, or the name of a program to be executed. If you want to escape to the DOS command processor, use the DOS Shell command as below. Warning: do not use this option to load TSR programs. This command will be retained in the initialization file.

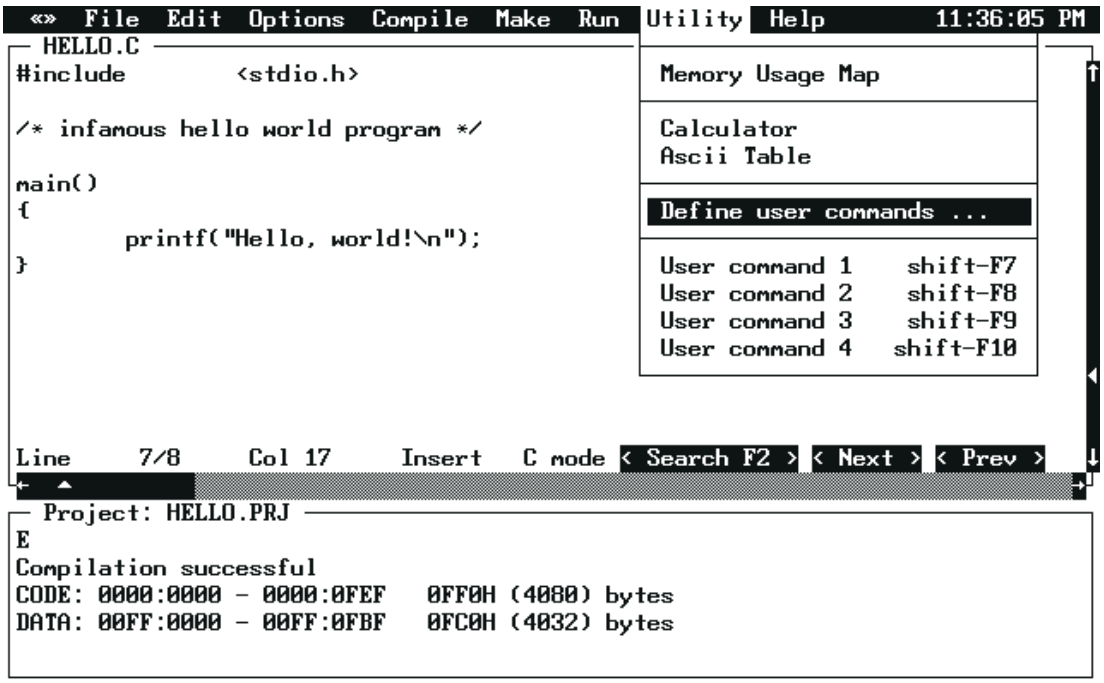


Figure 3-12; PPD utility menu

DOS Shell **alt-J**

This item will invoke a DOS COMMAND.COM shell, i.e. you will be immediately presented with a DOS prompt, unlike the DOS command item which prompts for a command. To return to PPD, type “exit” at the DOS prompt.

Debug PROG.EXE

This option runs the LUCIFER debugger with the current compiled program and symbol file. See the Lucifer manual for more information on debugging.

3.5.8 Utility Menu

The **Utility** menu contains any useful utilities which have been included in PPD.

String Search...

The String Search command will allow you to search multiple files for occurences of a regular expression. When selected, a dialog box opens that allows you to enter a search string, and a file list. The search string can be simply any sequence of characters, including spaces, that will be searched for anywhere in the named files. The file list is a space-separated list of file names. You can use wild cards in this list,

e.g. `*.c` would match all files with file type of `.c` in the current directory. There is also a check box to select case-sensitivity in the string search - unless this is checked upper and lower case are not distinguished in the search.

You can also choose to search the set of files listed in your **Source file list** in the current project. A radio button allows you to choose this option, in which case the file list is hidden.

After entering the string and selecting the files, press Enter or click the OK button, and PPD will search the files. The results will be displayed in a small windows at the bottom right of the screen. This consists of one line per match, with a file name and line number. To go to a particular match, double click on the entry. Note that the window is small, and there are likely to be additional entries not initially visible - use the scroll bar at the right to scroll up and down.

To perform another search, you can simply click the **Search** button in the window - selecting the **Utility/String search** menu command again will hide the search results window. You can also hide it by clicking the close widget in the top left of its frame. The usual methods of resizing and moving the window work.

Memory Usage Map

This menu option displays a window which contains a detailed memory usage map of the last program which was compiled. The actual contents of this window will vary between different target processors. See the processor specific section of the compiler manual for more details about the memory usage map window.

The memory usage map window may be closed by clicking the mouse on the close box in the top left corner of the frame, or by pressing ESC while the memory map is the front most window.

Calculator

This command selects the HI-TECH Software programmer's calculator. This is a multi-display integer calculator capable of performing calculations in bases 2 (binary), 8 (octal), 10 (decimal) and 16 (hexadecimal). The results of each calculation are displayed in all four bases simultaneously. Operation is just like a "real" calculator - just press the buttons! If you have a mouse you can click on the buttons on screen, or just use the keyboard. The large buttons to the right of the display allow you to select which radix is used for numeric entry.

The calculator window can be moved at will, and thus can be left on screen while the editor is in use. The calculator window may be closed by clicking the OFF button in the bottom right corner, by clicking the close box in the top left corner of the frame, or by pressing ESC while the calculator is the front most window.

Ascii Table

This option selects a window which contains an ASCII look up table. The ASCII table window contains four buttons which allow you to close the window and select display of the table in octal, decimal or hexadecimal.

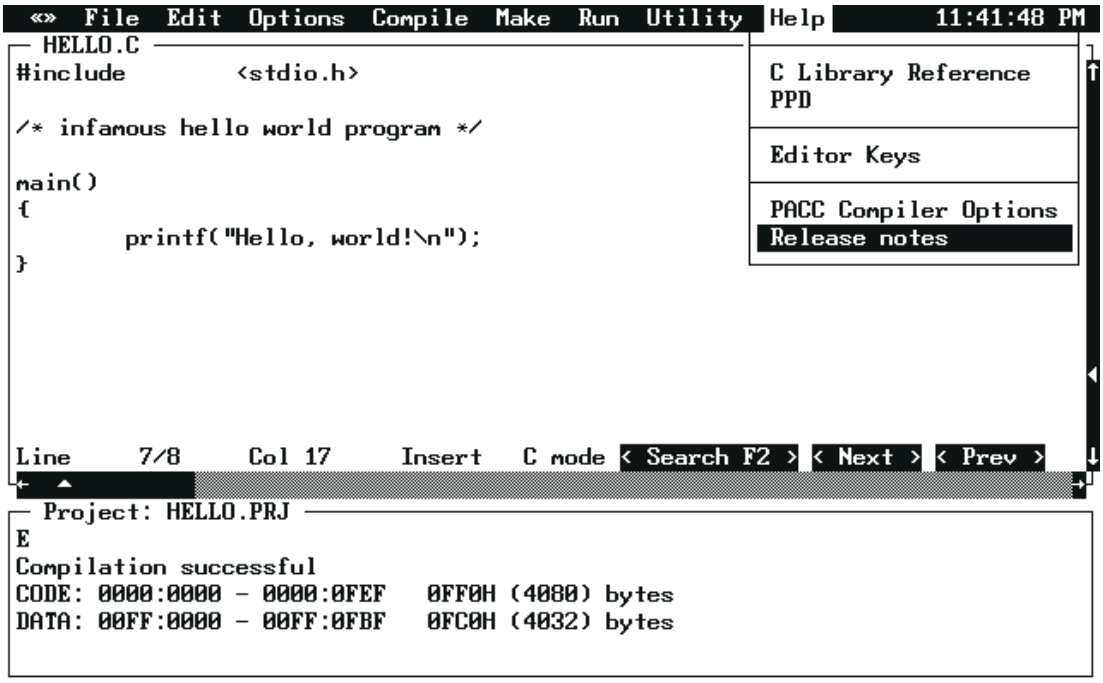


Figure 3-13; PPD help menu

The ASCII table window may be closed by clicking the CLOSE button in the bottom left corner, by clicking the close box in the top left corner of the frame, or by pressing ESC while the ASCII table is the front most window.

Define user commands...

In the Utility menu are four user-definable commands. This item will invoke a dialogue box which will allow you to define those commands. By default the commands are dimmed (not selectable) but will be enabled when a command is defined. Each command is in the form of a DOS command, with macro substitutions available. The macros available are listed in table 3-6.

Each user-defined command has a hot key associated. They are shift F7 through shift F10, for commands 1 to 4. When a user command is executed, the current edit file, if changed, will be saved to a temporary file, and the $(EDIT) macro will reflect the saved temp file name, rather than the original name. On return, if the temp file has changed it will be reloaded into the editor. This allows an external editor to be readily integrated into PPD.

3.5.9 Help Menu

The **Help** menu contains items allowing help about any topics listed to be obtained. For most cross compilers this menu will contain an on-line C library reference, and editor key table and an instruction set reference for the target processor.

On startup, PPD searches the current directory and the help directory for TBL files ,which are added to the **Help** menu. The path of the help directory can be specified by the environment variable HTC_Z80_HLP. If this is not set, it will be derived from the full path name used when PPD was invoked.. If the help directory cannot be located, none of the standard help entries will be available.

C Library Reference

This command selects an on-line manual for the standard ANSI C library. You will be presented with a window containing the index for the manual. Topics can be selected by double clicking the mouse on them, or by moving the cursor with the arrow keys and pressing return.

Once a topic has been selected, the contents of the window will change to an entry for that topic, similar to the one shown below. You can move around within the reference using the keypad cursor keys and the index can be re-entered using the INDEX button at the bottom of the window. If you have a mouse, you can follow *hypertext* links by double clicking the mouse on any word, for example if you are in the **printf** entry and double click on the reference to **fprintf**, you will be taken to the entry for **fprintf**.

This window can be re-sized and moved at will, and thus can be left on screen while the editor is in use.

Editor Keys

This menu item displays a window which contains a complete listing of the editor keyboard commands. The editor keys window may be scrolled using the scroll bar in the right edge of the window, or by using the **down arrow**, **PgDn**, **up arrow** and **PgUp** keys. The window may be closed by clicking on the close box in the top left corner of the frame, or by pressing ESC when the window is front most.

Runtime Organization

The runtime environment of Pacific C is quite straightforward; this is to a large degree due to the C language itself. C does not rely on a large runtime support module, rather there are a few library routines which are basic to the language, while all other runtime support is via specific library packages, e.g. the STDIO library.

4.1 Memory Models

The memory models supported by the compiler are shown table 4-1.In the table, *small* means not more than 64K bytes. Note also that in small model, the stack size is included in the data size. In large model the stack size may not exceed 64K. Code compiled for one model is not compatible with code compiled for another model.

4.2 Runtime Startup

The code located at the start address performs some initialization, notably clearing of the bss (uninitialized data) segment. It also calls a routine to set up the argv, argc values. Depending on compile-time options, this routine may or may not expand wild cards in file names (? and *) and perform I/O redirection.

The code for the startup tasks is in one of the startup modules listed in the table.

Having set up the argument list, the startup code calls the function **_main.** Note the underscore ('_') prepended to the function name. All symbols derived from external names in a C program have this character prepended by the compiler. This helps prevent conflict with assembler names. The function **main()** is, by definition of the C language, the "main program".

4.3 Stack Frame Organization

On entry to **_main,** and all other C functions, some code is executed to set up the local stack frame. The exact code depends on whether the function has any arguments, or uses any register variables and whether it is prototyped. This can involve saving **bp**, copying **sp** to **bp** then adjusting **sp** to allow space for local variables, then finally saving **si** and **di** on the stack if required. Typical code on entry to a function would be:

Table 4-1; Memory models

Model	Run-time startoff module	Standard library	Float library	Size limits
Small	RT86-DS.OBJ	86—DSC.LIB	86—DSF.LIB	small code, small data
Large	RT86-DL.OBJ	86—DLC.LIB	86—DLF.LIB	large code, large data

```
push    bp
mov     bp,sp
sub     sp,#10
push    si
push    di
```

This will allocate 10 bytes of stack space for local variables. The stack frame after this code will look something like figure 4-1.

All references to the local data or parameters are made via bp. The first argument is located at 8[bp] for the small memory model, and at 10[bp] for the large memory model. Each parameter occupies at least 2 bytes. If a char is passed as an argument, it is expanded to int length.

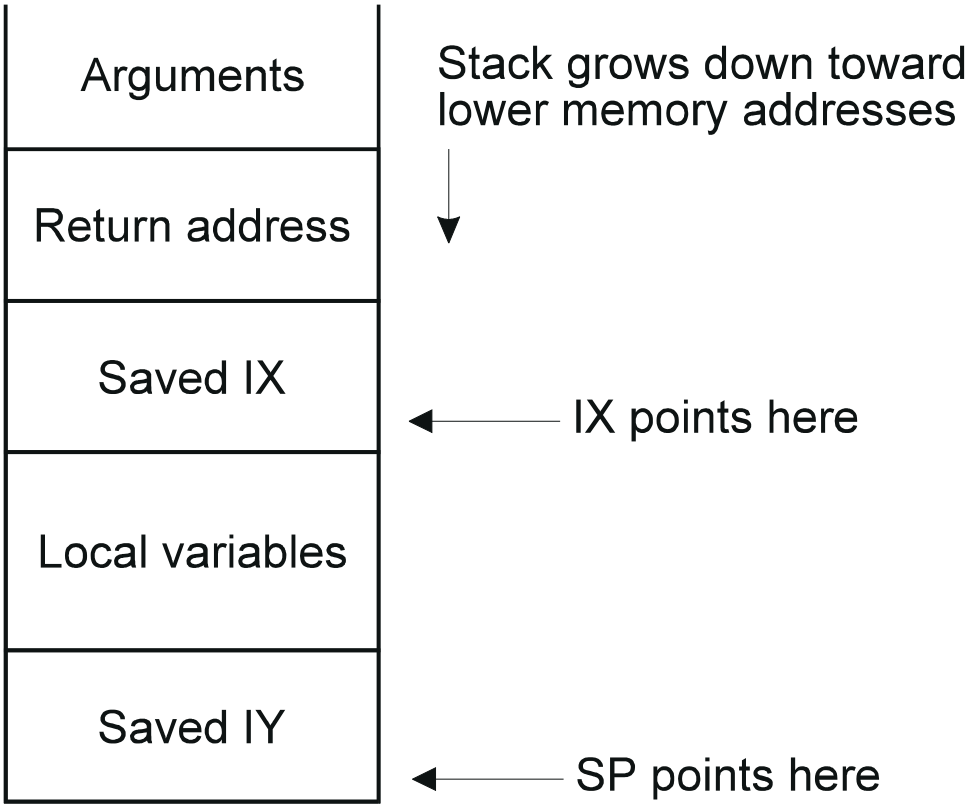


Figure 4-1; Stack frame

Local variables are accessed at locations -1[bp] downwards.

Exit from the function is accomplished by restoring register variables, reloading **sp** from **bp,** then popping the saved **bp**. A return (small code) or far return (large code) is then executed. It is the responsibility of the calling function to remove arguments from the stack.

4.4 Prototyped Function Arguments

The above description holds generally true for all functions, however it is modified for functions with prototypes. Where the compiler is given argument information via a prototype, it optimizes parameter passing as follows:

If a function has a fixed number of arguments, the first and second arguments will, if they are int length or less, be passed in **DX** and **AX** respectively. Arguments past the first two or bigger than int size will be passed on the stack as usual. If a function has a variable argument list (using ellipsis in the prototype) the argument immediately before the ellipsis will always be passed on the stack, to allow the variable part of the argument list to be anchored.

In addition, a prototyped function with a fixed argument list will deallocate the parameters off the stack itself, rather than the calling function doing it. It does this with the “return and deallocate” forms of the RET instruction, e.g. RET #4.

4.5 Stack and Heap Allocation

As previously mentioned, the stack grows downwards. On startup, the stack pointer is initialized to the top of available memory. The heap, i.e. the area of memory dynamically allocated via *sbrk(), calloc()* or *malloc(),* grows upwards from the top of statically allocated memory. This is defined by the top of the bss segment, which the linker gives the name **__Hbss**. *Sbrk()* checks the amount of memory left between the top of the heap and the bottom of the stack, and if less than 1k bytes (4k for the large model) would be left after granting the *sbrk()* request, it will deny it. This value may be modified by editing sbrk.as, re-assembling it and replacing it in the library.

4.6 Linkage to Assembler

It is quite feasible to write assembler routines to interface with C code. The following points should be noted:

- When a routine is called from C, the arguments are pushed onto the stack in reverse order, so that the lexically first argument to the call will have the lowest memory address. The same convention must be followed by an assembler routine calling a C function.

- When a function is called from C code, the registers **bp, si** and **di** must be preserved. If they are to be used, they should be saved on the stack and restored before returning. All other registers may be destroyed. Similarly, when calling a C function, an assembler routine should expect only those registers to be unmodified.
- All C external names have an underscore prepended. Any assembler symbol to be referenced from C must start with an underscore, which should be omitted when using the name in C code.
- Return values are in **ax** for words, and **ax** and **dx** for long words, with the high word in **dx**. Byte returns are in **al** but sign or zero extended as appropriate into **ah**. Floating point functions take arguments on the stack, but return values in **DX** and **AX** (32 bit floats), **BX, CX, DX** and **AX** (64 bit floats). The registers are listed in descending order of significance, i.e. **AX** will have the least significant bits.

4.6.1 Assembler Interface Example

The following piece of code illustrates a function to be called from C (small memory model only) which allows access to memory outside the programs allocated memory. The function is called peek(), and takes two arguments, representing the segment and offset parts of the address to be examined.

```
;        peek(seg, offs)
;        returns the byte at the
;        specified physical address

         .psect   _TEXT
         .globl   _peek
_peek:
         push     bp
         mov      bp,sp
         mov      es,4[bp] ;get segment
         mov      bx,6[bp] ;offset part
         mov      al,es:[bx]  ;get the byte
         mov      ah,#0       ;zero hi byte
         pop      bp
         ret                  ;finito
```

This function would be edited into a file called PEEK.AS. This may then be linked into a C program, e.g. by the command line

```
PACC -V MAIN.C PEEK.AS
```

If this function was prototyped, the argument passing would differ. For example:

```
;       peek(unsigned seg, unsigned offs)

        .psect  _TEXT
        .globl  _peek
_peek:
        mov     es,dx       ;segment part
        mov     bx,ax       ;offset part
        mov     al,es:[bx]
        mov     ah,#0
        ret
```


Note how much simpler the passing of arguments in **DX** and **AX** makes the code. But remember that this is only done if the function is prototyped.

4.6.2 Signatures

Because the differing calling conventions make it possible for a function to be called with incorrect calling sequences, the compiler and linker implement a feature called *signatures*. This is a means of allowing the linker to check that the declarations of a function are all consistent. The compiler automatically generates a signature for a function whenever it is called or defined. If you write an assembler language function you may opt to provide a signature to check that you are using it correctly when calling it. To get the correct signature, place a definition of the function in a sample C file, compile to assembler and copy the signature. E.g.

```
unsigned int
peek(unsigned seg, unsigned offs)
{
        return 0;
}
```

When compiled to assembler (using the -S option to C.EXE or the “Compile to .AS” menu selection of HPD) you will get a line in the .AS file of the form:

```
        .signat _peek,8250
```

Simply copy this line into your assembler file. This will allow the linker to check this against the compiler generated signatures from references to this function. Note that this does not guarantee you have written code that correctly accesses the arguments, etc. This is still up to you to ensure.

4.7 Data Representation

The sizes of various data types are shown in the table below. For the 8086 byte ordering is always low byte first.

4.7.1 Longs

Long values are represented in 32 bits, with the low order word first. Within each word, the 8086 imposes a byte ordering of low byte first. This means that a long value is laid out as shown in figure 4-3.

4.7.2 Pointers

Pointers in the tiny and small model are 16 bits, and represent only the offset portion of the address. The DS register supplies the segment portion of the address for all data references (SS is set to the same as DS) while the CS register supplies the segment part of all text (i.e. program code) references.

When the large memory model is used, all pointers occupy 32 bits. Note that all statically allocated data will be accessed within the segment pointed to by DS. ES is used direct accesses to local variables and parameters. SS and DS are initialized at program startup and are not modified.

4.7.3 Floats

Float and double are stored in 32 and 64 bit quantities respectively. The representation is as used by the 8087 numeric co-processor, i.e. IEEE format.

4.8 Linking 8086 Programs

The subject of object code linking in general is complex; where the 8086 is concerned it is complicated by its segmented architecture. It is necessary to have a reasonable understanding of the issues involved if you wish to use the linker directly, i.e. rather than using the C command to perform all linking. The HI-TECH linker is very flexible, being suitable for ROM code as well as .EXE files, however this flexibility naturally involves the user making appropriate decisions in the linking process. You may need

Table 4-2; Data types

Data type	Size (bits)
int	16
char	8
short	16
long	32
float	32
double	64

address+0	address+1	address+2	address+3
least significant	byte 1	byte 2	most significant

Figure 4-3; Long layout in memory

to read the following discussion several times and try examples before becoming comfortable with using the linker directly. In most cases you will not need to use the linker directly, as the C.EXE and HPD.EXE drivers will automatically run the linker with correct options.

4.8.1 Terms

Two terms are used to describe sections of memory, and it is important to understand the distinction. A *psect* is a section of memory treated by the linker as a contiguous block. When linking 8086 programs a psect will not normally exceed 64k in size. A *segment* is a block of memory up to 64k in size addressed (or pointed to) by a segment register. Locations within the segment are addressed by offsets from the segment register. The contents of the segment register is referred to as the *segment address* of the segment, and is equal to the physical memory address of the segment divided by 16. Since the segment address must be an integer, this implies that all segments must begin on a 16 byte boundary in physical memory.

In normal use one or more psects will be addressed within one segment.

4.8.2 Link and Load Addresses

The HI-TECH linker deals with two types of addresses called *link* and *load* addresses. The manner in which these addresses are specified for a psect is detailed in the linker section, but it is necessary to discuss how the two types of addresses fit into the 8086 run-time model.

4.8.3 Segmentation

The 8086 has a segmented architecture, where all addresses are represented as 16 bit offsets from segment registers. The segment registers are each 16 bits but are left shifted 4 bits before use and can thus address 1 megabyte of memory. Every memory reference involves a physical memory address calculated by adding to the shifted segment address a 16 bit offset derived from the instruction operand. Thus the 8086 requires addresses to be split into segment parts and offset parts. These correspond to load and link addresses respectively.

4.8.4 Link Addresses

The link address of a psect is to the linker the address that will be used in all normal relocation calculations, i.e. if a relocation requires that the base address of a psect be added to a memory location then it will be the link address of that psect that will be added. Similarly if relocation by an external symbol is required,

it will be performed by adding to the memory location the base address of the psect in which the external symbol is defined plus the offset of that symbol from the base of the psect. These calculations correspond to calculating offset addresses for the 8086 instructions.

4.8.5 Load Addresses

The load address of a psect is to the linker the physical address of the psect, i.e. its actual location in the excutable image file. At the time an excutable program is loaded into memory by an operating system its physical location in memory will be determined by the operating system, but the relative position of data in the file will be maintained. We will discuss later what action is required to compensate for the fact that the linker cannot know where in memory the program will execute. As far as the linker is concerned the physical address of a psect is its address relative to the beginning of the executable file.

Load addresses are thus the basis of the segment part of 8086 addresses; however they are not the same, since load addresses, like all addresses handled by the linker, are linear 32 bit values, while the segment part of an address is only the top 16 bits of a 20 bit value. Thus to convert a load address to a segment address the linker must right shift it 4 bits, or divide it by 16. 16 happens to be the relocatability quantum of 8086 psects, and it is via the *reloc=16* flag on a *.psect* directive that the linker knows to divide by 16 to convert a load address to a segment address. For this reason it is important to have a *reloc=16* flag on all initial psect definitions (it is only necessary to do this in one module for each global psect).

4.8.6 Linking and the -P option

When linking an 8086 program it is very important to understand what arguments should be given to the linkers **-P** option in order to correctly link the program.

A small model 8086 program has two segments; the *code* (or *text*) segment and the *data* segment. The code segment is addressed by the **CS** register, while the data segment is addressed by the **DS** register. The **SS** register is also set to point to the data segment (i.e. has the same value as the DS register) since the stack is a part of the data segment. The *psects* that make up a small model program are each placed into one of the two segments. The C compiler uses 3 psects for small model programs. These are the **_TEXT** psect, which is placed into the code segment, and the **data** and **bss** psects, which are both placed in the data segment (the data psect comes before the bss psect).

The offset addresses of each *segment* begin at 0, since the segment register associated with each segment points to the base of that segment. Thus the *link* addresses of the _TEXT and data *psects* will both be 0, since each starts at the beginning of their segments. The link address of the bss psect will be equal to the size of the data psect, rounded up to a 16 byte multiple.

The load addresses of each psect must, however, be different, since it would be impossible (or at least useless) for the _TEXT and data psects to occupy the same physical memory (or file) space. This is the reason for allowing the specification of link and load addresses separately. While two psects may have

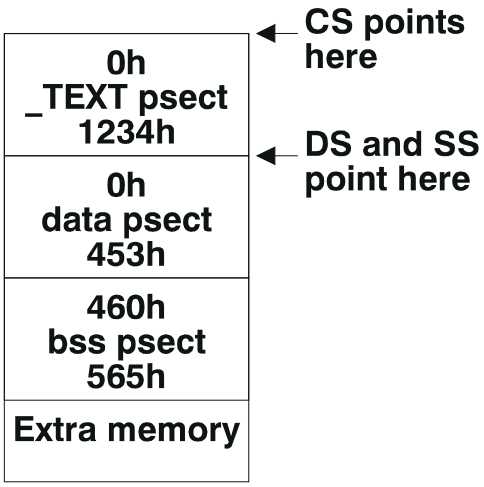


Figure 4-4; Small model memory layout

overlapping link addresses their load addresses must be disjoint. The diagram in figure 4-4 illustrates the arrangement of addresses for a program with a _TEXT psect 1234(hex) bytes long, a data psect 453(hex) bytes long, and a bss psect 105(hex) bytes long.

The values inside the boxes represent offset addresses. When linking this program the C compiler will supply a -P option like this:

```
-P_TEXT=0,text,data=0/,bss
```

This sets up the psect link and load addresses as follows: the **_TEXT** psect is given a link address of 0. Since its load address is unspecified it defaults to the same as the link address. The **text** psect is semantically equivalent to the **_TEXT** psect and is included here for backwards compatibility with older Pacific C compilers. It has no '=' sign so it is concatenated for both link and load addresses with **_TEXT**. In practice the **text** psect will always be empty.

The **data** psect has its link address set to 0 (remember the link address appears immediately after the '=' sign). Following the link address is a '/' character which introduces a load address, however the load address is omitted. This indicates to the linker that the load address should follow on from the previous psects load address, i.e. that the **data** psect should immediately follow the **text** psect in physical memory. The **bss** psect has no link or load address specified, so it is concatenated with **data**. Thus this achieves the layout shown in fig. 2. Note that as far as the linker is concerned the program is to be loaded in memory

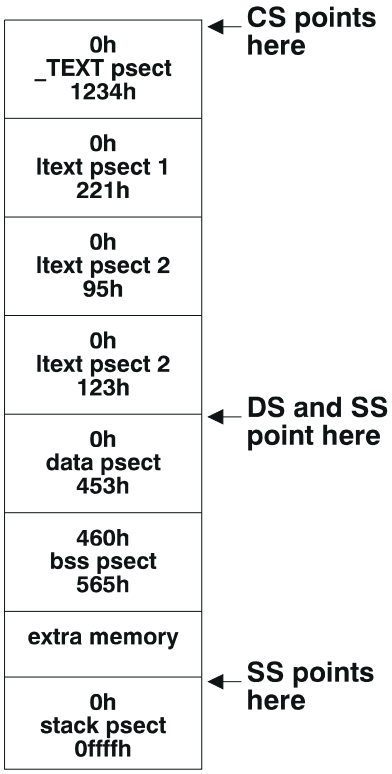


Figure 4-5; Large model memory layout

at physical 0. If it is to be run under MS-DOS then it will of course be loaded somewhere else. This does not matter in the case of the small model program as all addresses within the program are offset addresses, derived from link addresses.

A large model program has more than two segments: it has the same data segment as a small model program, but a separate stack segment and multiple code segments. One of the code segments corresponds to the small model code segment in that it contains the **_TEXT** and **text** psects but there are additional code segments each containing one local psect named **ltext**. There is one of these **ltext** psects for each C source module. A typical large model program might be laid out as in figure 4-5.

While the CS register is shown as pointing to the base of the _TEXT psect, during execution it will point to the base of other code segments when they are executing. This is achieved via the far (or inter-segment) call and return instructions used in large model programs. The DS register always points to the data

segment, and the SS register always points to the stack segment. If the amount of memory available for the stack and data segments combined is less than 64K then the SS register will be set to the base of the data segment. In this case the stack pointer SP will be initialized with a value less than 64K. The -P option to the linker to achieve this layout would be:

```
-P_TEXT=0,text,CODE=0/,data=0/,bss,stack=0/
```

Compare this with the -P option for the small model and note the addition of the **CODE** class-name and the **stack** psect. The **CODE** class embraces all the **ltext** psects, and allows all the **ltext** psects to be referred to by one entry in the -P option list. The zero link address and concatenated load address given after **CODE** is applied to each **ltext** psect in turn. The **stack** psect is also given a link address of 0 and a load address concatenated with **bss**. It will be noticed that all psects are concatenated with the previous for load address. This is of course required to ensure that all psects occupy disjoint physical memory.

Since the large model does embed segment addresses in the code, it is necessary for any such addresses to be relocated (or fixed up) when MS-DOS loads the program into memory. MS-DOS is quite happy to perform the fixups, but it must be given the information on what addresses to fixup. This is achieved by the C compiler in a two step process. The linker has a **-LM** option which directs the linker to retain in the output file information about what addresses were subjected to a segment relocation. This information is then used by objtohex to build a list of addresses in segment:offset form which is then placed into the .EXE file In the .EXE file the addresses are placed in the .EXE header. MS-DOS goes through this list after loading the program and adjusts the memory locations specified. Each word location has the programs base segment address added to it.

4.9 Absolute Variables

A global or static variable can be located at an absolute address by following its declaration with the construct @ address, for example:

```
volatile unsigned char portvar @ 0x1020;
```

will declare a variable called portvar located at 1020h. Note that the compiler does not reserve any storage, but merely equates the variable to that address, the compiler generated assembler will include a line of the form:

```
_Portvar  equ  1020h
```

Note that the compiler and linker do not make any checks for overlap of absolute variables with other variables of any kind, so it is entirely the programmer's responsibility to ensure that absolute variables are allocated only in memory not in use for other purposes.

4.10 Port Type Qualifier

The 8086 I/O ports may be accessed directly via port type qualifier. For example, consider the following declaration:

```
port unsigned char *  pptr;

port unsigned char   io_port @ 0xE0;
```

The variable pptr is a pointer to an 8 bit wide port and the variable io_port is mapped directly onto port 0E0H and will be accessed using the appropriate IN and OUT instructions. For example, the statements

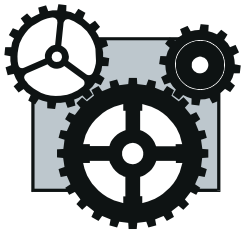
```
io_port |= 0x40;

*pptr = 0x10;
```

will generate the following code:

```
;x.c: 8: io_port |= 0x40;
        mov     dx,#224
        in      al,[dx]
        or      al,#64
        out     [dx],al
;x.c: 10: *pptr = 0x10;
        mov     al,#16
        mov     dx,_pptr
        out     [dx],al
```

8086 Assembler Reference Manual



5.1 Introduction

The assembler incorporated in the Pacific C compiler system is a full-featured relocating macro assembler. The syntax of the language accepted is not the same as the Intel (or Microsoft) assembler, but the opcode mnemonics are largely compatible. A discussion of the differences in detail follows later.

AS86 is an optimizing assembler; it will produce the smallest possible code for given input by producing short jumps and using short addressing forms wherever possible. This feature does require, however, that more than two passes are made over the source code. This results in slower assembly times but an option is provided (-Q) which will perform a quick assembly, using only two passes, and insert no-ops wherever necessary to compensate for optimizations performed on the second pass which were not done on the first pass (e.g. short branches forward).

5.2 Usage

The assembler is named AS86, and is invoked as follows:

```
AS86 options files ...
```

The files are one or more assembler source files which will be assembled, but note that all the files are assembled as one, not as separate files. To assemble separate files, the assembler must be invoked on each file separately. The options are zero or more options from the following list:

5.2.1 -Q

The -Q option provides a quick assembly, as discussed above.

5.2.2 -U

Treat undefined symbols as external. The -U option will suppress error messages relating to undefined symbols. Such symbols are treated as externals in any case. The use of this option will not alter the object code generated, but merely serves to suppress the error messages.

5.2.3 -O*file*

Place the object code in *file*. The default object file name is constructed from the name of the first source file. Any suffix or file type (i.e. anything following the rightmost dot ('.') in the name is stripped, and the suffix .obj appended. Thus the command

AS86 file1.as file2.as

will produce an object file called *file1.obj*. The use of the -O option will override this default convention, allowing the object file to be arbitrarily named. For example:

AS86 -Ox.obj file1.as

will place the object code in *x.obj*.

5.2.4 -L*list*

Place an assembly listing in *list*, or on standard output if *list* is null.

5.2.5 -W*width*

The listing is to be formatted for a printer of given *width.* The default width is 80 columns if the listing is going to a device (e.g. a printer) or 132 columns if going to a file.

5.2.6 -X

Do not include local symbols in the output file. This reduces the size of the object file. Local symbols are not used by the linker and are of use only for debugging.

5.2.7 -E

Suppress the listing of errors on the console. Error codes will still be included in the listing.

5.2.8 -M

Generate Intel compatible object code rather than HI-TECH Software proprietary object code. The object module thus generated will be able to be linked with the MS-DOS linker or another linker accepting standard Intel object code format, but not the HI-TECH linker.

5.2.9 -1,-2,-3,-4

Permit the assembly of the extra instructions supported by the the 80186, 286, 386 or 486 processors. If an instruction is encountered that is not supported by the selected processor, an error will occur. This facility is also available with the **.186**, **.286**, **.386** and **.486** directives.

5.2.10 -N

Suppress checking for arithmetic overflow. The assembler performs arithmetic internally in 32 bits but reports overflows out of the lower 16 bits. The -N option suppresses this checking.

5.2.11 -I

Force listing of include files, even if **.nolist** directives are encountered.

5.2.12 -S

Suppress error messages which would be generated through initializing a byte memory location with a quantity which will not fit in 8 bits.

5.2.13 -C

Generate cross reference information. The raw cross reference data will be written to a file derived from the name of the first source file, with the suffix **.crf**. The cross reference generator CREF must then be used to produce the actual cross reference listing.

5.3 The Assembly Language

The assembly langugage follows the Intel mnemonics, with the following differences:

- The opcode forms such as movsb and movsw are not provided; to indicate the operand size where it is not otherwise defined the additional operands **byte** or **word** should be used, e.g. **movs word** instead of **movsw**.
- Since no provision is made for declaring labels or functions as far or near, additional forms of the **jmp, call** and **ret** opcodes are provided to indicate inter-segment control transfers. These are formed by adding an **f** suffix to the opcode, e.g. **callf.**
- Additional opcodes are provided which allow long conditional branches to be indicated. These opcodes are formed by substituting the **j** in a branch opcode with **br,** e.g. **brnz.** This will assemble to a short branch if possible, otherwise to a short branch of the opposite condition around a long unconditional jump.

5.3.1 Symbols

The symbols (labels) accepted by the assembler may be of any length, and all characters are significant. The characters used to form a symbol may be chosen from the upper and lower case alphabetics, the digits 0-9, and the special symbols underscore (‘_’), dollar (‘$’) and question mark (‘?’). The first character may not be numeric. Upper and lower case are distinct. Note that there is no distinction between labels appearing in code or in data; there is no concept of a variable with attributes. The following are all legal and distinct symbols.

```
An_identifier
an_identifier
an_identifier1
$$$
?$_123455
```

Note that the symbol $ is special (representing the current location) and may not be used as a label. Nor may any opcode or pseudo-op mnemonic, register name or condition code name.

See the discussion of addressing modes below for differences in referencing symbols, compared with the Intel or Microsoft assemblers.

5.3.1.1 Temporary Labels

The assembler implements a system of temporary labels, useful for use within a localized section of code. These help eliminate the need to generate names for labels which are referenced only in the immediate vicinity of their definition, for example where a loop is implemented.

A temporary label takes the form of a digit string. A reference to such a label requires the same digit string, plus an appended b or f to signify a backward or forward reference respectively. Here is an example of the use of such labels.

```
entry_point:      ;referenced elsewhere
        mov       cl,#10
1:      dec       bx
        jnz       2f        ;branch forward
        mov       bx,#8
        loop      1b        ;loop back to 1:
        jmp       1f        ;branch forward
```

Table 5-1; Constant radix suffixes

Character	Radix	Name
B	2	binary
O	8	octal
Q	8	octal
o	8	octal
q	8	octal
H	16	hexadecimal
h	16	hexadecimal

```
2:      call    fred    ;here from jnz 2f
1:      ret             ;here from jr 1f
```

The digit string may be any positive decimal number 0 to 65535. A temporary label value may be re-used any number of times. Where a reference to e.g. 1b is made, this will reference the closest label 1: found by looking backwards from the current point in the file. Similarly 23f will reference the first label 23: found by looking forwards from the current point in the file.

5.3.2 Constants

Constants may be entered in one of the radices 2, 8, 10 or 16. The default is 10. Constants in the other radices may be denoted by a trailing character drawn from the set listed in table 5-1.

Note that a lower case b may not be used to indicate a binary number, since 1b is a backward reference to a temporary label 1:.

5.3.2.1 Character Constants

A character constant is a single character enclosed in single quotes (‘).

5.3.2.2 Floating Constants

A floating constant in the usual notation (e.g. 1.234 or 1234e-3) may be used as the operand to a .FLOAT pseudo-op.

5.3.3 Expressions

Expressions are constructed from symbols, constants and operators.

5.3.3.1 Operators

The operators shown in table 2-2 may be used in expressions.

5.3.3.2 Relocatability

AS86 produces object code which is relocatable; this means that it is not necessary to specify at assembly time where the code is to be located in memory. It is possible to do so, however the preferred approach is to use relocatable program sections or psects. A psect is a named section of the program, in which code or data may be defined at assembly time. All parts of a psect will be loaded contiguously into memory, even if they were defined in separate files, or in the same file but separated by code for another psect. For example, the code below will load some executable instructions into the psect named text, and some data bytes into the data psect.

```
        .psect  text, global

alabel: mov     bx,#astring
```

Table 5-2; AS86 expression operators

Operator	Meaning
&	Bitwise AND
*****	Multiplication
+	Addition
-	Subtraction
.and.	Bitwise AND
.eq.	Equality test
.gt.	Signed greater than
.high.	Hi byte of operand
.low.	Low byte of operand
.lt.	Signed less than
.mod.	Modulus
.not.	Bitwise complement
.or.	Bitwise or
.shl.	Shift left
.shr.	Shift right
.ult.	Unsigned less than
.ugt.	Unsigned greater than
.xor.	Exclusive or
/	Divison
<	Signed less than
=	Equality
>	Signed greater than
^	Bitwise or
.seg.	The segment part of an address
seg	Same as .seg.


```
        call    putit
        mov     bx,#anotherstring

        psect   data, global
astring:
        .byte   'A string of chars', 0
anotherstring:
```

```
        .byte   'Another string', 0

        .psect  text
putit:
        mov     al,[bx]
        or      al,al
        bz      1f
        call    outchar
        inc     bx
        jmp     putit
1:      ret
```

Note that even though the two blocks of code in the text psect are separated by a block in the data psect, the two text psect blocks will be contiguous when loaded by the linker. The instruction "mov bx,#another-string" will fall through to the label "putit:" during execution. The actual location in memory of the two psects will be determined by the linker. See the linker manual for information on how psect addresses are determined.

A label defined in a psect is said to be relocatable, i.e. its actual memory address is not determined at assembly time. Note that this does not apply if the label is in the default (unnamed) psect, or in a psect declared absolute (see the .PSECT pseudo-op description below). Any labels declared in an absolute psect will be absolute, i.e. their address will be determined by the assembler.

5.3.4 Pseudo-ops

The pseudo-ops are described below.

5.3.4.1 .BYTE

This pseudo-op should be followed by a comma-separated list of expressions, which will be assembled into sequential byte locations. Each expression must have a value between -128 and 255 inclusive, or can be a multi-character constant. Example:

.BYTE 10, 20, 'axl', 0FFH

5.3.4.2 .WORD

This operates in a similar fashion to .BYTE, except that it assembles expressions into words, which may range -32767 to 65535, and character strings are not acceptable. Example:

.WORD -1, 3664H, 'A', 3777Q

5.3.4.3 .DWORD

The .DWORD pseudo-op is similar to .WORD, but assembles double word quantities (32 bits).

5.3.4.4 .FLOAT

This pseudo-op will initialize a memory location to the binary representation of the floating point number given as an argument. The default conversion is to a 4 byte floating number, but the variant .FLOAT8 will convert to an 8 byte number. Example:

```
        .FLOAT   23.12e4
```

5.3.4.5 .BLKB

This pseudo-op reserves memory locations without initializing them. Its operand is an absolute expression, representing the number of bytes to be reserved. This expression is added to the current location counter. Note however that locations reserved by .BLKB may be initialized to zero by the linker if the reserved locations are in the middle of the program. Example:

```
.BLKB   20h     ; reserve 16 bytes of memory
```


5.3.4.6 EQU

Equ sets the value of a symbol on the left of EQU to the expression on the right. It is illegal to set the value of a symbol which is already defined. Example:

```
SIZE  equ   46
```

5.3.4.7 SET

This is identical to EQU except that it may redefine existing symbols. Example:

```
SIZE  set   48
```

5.3.4.8 .END

The end of an assembly is signified by the end of the source file, or the .END pseudo-op. The .END pseudo-op may optionally be followed by an expression which will define the start address of the program. Only one start address may be defined per program, and the linker will complain if there are more. Example:

```
.END    somelabel
```

5.3.4.9 IF

Conditional assembly is introduced by the .IF pseudo-op. The operand to .IF must be an absolute expression. If its value is false (i.e. zero) the code following the .IF up to the corresponding .ENDIF pseudo-op will not be assembled. .IF/.ENDIF pairs may be nested. Example:

```
 .IF     Debug
call    trace    ;trace execution
 .ENDIF
```

5.3.4.10 .ENDIF

See .IF.

5.3.4.11 .ENDM

See .MACRO.

5.3.4.12 .PSECT

This pseudo-op allows specification of relocatable program sections. Its arguments are a psect name, optionally followed by a list of psect flags. The psect name is a symbol constructed according to the same rules as for labels, however a psect may have the same name as a label without conflict. Psect names are recognized only after a .PSECT pseudo-op. The psect flags are as follows:

5.3.4.12.1 ABS

Psect is absolute

5.3.4.12.2 GLOBAL

Psect is global

5.3.4.12.3 LOCAL

Psect is not global

5.3.4.12.4 OVRLD

Psect is to be overlapped by linker

5.3.4.12.5 PURE

Psect is to be read-only

5.3.4.12.6 SIZE

Allows max size of the psect to be set, e.g. SIZE=65535

5.3.4.12.7 RELOC

Allows relocation granularity of the psect to be set, e.g. for the 8086 since all segments must start on a 16 byte boundary, most psects will have a RELOC=16 flag.

5.3.4.12.8 CLASS

Allows local psects to have a class-name associated with them, for use in -P options to the linker. E.g. CLASS=txtgrp

5.3.4.12.9 STACKSEG

This flag is effective only when generating Intel object code and is used to let the linker know that this psect is to be a stack segment. The MS-DOS linker will set the initial stack segment value to the base of this psect.

5.3.4.12.10 USE32

This psect flag informs the assembler that code in this psect will be executed in an 80386/486 segment with the D bit set, and thus the default operand and address size will be 32 bits. By default the assembler assumes 16 bit operands and addressing. This flag may be used only if a **.386** or **.486** directive or a -3 or -4 option has been used.

If a psect is global, the linker will merge it with any other global psects of the same name from other modules. Local psects will be treated as distinct from any other psect from another module. Psects are global by default.

By default the linker concatenates code within a psect from various modules. If a psect is specified as OVRLD, the linker will overlap each module's contribution to that psect. This is particularly useful when linking modules which initialize e.g. interrupt vectors.

The PURE flag instructs the linker that the psect is to be made read-only at run time. The usefulness of this flag depends on the ability of the operating system to enforce the requirement.

The ABS flag makes a psect absolute. The psect will be loaded at zero. Examples:

```
.PSECT   text, global, pure
.PSECT   data, global
.PSECT   vectors, ovrld, reloc=100h
.PSECT   ltext, local, class=txtgrp,reloc=16,size=65535
```

5.3.4.12.11 .GROUP

This pseudo-op is used only with the -M option, i.e. when generating Intel compatible object code. It allows the naming of a group and associate psects. The significance of a group is that the MS-DOS linker will ensure that all members of a group are located within the same 64K segment. Even more important is that all addressing within segments in that group will be calculated with the group as the reference frame, rather than the psect itself. This is necessary with the Intel object code relocation scheme to ensure correct relocation. Example:

.GROUPDGROUP,data,bss

5.3.4.13 .GLOBL

The directive .GLOBL should be followed by one more symbols (comma separated) which will be treated by the assembler as global symbols, either internal or external depending on whether they are defined within the current module or not. Example:

```
.GLOBL  label1, putchar, _printf
```

5.3.4.14 .LOC

An .LOC pseudo-op sets the the location counter in the current psect to its operand, which must be an absolute expression. This directive does not change the current psect. To generate absolute code, you must use a **.PSECT** directive, specifiying **ABS,OVRLD**. Example:

```
.LOC    100H
```

5.3.4.15 .MACRO and .ENDM

These directives provide for the definition of macros. The MACRO directive should be preceded by the macro name and followed by a comma-separated list of formal parameters. When the macro is used, the macro name should be used in the same manner as a machine opcode, followed by a list of arguments to be substituted for the formal parameters. For example:

```
copy     .MACRO   par1,par2,par3    ;copy par1 to par2
         mov      ax,par1&par3
         mov      par2&par3,ax
         .ENDM

         copy     loc1,loc2

;        expands to:

         mov      ax,loc1
         mov      loc2,ax

         copy     loc1,loc2,x_

;        expands to:

         mov      ax,loc1_x
```

```
          mov       loc2_x,ax
```

Points to note in the above example: in the first expansion the third argument was unspecified, so nothing was substituted for it; the & character is used to permit the concatenation of macro parameters with other text, but is removed in the actual expansion; this enabled the **_x** argument to the third parameter in the second expansion to form part of the operands.

The NUL operator may be used within a macro to test a macro argument. A comment may be suppressed within the expansion of a macro (thus saving space in the macro storage) by opening the comment with a double semicolon (;;).

5.3.4.16 .LOCAL

The LOCAL directive allows unique labels to be defined for each expansion of a given macro. Any symbols listed after the LOCAL directive will have a unique assembler-generated symbol substituted for them when the macro is expanded. For example:

```
xxx       .MACRO    src,dst,cnt
          .LOCAL    back
          mov       cx,#cnt
          mov       ah,#0
back:     mov       al,[src]
          mov       [dst],ax
          inc       src
          add       dst,#2
          loop      back
          .ENDM

          xxx       si,di,23

;         expands to

          mov       cx,#23
          mov       ah,#0
??0001:   mov       al,[si]
          mov       [di],ax
          inc       si
          add       di,#2
          loop      ??0001
```

5.3.4.17 .REPT

The .REPT directive temporarily defines an unnamed macro then expands it a number of times as determined by its argument. For example:

```
        .REPT   3
        movs    word
        .ENDM

;       expands to

        movs    word
        movs    word
        movs    word
```

5.3.4.18 .IRP and .IRPC

The .IRP and .IRPC directives operate similarly to .REPT, however instead of repeating the block a fixed number of times, it is repeated once for each member of an argument list. In the case of .IRP the list is a conventional macro argument list, in the case or .IRPC it is each character in one argument. For each repetition the argument is substituted for one formal parameter. For example:

```
        .IRP    arg,lab1,lab2,#23
        mov     ax,arg
        stos    word
        .ENDM

;       expands to

        mov     ax,lab1
        stos    word
        mov     ax,lab2
        stos    word
        mov     ax,#23
        stos    word

        .IRPC   arg,1982
        imul    ax,#10
        add     ax,#arg
        .ENDM
```

```
;        expands to

         imul     ax,#10
         add      ax,#1
         imul     ax,#10
         add      ax,#9
         imul     ax,#10
         add      ax,#8
         imul     ax,#10
         add      ax,#2
```

5.3.4.19 Macro Invocations

When invoking a macro, the argument list must be comma-separated. If it is desired to include a comma (or other delimiter such as space) in an argument then angle brackets (< and >) may be used to quote an argument. In addition the exclamation mark (!) may be used to quote a single character. The character immediately following the exclamation mark will be passed into the macro argument even if it is e.g. a semicolon, normally a comment indicator.

If an argument is preceded by a percent sign (%) then that argument will be evaluated as a expression and passed as a decimal number, rather than as a string. This is useful if evaluation of the argument inside the macro body would yield a different result.

5.3.4.20 .TITLE

This pseudo-op sets a title string to be used on the assembler listing. White space following the pseudo-op is skipped, then the rest of the line used as the title. This pseudo-op will not be listed. E.g.

```
         .TITLE   Widget control program
```

5.3.4.21 .SUBTTL

Subttl defines a sub-title for the program listing. The argument to it is similar to that for .TITLE. .SUBTTL will also cause a .PAGE to be executed. This pseudo-op will not appear in the listing. Example:

```
         .SUBTTL  Initialization phase
```

5.3.4.22 .PAGE

This causes a new page to be started in the listing. .PAGE is never listed.

5.3.4.23 .INCLUDE

This directive allows the inclusion of other files in the assembly. For example:

```
         .INCLUDE macrodefs.i
```

will cause the text of macrodefs.i to be included in the assembly at the point at which the directive appears. The .INCLUDE directive will not be listed.

5.3.4.24 .LIST

This directive and the corresponding .NOLIST turn listing on and off respectively. Note that if these directives are used in a macro or include file, the previous listing state will be restored on exit from the macro or include file. This allows the selective listing or otherwise of macros. For example:

```
fred    .MACRO
        .NOLIST
        ;This line will not appear in the listing file.
        .ENDM

        .LIST   ;turn listing on
        fred    ;expand the macro; this line appears
        ;       and so does this line
```


5.3.5 Addressing Modes

The 8086 (16 bit) addressing modes are represented as follows:

5.3.5.1 Register

This is identical to the Intel register addressing mode. Example:

```
        mov     ax,bx
```

5.3.5.2 Immediate

Immediate addressing is indicated by a '#' character. This also applies to **int** instructions. Examples:

```
        int     #0E0h
        mov     al,#2
        mov     si,#alabel
```

If you are used to the Intel or Microsoft assemblers then it is important to understand the difference in the way immediate addressing is handled. In the Microsoft style of assembler immediate addressing is assumed when a constant (e.g. 345) or a label (i.e. a label in code, as opposed to a label identifying data, which is treated as a *variable*) is referenced. When a variable is referenced then direct addressing is assumed. Thus the assembler (and the programmer) must know what kind of operand is being addressed to decide whether immediate or direct addressing is to be used. To force immediate addressing with a variable the operator **offset** must be used.

The HI-TECH assembler always uses direct addressing when a label or constant is referenced unless the '#' character is present. In this case immediate addressing is always used.

5.3.5.3 Direct

Direct addressing requires simply the address expression of the target. A **byte** or **word** operand may be required, as with other memory addressing modes, where the operand size is not otherwise determinable. Examples:

```
inc     fred
mov     cs:alabel+4,#5,byte
mov     ax,some_address
mov     10,cs
```

5.3.5.4 Register indirect

Square brackets are used in general to denote indirect addressing; register indirect requires the register name to be enclosed in brackets, e.g.

```
mov     al,[si]
```

5.3.5.5 Indexed

Indexed addressing in general, including indexing from the registers **bp, bx, si** and **di** requires the one or two register names, each enclosed in square brackets, to be preceded by an offset. The usual restrictions on which registers may be used in a double indexed mode apply. The order of the registers is not important. Examples:

```
mov     al,10[bx]
mov     an_offset[bp][si],#23h,word
mov     [si][bx],dx
```

Note here that the Microsoft assembler requires constant offsets to be inside the square brackets, e.g. [si+10], while variable names should be outside the brackets, e.g. fred[si]. The HI-TECH assembler always expects to see offset expressions outside the brackets (it is normal for the offset expression to precede the brackets, but not essential), e.g. 10[si] or fred[si].

5.3.5.6 String

String instructions require a **byte** or **word** operand to permit the assembler to determine the size of the data to be moved. No other operands are acceptable to the string instructions, however if a segment override is required, it can be added. Examples:

```
rep movs byte
cmps word,es:
```

5.3.5.7 I/O

I/O addressing is treated in the same manner as direct addressing, or register indirect addressing (for **dx** only). Examples:

```
in      al,a_port
out     [dx],ax
```

5.3.5.8 Segment prefixes

Segment prefixes may be applied to any memory address in the usual manner. e.g.

```
        mov     ss:24,#4,word
```


5.3.5.9 Jump

The operands to jump and call instructions are normally treated as direct addressing, e.g.

```
        jmp     jail    ;go directly to jail
```

Indirect addressing is indicated by enclosing a direct address in square brackets, e.g.

```
        jmpf    [somewhere+4]
```

Indexed addressing is as described above.

5.3.6 80386 Addressing Modes

The addresing modes for the 80386 are basically the same as above, except that 32 bit offsets and direct addresses are used instead of 16 bits. The indexed addressing form is however completely different. Refer to an 80386 programming handbook for full details, however the basic modes are summarized below. Note that any 32 bit general register plus ESP may be used in 386 indexed addresses.

5.3.6.1 Segment Registers

The 80386 has two extra segment registers (FS and GS) which can be used in the same manner as normal 8086 segment registers. **FS:** and **GS:** can be included as segment overrides in instructions in the normal manner. Two new instructions, **LFS** and **LGS** are present and used in the same manner as **LES**. Example:

```
lfs     eax,10[esp]
mov     ax,gs:[bx][di]
```

5.3.6.2 String

The string instructions take the same form as on the 8086, except that 32 bit quantities may also be moved using the **dword** operand. Segment overrides are used in the normal manner. Example:

```
cmps    byte
stos    dword
movs    word,gs:
```

5.3.6.3 Single register

This form has a single register optionally with an offset. Example:

```
mov     eax,24[ebx]
```

5.3.6.4 Double indexed

This form allows two index registers, optionally with an offset. Example:

```
mov     ebp,alabel[edx][esi]
```

5.3.6.5 Scaled index

This form allows one register to be multiplied by 2, 4 or 8 before being used in the address. Example:

```
lea     esi,4[eax][edx*4
```

5.4 Diagnostics

An error message will be written on the standard error stream for each error encountered in the assembly. This message identifies the file name and line number and describes the error. In addition the line in the listing where the error occurred will be flagged with a single character to indicate the error. The following listing presents the error messages in alphabetical order with an explanation of each.

32 bit addressing illegal
32 bit addressing (i.e. using 32 bit registers in an index expression) is illegal unless generating 386 or 486 code. Use a .386 or .486 directive at the beginning of the file.

80186 instruction/addressing mode
An instruction or addressing mode not supported by the 8088/8086 was encountered. Use a .186, .286. .386 or .486 directive to allow these.

80286 instruction
An instruction peculiar to the 80286 encountered. Use a .286. .386 or .486 directive to allow these.

80386 instruction/addressing mode
An instruction or addressing mode peculiar to the 80386 encountered. Use a .386 or .486 directive to allow these.
80486 instruction/addressing mode
An instruction or addressing mode peculiar to the 80486 encountered. Use a .486 directive to allow these.
absolute expression required
An absolute expression is required in this context.
bad character const
This character constant is badly formed.
bad operand size
This instruction requires an operand size of byte. Any other size is illegal.
conflicting operand sizes
This instruction has more than one operand size specified. They are incompatible.
division by zero
An expression resulted in a division by zero.
duplicate base register
Only one base register, i.e. only one of BP and BX may be used in a 16 bit indexed addressing form.
duplicate displacement in operand
The operand on this instruction has two offsets. Only one is permitted.
duplicate index register
Only one index register (SI or DI) may be used in a 16 bit indexed form.
esp not permitted as index register
ESP is not permitted as an index register.
expression error
There is a syntax error in this expression.
flag * unknown
This option used on a “PSECT” directive is unknown to the assembler.
floating number expected
The arguments to the “DEFF” pseudo-op must be valid floating point numbers.
garbage after operands
garbage on end of line
There were non-blank and non-comment characters after the end of the operands for this instruction. Note that a comment must be started with a semicolon.
illegal addressing mode
This addressing mode is not permitted.
illegal stack index
The index used in a cop-processor instruction must be an absolute value 0-7.
illegal switch *
This command line option was not understood.

indirection on illegal register
Indirection on this register is not possible.
insufficient memory for macro def'n
out of memory
The assembler has run out of memory. Try splitting your source file into a number of smaller modules.
jump target out of range
The address that this jump refers to is too far away to be reached by this jump. Use a longer jump form, or a short branch around a direct jump.
lexical error
An unrecognized character or token has been seen in the input.
local illegal outside macros
The "LOCAL" directive is only legal inside macros. It defines local labels that will be unique for each invocation of the macro.
macro argument after % must be absolute
If the charactyer "%" is used to force evaluation of macro argument, the argument must be an expression that evaluates to an absolute constant.
macro argument may not appear after local
The list of labels after the directive "LOCAL" may include any of the formal parameters to the macro.
macro expansions nested too deep
include files nested too deep
Macro expansions and include file handling have filled up the assembler's internal stack.
missing ')'
A closing parenthesis was missing from this expression.
missing ']'
A closing square bracket was missing from this expression.
mixed 16 and 32 bit index registers
An indexed addressing mode must use only 16 bit or only 32 bit registers. They may not be used together.
multiply defined symbol *
This symbol has been defined in more than one place in this module.
no arg to -o
The assembler requires that an output file name argument be supplied after the "-O" option. No space should be left between the -O and the filename.
no file argument
There was no file argument to the assembler.
no space for macro def'n
The assembler has run out of memory.
one segment override only permitted
Only one segment override is permitted in an instruction.
oops! -ve number of nops required!
An internal error has occured. Contact HI-TECH.

operand error
The operand to this opcode is invalid. Check you Z80 reference manual.
operand size undefined
undefined operand size
The size of the operand to this instruction was not defined. Use ",byte" or ",word", or ",dword" as appropriate.
page width must be >= 41
The listing page widthe must be at least 41 characters. Any less will not allow a properly formatted listing to be produced.
phase error
phase error in macro args
The assembler has detected a difference in the definition of a symbol on the first and a subsequent pass.
pop immediate illegal
It is not possible to pop into an immediate value.
psect * in more than one group
This psect has been defined to be in more than one group.
psect may not be local and global
A psect may not be declared to be local if it has already been declared to be (default) global.
psect reloc redefined
The relocatability of this psect has been defined differently in two or more places.
psect selector redefined
The selector associated with this psect has been defined differently in two or more places.
psect size redefined
The maximum size of this psect has been defined differently in two or more places.
radix value out of range
The value given for the current radix is out of range. It must be between 2 and 16.
relocation error
It is not possible to add together two relocatable quantities. A constant may be added to a relocatable value, and two relocatable addresses in the same psect may be subtracted. An absolute value must be used in various places where the assembler must know a value at assembly time.
remsym error
Internal error.
rept argument must be >= 0
The argument to a "REPT" directive must be greater than zero.
scale value invalid
The scale value in the operand is invalid. It may only be 1, 2, 4 or 8.
scale value must be a constant
The scale value in an operand must be an absolute constant.

size error
You have attempted to store a value in a space that is too small, e.g. trying to initialize a byte location with an address that is 16 bits.

syntax error
A syntax error has been detexted. This could be caused a number of things.

too many errors
There were too many errors to continue.

too many macro parameters
There are too many macro parameters on this macro definition.

too many symbols
There are too many symbols to fit into the symbol table.

too many temporary labels

too may symbols
There are too many symbols for the assemblers symbol table. Reduce the number of symbols in your program.

undefined symbol *
The named symbol is not defined, and has not been specified “GLOBAL”.

undefined temporary label
A temporary label has been referenced that is not defined. Note that a temporary label must have a number >= 0.

unknown directive
This directive is not known to the assembler.

write error on object file
An error was reported when the assembler was attempting to write an object file. This probably means there is not enough disk space.

Lucifer - A Source Level Debugger

6.1 Introduction

Lucifer is a source level debugger for use with the Pacific C compiler for the 8086 family. This version of Lucifer is capable of operating with 8086 family chips up to and including the 80286, it will work correctly in real mode on an 80386 or 80486 but does not handle the 80386 instruction set.

Lucifer provides a debugging environment which allows source code display, symbolic disassembly, memory dumps, instruction single stepping and tracing, breakpoints, and many other functions.

Lucifer is provided with the standard version of Pacific C for debugging DOS executable programs. With the ROM development edition of Pacific C you also get a remote version of Lucifer for debugging in a target system, connected via a serial link to your PC. When using PPD, the appropriate debugger will be selected depending on what kind of executable program you are generating. See the chapter “ROM Development with Pacific C” for more information on remote debugging.

6.2 Usage

To use Lucifer you should compile your program with a **-G** option. This will produce a symbol file with line number and filename symbols included. If you use a **-H** option you will get a symbol file but without the filename and line number symbols. If using PPD, select the **Source Level Debug Info** menu entry from the **Options** menu. From within PPD you can also invoke Lucifer from the **Run** menu, rather than using the command line as described below.

To compile a program “TEST.C” for use with the debugger, use the command:

```
PACC -GTEST.SYM TEST.C
```

Then invoke Lucifer as follows:

```
LUCIFER TEST.EXE TEST.SYM [arguments]
```

Lucifer will display a signon message, then attempt to load the specified executable and symbol files. It then prints a colon prompt ‘:’ and waits for commands. For a list of commands, type the command **?**. Note that all commands should be in lower case. Symbols should be entered in exactly the same case as they were defined. Any command line arguments typed after the name of the symbol file are passed through to the user program as *argv[1]*, etc. Where an expression is required, it may be of the form:

```
symbol_name
symbol+hexnum
symbol-hexnum
$hexnum:[$hexnum]
:linenumber
```

i.e. a symbol name (e.g. *main*), a symbol name plus a hex offset (e.g. *afunc+1a*), a symbol minus a hex offset, a hex number preceded by a dollar sign, or a line number preceded by a colon.

Symbol references and line numbers always resolve to full 32 bit addresses (segment:offset). Non symbolic addresses may be entered by specifying either a segment:offset pair or just the offset. If only an offset is given, the segment value defaults to the last segment accessed, unless the offset specified is one of the pointer registers **ip**, **sp** or **bp**, in which case the segment value is taken from the appropriate segment register (**cs** for **ip**, **ss** for **sp** and **bp**). In all cases except line numbers, a constant may be added or subtracted from the offset part of the address. Note that address arithmetic never changes the segment value, wrapping around within the segment if overflow or underflow occurs.


```
More examples:

bp-4              (address SS:BP - 4)
ds:dx     (address DS:DX)
0f121     (address 0f121 in default segment)
ds:200    (address DS:0200)
```

In the **b** (breakpoint) command any decimal number will be interpreted as a line number by default, while in the **u** (unassemble) command any number will be intepreted as a hex number representing an address by default. These assumptions can always be overridden by using the colon or dollar prefixes.

When entering a symbol, it is not necessary to type the underscore prepended by the C compiler, however when printing out symbols the debugger will always print the underscore. Any register name may also be used where a symbol is expected.

6.3 Commands

6.3.1 The A Command: set command line arguments

The **a** command is used to set the command line arguments. When the user program is run, *argc* is set to the number of arguments typed, *argv[1]* is set to the first argument typed, *argv[1]* to the second, and so on. *argv[0]* is always a *NULL* pointer.

6.3.2 The B Command: set or display breakpoints

The **b** command is used to set and display breakpoints. If no expression is supplied after the **b** command, a list of all currently set breakpoints will be displayed. If an expression is supplied, a breakpoint will be set at the line or address specified. If you attempt to set a breakpoint which already exists, or enter an expression which Lucifer cannot understand, an appropriate error message will be displayed. Note: any decimal number specified will be interpreted as a line number by default, if you want to specify an absolute address, prefix it with a dollar sign or use a symbol name.

```
: b 10
Set breakpoint at _main+27
:

Example: (setting a breakpoint at line
          10 of a C program)
```

6.3.3 The D Command: display memory contents

The **d** command is used to display a hex dump of the contents of memory on the target system. If no expressions are specified, one byte is dumped from the address reached by the last **d** command. If one address is specified, 16 bytes are dumped from the address given. If two addresses are specified, the contents of memory from the first address to the second address are displayed. Dump addresses given can be symbols, line numbers, register names or absolute memory addresses.

6.3.4 The E Command: examine C source code

The **e** command is used to examine the C source code of a function or file. If a function name is given, Lucifer will locate the source file containing the function requested and display from just above the start of the function. If a file name is given, Lucifer will display from line 1 of the requested file.

```
: e main
2:
3:int    value, result;
4:
5:main()
6:{
7:       scanf("%d",&value);
8:       result = (value << 1) + 6;
9:       printf("result = %d\\n",result);
10:}
:
```

Example: (use of **e command to list C** *main() function)*

6.3.5 The G Command: commence execution

The **g** command is used to commence execution of code on the target system. If no expression is supplied after the **g** command, execution will commence from the current value of PC (the program counter). If an expression is supplied, execution will commence from the address given. Execution will continue until a breakpoint is reached, or the user interrupts with control-C. After a breakpoint has been reached, execution can be continued from the same place using the **g**, **s** and **t** commands.

6.3.6 The I Command: toggle instruction trace mode

The **i** command is used to toggle instruction trace mode. If instruction trace mode is enabled, each instruction is displayed before it is executed while stepping by entire C lines with the **s** command.

```
Assume PC points to 9:
printf("result = %d\\n",result);

With instruction trace turned OFF,
stepping would produce:

: s
result = 20
Stepped to
10:}
:

With instruction trace turned ON,
stepping would produce:

: s
_main+4D PUSH     _result
_main+51 MOV      DX,#0083
_main+54 PUSH     DX
_main+55 CALL     _printf
result = 20
_main+58 ADD      SP,#04,WORD

Stepped to
10:}
```

```
:

Note: the library function printf() was
not traced by the stepping mechanism
and thus functioned correctly.

Example: (showing effect of i command on
           step display)
```

6.3.7 The Q Command: exit to DOS

The **q** command is used to exit from Lucifer to DOS.

6.3.8 The R Command: remove breakpoints

The **r** command is used to remove breakpoints which have been set with the **b** command. For each breakpoint, the user is prompted as to whether the breakpoint should be removed or not.

```
: r
Remove _main+27 “tst.c” line 10 ? y
Remove _main+44 “tst.c” line 13 ? n
Remove _test “tst.c” line 22 ? n
:

Example: (removes breakpoint at main+27)
```


6.3.9 The S Command: step one C line

The **s** command is used to step by one line of C source code. This is normally implemented by executing several machine instruction single steps, and therefore can be quite slow. If Lucifer can determine that there are no function calls or control structures (*break, continue, etc.*) in the line, it will set a temporary breakpoint on the next line and execute the line at full speed.

When single stepping by machine instructions, the step command will execute subroutine calls to external and library functions at full speed, thereby avoiding the slow process of single stepping through complex library routines such as *printf()* or *scanf()*. Normal library console I/O works correctly during single stepping.

```
Assume current PC points to line 6:
{

: s
```

```
Stepped to
7:       scanf("%d",&value);
: s
Target wants input: 7
Stepped to
8:       result = (value << 1) + 6;
: s
Stepped to
9:       printf("result = %d\\n",result);
: s
result = 20
Stepped to
10:}
:

Example: (stepping through several
           lines of C code)
```


6.3.10 The T Command: trace one instruction

The **t** command is used to trace one machine instruction on the target. The current value of PC (the program counter) is used as the address of the instruction to be executed. After the instruction has been executed, the **next** instruction and the contents of all registers will be displayed.

6.3.11 The U Command: disassemble

The **u** command disassembles object code from the target system's memory. If an expression is supplied, the disassembly commences from the address supplied. If an address is not supplied, the disassembly commences from the instruction where the last disassembly ended. The disassembler automatically converts addresses in the object code to symbols if the symbol table for the program being disassembled is available. If the source code for a C program being disassembled is available, the C lines corresponding to each group of instructions are also displayed. Note: any values specified will be interpreted as absolute addresses by default, if you want to specify a line number, prefix it with a colon.

```
: u :10
10:      answer += (value + 7) << 1;
_main+1B MOV      DX,_value
_main+1F ADD      DX,#07,WORD
_main+22 SHL      DX,#1
_main+24 ADD      _answer,DX
11:      printf("answer = %d\\n", answer);
```

```
_main+28 PUSH     _answer
_main+2C MOV      DX,#0074
_main+2F PUSH     DX
_main+30 CALL     _printf
_main+33 ADD      SP,#04,WORD
12:      exit(answer << 1);
_main+36 MOV      DX,_answer
_main+3A SHL      DX,#1
_main+3C CALL     _exit
:

Example: (disassembly from line 10 of
           a C program)
```

6.3.12 The X Command: examine or change registers

The **x** command is used to examine and change the contents of the target CPU registers. If no parameters are given, the registers are displayed without change. To change the contents of a register, two parameters must be supplied, a valid register name and the new value of the register. After setting a new register value, the contents of the registers are displayed.

Any valid 8086 register name may be used. **pc** and **ip** are both accepted for the program counter. **psw** and **flags** are both accepted for the flags register. The low and high bytes of the general purpose registers may be accessed with **al**, **ah**, **bl**, **bh**, etc.

6.3.13 The @ Command: display memory as a C type

The **@** command is used to examine the contents of memory interpreted as one of the standard C types. The form of the **@** command is:

where **t** is the type of the variable to be displayed, **m** consists of zero or more modifiers which effect the way the type is displayed, **i** consists of zero or more indirection operators '*', and expr is the address of the variable to be displayed.

't'	Type	Modifiers
c	char	u = unsigned, x = hex, o = octal
d	double	none
f	float	none
fp	far pointer	none
i	int	u = unsigned, x = hex, o = octal

l	long	u = unsigned, x = hex, o = octal
p	pointer	f = far pointer
s	string	none

```
Examples:

To display a long variable “longvar” in hex:

@lx longvar

To display a character, pointed at by a far pointer “cptr”:

@c f*cptr

To de-reference “ihandle”: a pointer to a pointer to an unsigned
int:

@iu **ihandle
```


After displaying the variable, the current address is advanced by the size of the type displayed, making it possible to step through arrays by repeatedly pressing return. On-line help for the **@** command may be obtained by entering **?@** at the ‘:’ prompt.

6.3.14 The ^ Command: print a stack trace

The ^ command is used to display a trace of the currently active functions by back-tracking frame pointers and return addresses up the user program’s stack.

The name, frame pointer, and caller of each currently active function will be displayed.

```
Assume execution was stopped by a breakpoint
in f2()

: ^
BP = FF6E         f2()    called by “tst.c”, line 24
BP = FF76         f1()    called by “tst.c”, line 45
BP = FF82         main()  called by start+CC
BP = 0000         start()
:
```

The stack trace displayed above means that at line 45 in "tst.c", *main()* called *f1()*, which in turn called *f2()*. *start()* is the C runtime startoff module.

6.3.15 The $ Command: reset to initial configuration

The **$** command is used to reset Lucifer to its' initial configuration, ready to execute the user program from the start. Note: it is not always safe to use this command, especially with programs which use dynamically allocated memory. If in doubt, exit and re-load Lucifer from DOS.

6.3.16 The . Command: set a breakpoint and go

The **.** command is used to set a temporary breakpoint and resume execution from the current value of PC (the program counter). Execution continues until **any** breakpoint is reached, or the user interrupts with control-C, then the temporary breakpoint is removed. Note: the temporary breakpoint is removed even if execution stops at a different breakpoint or is interrupted. If no breakpoint address is specified, the . command will display a list of active breakpoints.

6.3.17 The ; Command: display from a source line

The **;** command is used to display 10 lines of source code from a specified position in a source file. If the line number is omitted, the last page of source code displayed will be re-displayed. Example:


```
: ; 4
4:
5: main()
6: {
7:     scanf("%d",&value);
8:     result = (value << 1) + 6;
9:     printf("result = %d\\n",result);
10:}
```

6.3.18 The = Command: display next page of source

The = Command is used to display the next 10 lines of source code from the current file. For example, if the last source line displayed was line 7, = will display 10 lines starting from line 8.

6.3.19 The - Command: display previous page of source

The **-** Command is used to display the previous 10 lines of source code from the current file. For example, if the last page displayed started at line 15, **-** will display 10 lines starting from line 5.

6.3.20 The / Command: search source file for a string

The / command is used to search the current source file for occurences of a sequence of characters. Any text typed after the / is used to search the source file. The first source line containing the string specified is displayed. If no text is typed after the /, the previous search string will be used. Each string search starts from the point where the previous one finished, allowing the user to step through a source file finding all occurences of a string.

```
: /printf
10:     printf("Enter a number:");
: /
14:     printf("Result = %d\\n",answer);
: /
Can't find printf
:

Example: (use of / command to find all
          uses of printf() )
```


6.3.21 The ! Command: execute a DOS command

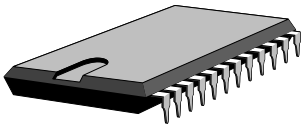
The **!** command is used to execute an operating system shell command line without exiting from Lucifer. Any text typed after the **!** is passed through to the shell without modification.

6.3.22 Other Commands

In addition to the commands listed above, Lucifer will interpret any valid decimal number typed as a source line number and attempt to display the C source code for that line.

Pressing return without entering a command will result in re-execution of the previous command if the previous command was **@**, **d**,**e**, **s**, **t**, **u**. In all cases the command resumes where the previous one left off. For example, if the previous command was **d 2000**, pressing return will have the same effect as the command **d 2010**.

ROM Development with Pacific C



7.1 Requirements for ROM Development

ROM development differs from generating executable files for MS-DOS, as discussed in the rest of this manual. To produce embedded code in ROM from Pacific C, you will need to have purchased either the Pacific C ROMDEV package or addon, or have obtained the Pacific C Special Edition - the Special Edtion is available only through educational institutions. If you have only the standard Pacific C package, you can obtain the ROMDEV addon from HI-TECH Software or your reseller.

Once you have the appropriate software, you will need two pieces of hardware, apart from your PC. You will need a target system for development. This will usually consist of an evaluation board, or some other stand-alone circuit board with an 8086 family processor (e.g. an 80C188EB) and at least a ROM socket. The board will normally also have a RAM socket, and in order to use the remote debugger you will also need a serial port that can be connected to a serial port on your PC. The 80C188EB has two on-board serial ports which are suitable for this purpose.

The remainder of this chapter will concentrate mainly on using the 80C188EB, as it is a very popular embedded variant of the 8086 family, however the techniques discussed are equally applicable to other processors, but may require additional work on your part to customize run-time startups and other components of the run-time environment. The exact nature of these changes will depend on the particular hardware.

The other piece of equipment you will need is an EPROM programmer (or EPROM emulator). This is used to transfer your compiled program into non-volatile memory for execution in your target system. Most EPROM programmers include a software program that runs under DOS and will read a HEX or binary file and program it into the EPROM. The EPROM is then plugged into your target system where it will be executed.

7.2 ROM Code vs. DOS .EXE Files

When generating DOS programs, the Pacific C compiler will take your source code, in one or more C files, and translate it to 8086 machine code. This is then linked with an appropriate set of library routines, and converted into a DOS .EXE (*dot-eksi*) file that can be run under DOS. The libraries used for DOS programs make calls to DOS system functions (e.g. via int #21h) to perform all I/O and many other functions.

In an embedded system where the code is stored in a ROM (*Read Only Memory*) DOS is not available, and the int #21 calls that the DOS library routines would make are not available. There are two implications of this;

- No file I/O is available, due to lack of an operating system and filesystem;
- A .EXE file will not run, due to the DOS calls it makes

In order to generate programs that will run in a target system where the only resources available are the bare hardware, the Pacific C ROMDEV system includes an alternate set of library routines and run-time startup routines that provide a full set of non-I/O routines and a limited set of character I/O routines, relying only on available hardware. The basic I/O routines like `printf()`are supported by small routines that use a serial port or other hardware to put and get characters.

The linker also produces output files in different formats; rather than a .EXE file an embedded program will usually be presented in its final (executable) form as a HEX or raw binary file. A binary file consists only of an image of the bytes to be programmed into the EPROM; a HEX file can take a variety of formats, but contains both the program bytes, plus addressing information. In either case the program is *absolute*, i.e. it has been linked for and can run only at a specific physical address, unlike a .EXE program which has a header containing information on addresses that need fixing-up after loading, enabling it to be loaded and run at any address.

7.3 What You Need to Know

In order to develop an embedded program, you need some information that you would normally not need to generate a DOS program. You will need to know:

- The memory organization of your target system;
- The initialization sequence required for your target system;
- I/O port definitions and functions
- The steps required to program an EPROM for your target, or some other means of getting the compiled code into your target.

The *memory organization* of your system means the starting address and size of your RAM and ROM. The ROM is used to hold the program, and the RAM is used for run-time storage of variables and stack. A typical 8086 based system will have ROM at high memory addresses, and RAM at low memory addresses, starting usually from zero. This is because the 8086 begins program execution after reset at a high memory address, usually FFFF0 hex - i.e. 16 bytes below the top of the 1Mbyte addressable memory range. Low memory holds interrupt vectors, which are best placed in RAM as they may need to be changed at run-time. The actual reset address is another piece of information you will need, but for any of the 8086, 8088, 80186 and 80188 processors this will be FFFF0.

The *initialisation sequence* is a very important piece, or set, of information that you will require. Some target systems will have a trivial initialization sequence - for example an 8088 with hardwired address decoders will reset in a state where it can run code immediately. However other systems will require a precise sequence of operations to be performed, often in a minimum number of bytes or cycles, before the configuration is set up to access available memory properly. The arcane nature of these initialization sequences and the requirement that they be executed in a precise manner, often leads to them being referred to as *incantations*. Get the spell wrong and your target system will remain an inanimate lump of plastic and silicon.

Fortunately, if you are using an 80C188EB in the very common configuration of a single block of ROM enabled by the Upper Chip Select, and a single block of RAM enabled by the Lower Chip Select, Pacific C provides a canned solution to the initialization problem. By selecting the appropriate options in the **Memory Model and Chip Type** dialog box in PPD, a standard initialization routine will be linked in to your program which will set the chip select registers to the right values. If you are using another setup, you will need to determine the initialization sequence yourself. Use the 80C188EB code, in the file `pwrupeb.as`, as a starting point, and modify it accordingly.

The I/O ports available in your particular target system will vary widely, depending on which chip you are using and what additional hardware your board has. The 80C188EB has a standard on-board set of I/O ports, including parallel I/O, serial I/O and timers, but other processors, such as the 8088, have no on-board I/O and so you will need to know what peripheral chips are used and what I/O ports they implement.

In addition to knowing what the I/O ports are, and what addresses they are mapped to, you will need to have both knowledge and understanding of how to program the I/O ports to achieve the functions you want with your embedded system. This kind of information is beyond the scope of this manual and depends on both experience and close study of manufacturer’s data books, as well as the design of your system.

Once the program is compiled, you will have to transfer the disk file representing your program into an EPROM, or download it into an EPROM emulator, or use a remote debugger if you already have this set up. If you’re lucky, you will have purchased a board that comes supplied with a target ROM implementing our remote debugger, LUCIFER. In this case you can download directly from PPD. Otherwise you will have to follow whatever steps are required for your particular setup. This also is beyond the scope of this manual.

7.4 First Steps

Before attempting to generate code for an embedded system, it will be useful for you to familiarize yourself with the Pacific C compiler in general, and PPD in particular. Compile and run some of the programs in the EXAMPLES directory, then create one or two of your own programs and compile and run them under DOS. This will ensure that you are comfortable using PPD before following the procedures below.

When compiling a first C program, it has been traditional to use a “Hello, world” program that does nothing but print a string. In an embedded system this seemingly simple task is quite an achievement - it requires not only that a program should start correctly, but that the serial port driver is correct, right down to the correct baud rate setting for the system crystal frequency. Then you must have the serial port wired correctly and connected to a terminal or terminal program. All of this is difficult to guarantee in one step.

Accordingly, we present here some rather more basic code suitable for a first embedded program. This piece of code simply turns an I/O port pin on and off, so that if a LED is connected to the I/O port pin, or even a CRO, it will produce a visible signal that the program is running correctly.

```
#include <80c188eb.h>

/*
 *       Demo program for the 80C188EB. This program
 *       flashes a LED attached to bit 3 of port 1.
 */

main()
{
        int     i;

        _P1CON &= ~0x8;      /* set bit 3 to output */
        for(;;) {
                for(i = 0 ; i != 0x7fff ; i++)
                        continue;
                _P1LTCH ^= 0x8;   /* toggle the bit */
        }
}
```

Figure 7-1 - The Hello Led Program

This particular program is written for an 80C188EB with a LED connected to bit 3 of port 1. It programs the port pin to be an output, and toggles it repeatedly with a delay. Like all good embedded programs, it never exits, but sits in an infinite loop. If you are using a JED PC-540 board, there is conveniently a LED driver connected to that pin, and brought out to the I/O connector. See the JED documentation for more details.

If you are using different hardware, you will need to modify the program accordingly. The source code is shown in figure -. This is also in the file `LED.C` in the `LUCIFER` directory, within the Pacific C installation directory.

7.4.1 Entering the Program

To enter this program, change into the LUCIFER directory, and start PPD with the argument LED - simply type:

```
PPD LED<Enter>
```

Unless you have previously created a LED.PRJ file, PPD will start up and load the file LED.C, but will not load a project file. If you need to change the LED.C file, make the changes now. Then select **New Project** from the **Make** menu. PPD will prompt you for a project file name, with the default being LED.PRJ. Press `<Enter>` to accept this default. PPD will now prompt you with a series of dialogs to allow you to specify necessary information. The first dialog, shown in figure -, allows selection of ROM code rather than DOS code. Select **ROM code** with the mouse, or press **M** and then press `Enter` or click the **OK** button.

Figure 7-2 - Select DOS or ROM code

7.4.2 Selecting Processor Type

The next dialog presented will allow you to select the memory model, which for now we will leave at the default of Small, and the processor type. The choices are **8086 or 8088** (the default) and **80186, 80188 or 80286**. Select the appropriate choice for your particular hardware. If you select **80188** code you will now see become visible another check box, marked **80C188EB specific powerup**. Checking this option will enable standard powerup initialisation code for the 80C188EB to be linked in, as discussed earlier.

If you are not using an 80C188EB and you need a powerup routine to initialise the processor, see the discussion below on user-defined powerup sequences. Figure - shows the dialog after selecting 80188 code with the EB powerup. Now press Enter to step to the next dialog.

Figure 7-3- Select Processor Type

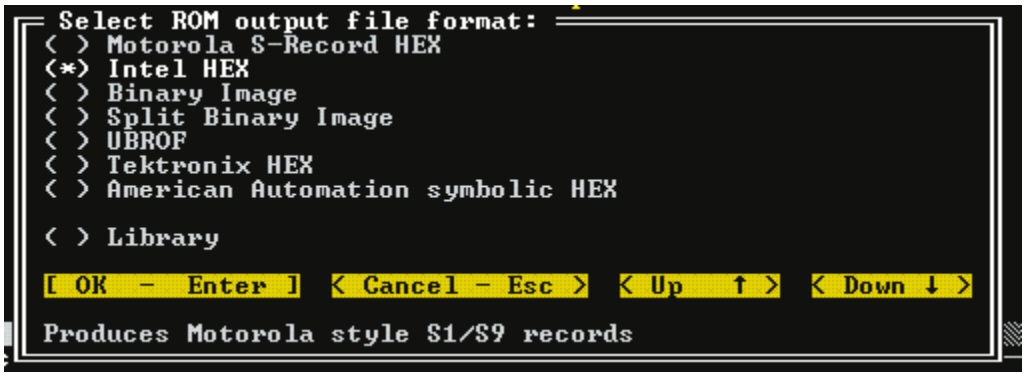


Figure 7-4- Selecting the ROM file type

7.4.3 Choosing the File Type

The dialog shown in figure - allows you to choose the type of output file you would like. This will be determined by the requirements of you EPROM programmer or other hardware that you will be using to get your code into your target system. If you will be using the Lucifer debugger, you should leave the default of Intel HEX. After choosing the type, press Enter to accept the selection.

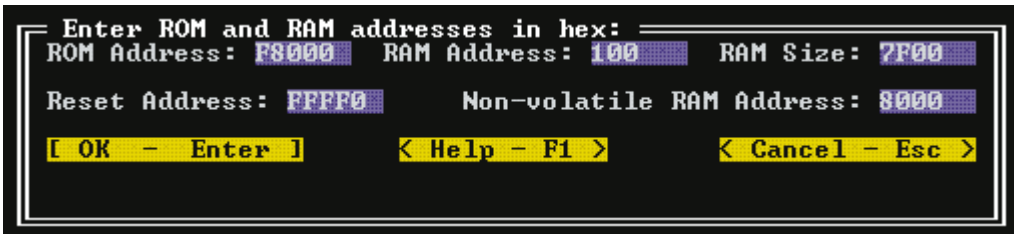


Figure 7-5- Memory Addresses

7.4.4 Setting Memory Addresses

As discussed previously, you need to know the address and size of the ROM and RAM available in your system. The default values shown in figure - represent a 32Kbyte ROM (e.g. a 27256) located at the top of the memory space, and a 32K RAM (e.g. a 62256) located at zero.Note that it is not necessary to exactly match the actual sizes for the program to work; for example a 64K byte ROM would still work with this setup, but only the top half would be accessible, and when programming it the code would have to be programmed at offset 8000h. Note also that the addresses are given in bytes, not paragraphs, but in general will always be multiples of 16 since all memory addressing in the 8086 family is done in increments of 16 bytes. Your EPROM programmer may require you to enter a segment address when you load the hex file - in this case the segment address would be F800 since the physical address of the start of the ROM is F8000.

The RAM address is given as 100h, even though it probably actually starts at 0. This is to leave the interrupt vectors in the bottom 256 bytes of memory alone. The RAM size then becomes 7F00, since this represents the number of bytes available from the starting address given. The RAM starting address will be used to set the location for storage of variables; the RAM size plus RAM address will give the initial stack pointer, i.e. at the top of the available RAM.

The non-volatile RAM address is unimportant for this program since we will not be using any *persistent* variables. If you do use non-volatile RAM, you should enter here the address of the battery backed RAM. It must not overlap the other RAM area, so that if all RAM is battery backed, you might choose to set non-volatile RAM at 100, and the normal RAM at 200 with a size of 7E00.

If you are going to execute the code under Lucifer or another debugger, the memory addresses you set will be different. In this case you will set both the ROM and RAM addresses to values within the RAM in your target system that is available for running programs under Lucifer. For this example proram it should be adequate to set RAM address to 400h (above the interrupt vectors), RAM size to 400h, and ROM address to 800h (equal to the sum of the RAM address and RAM size). If Lucifer is pre-implemented on your target system, it should have information about what memory is available, or if you have implemented it yourself you will know what addresses you can use. More information about implementing Lucifer follows later in this chapter.

After completing the entry of the memory addresses, press Enter to proceed to the **Optimization** dialog.

7.4.5 Selecting Optimizations

When compiling it is possible to select from a set of available optimizations. These are discussed earlier in this manual, and for ROM code are no different to those used with DOS code. For this demonstration you may select all or none of the available choices, or any combination of them.

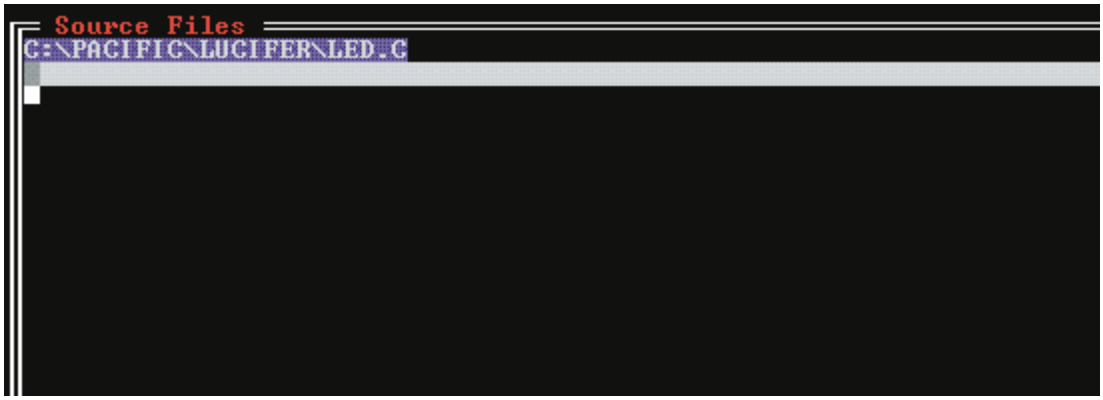


Figure 7-6- Source File List

7.4.6 Entering Source Files

After the **Optimisation** dialog, you will be presented with the **Source File List**. This allows you to enter the names of the source files you wish to include in this project. Since the list is initially empty, the name of the currently loaded editor file will be included automatically. For this demonstration this is all you need, but if you wish to enter further files, just press Enter to go to a new line, and type the name of another source file. The file does not have to exist at this point in time. Enter each file on a separate line, and include the **.C** filetype (or **.AS** for an assembler source file).

Now press Escape or click in the menu bar, or click the **Done** button and you will return to the normal mode of operation.

7.4.7 Making the Program

Press the F5 key, or select the **Make** menu then the **Make** menu entry. PPD will evaluate the dependencies of the source file list, i.e. it will compare the modification times of the source files with the modification times of the corresponding object files, and determine if the source files need recompiling. In this instance there is only one source file, and it has not yet been compiled, so the object file will not exist. If the object file did exist and was newer, PPD would then check any #include files used by the source file. If any of these was newer, the source file would be recompiled anyway. Similarly, after checking the source files, the object files and libraries will be checked against the HEX or binary file, to determine if a re-link is necessary. Again, in this instance, the fact that no object file exists will guarantee a re-link.

During the compilation and linking phases PPD will display completion bars, as you will have seen during DOS compilations. At the end of the link phase, a completion message will be presented as shown in figure -.

.You will notice that there are two CODE areas reported - one is the main body of the program, the other is the power-on reset vector area at FFFF0. The **Utility/Memory usage map** menu entry will give a slightly more comprehensive report of the same information. The addresses of these code areas should agree with the values you gave in the **ROM and RAM addresses** dialog.

Figure 7-7- Compilation Successful Report

7.4.8 Running the Code

Now that you have a HEX file or binary file, you need to get it into your target system. Run your EPROM programmer or other utility to place the compiled program into your EPROM. You can use the user-defined commands in the **Utility** menu as a convenient way of invoking an EPROM programmer. After the EPROM is programmed, plug it into your system, and power it up. The led should flash!

If you have a Lucifer target ROM in your system, you should have connected the serial port on your target system to either COM1 or COM2 on your PC. You will need to know what baud rate the Lucifer target is set up to use (some implemenations will auto-detect the baud rate). Select the **Run** menu, then the **Download LED.HEX** menu item. The act of making the program will have enabled this menu selection

You will then be presented with a dialog box asking you to choose a COM port and baud rate. Make the appropriate selections, the press Enter. The COM port and baud rate will be saved in the project file.

Lucifer will then be invoked, and it will attempt communication with the target system. If your target system is working, and you have correctly connected the communications cable, and you have got the baud rate correct, it will announce itself then download the program. Type **g** then **Enter** to start it running. The LED should flash!

7.5 Going Further

Congratulations! You have compiled and run a program in an embedded system. Having achieved this you have taken the most difficult step. Let's move on now to a more complex program. Run PPD and open the file **luctest.c**. Now select **Make/New project** and set memory addresses etc. just as before. At the source file list, you will have **luctest.c** entered automatically. Press **Enter** to step to a new line and type **serial.c**, then **ESC** to exit the list This program is a test for the serial port. The module **serial.c** contains a simple driver for the 80C188EB serial port, which is called by the code in **luctest.c**. If you need a custom powerup routine, add it to the source file list also. If your 80C188EB is not using a 40MHz crystal, edit **serial.c** appropriately. If you are not using an 80C188EB, you will need to modify **serial.c** to driver whatever serial port you do have.

Now make the program with F5 or **Make/Make**. Program into an EPROM and run it, with a dumb terminal or terminal program on your PC connected to the serial port You should see a pattern of characters displayed, followed by a prompt. Type one line of text at the prompt and press Enter, and the characters will be echoed back as hex bytes.

If this is successful, you have a working serial port, and can move onto implementing Lucifer in your board, to enable you to download and debug programs without burning EPROMS.

7.6 Implementing Lucifer

Included in the LUCIFER directory are two target monitors for Lucifer; one is for the 80C188EB, and the other is for a generic 8086 type processor with a Zilog SCC serial port. If your target board does not use either of these configurations you will need to generate your own target program, using one of these as a starting point. If using the 8086/SCC target program (TARGET86.C) you will probably still need to edit it to set port addresses, baud rate etc.

The 80C188EB target program is TARGETEB.C. There is a project file for this, TARGETEB.PRJ, so you can simply run type 'PPD TARGETEB' and it will load the project file - or use the ALT-P command from within PPD to load a new project. You may need to edit TARGETEB.C to set the baud rate - it includes a #error directive to force you to edit it, and has comments to indicate what needs changing.

Then make the HEX or binary file, and program it into an EPROM. Plug this into your target board, and use the same serial connection you used for testing LUCTEST. From within PPD, select the **Run/Debug using...** menu item - it's not important at this stage what symbol file you have. Set the baud rate when requested to 38400, then you should get the Lucifer sign on and a message "Target identifies as...". This means that the Lucifer debugger on your PC is communicating with the target system.

If it doesn't work, it is likely to be due to incorrect serial connections or programming, incorrect memory addresses, or EPROM programming errors (including programming the code into the wrong part of the EPROM).

To run Lucifer from the command line, you can invoke it as

```
LUC86 -S38400 -PCOM1
```

Subsititute the appropriate baud rate and com port if required. The defaults for baud rate and com port are 19200 and COM1. The 188EB target is set up for 38400 baud, which gives faster downloading, but if you invoke Lucifer without a -S38400, it will not talk to the target board.

7.7 Debugging with Lucifer

There is a separate chapter that details the commands to Lucifer, and these will not be repeated here. On-line help is also available from within Lucifer. When compiling a user program for use with Lucifer, you should choose ROM and RAM addresses that lie within the RAM available in your system. For example, if you had 64K bytes of RAM from location zero, Lucifer would normally use the top 100h bytes, and addresses below 400h should be reserved for interrupts, so addresses from 400h to FF00 are available. A good strategy is to set the RAM address to 400h, and RAM size to a value you know is big enough for your program's data and stack (400h as a size is good for most small programs). Then set the ROM address to the sum of the RAM size and address, e.g. 800h. The code will extend up from there towards the top of RAM. Check a link map or the psect map provided after compilation to ensure that the code does not overlap the Lucifer RAM area at the top.

After debugging your program, you need only relink it with different addresses to put it into ROM. Have fun!

Chapter 7 - ROM Development with Pacific C

7

Linker Reference Manual

8.1 Introduction

Pacific C incorporates a relocating assembler and linker to permit separate compilation of C source files. This means that a program may be divided into several source files, each of which may be kept to a manageable size for ease of editing and compilation, then each object file compiled separately and finally all the object files linked together into a single executable program.

The assembler is described in its own chapter. This chapter describes the theory behind and the usage of the linker.Note however that in most instances it will not be necessary to use the linker directly, as the compiler drivers (PPD or command line) will automatically invoke the linker wil all necessary arguments. Using the linker directly is not simple, and should be attempted only by those with a sound knowledge of the compiler and linking in general.

If it is absolutely necessary to use the linker directly, the best way to start is to copy the linker arguments constructed by the compiler driver, and modify them as appropriate. This will ensure that the necessary startup module and arguments are present.

Note also that the linker supplied with Pacific C is generic to a wide variety of compilers for several different processors. Not all features described in this chapter are applicatble toall compilers.

8.2 Relocation and Psects

The fundamental task of the linker is to combine several relocatable object files into one. The object files are said to be *relocatable* since the files have sufficient information in them so that any references to program or data addresses (e.g. the address of a function) within the file may be adjusted according to where the file is ultimately located in memory after the linkage process. Thus the file is said to be relocatable. Relocation may take two basic forms; relocation by name, i.e. relocation by the ultimate value of a global symbol, or relocation by psect, i.e. relocation by the base address of a particular section of code, for example the section of code containing the actual excutable instructions.

8.3 Program Sections

Any object file may contain bytes to be stored in memory in one or more program sections, which will be referred to as *psects*. These psects represent logical groupings of certain types of code bytes in the program. In general the compiler will produce code in three basic types of psects, although there will be several different types of each. The three basic kinds are *text* psects, containing executable code, *data* psects, containing initialized data, and *bss* psects, containing uninitialized but reserved data.

The difference between the data and bss psects may be illustrated by considering two external variables; one is initialized to the value 1, and the other is not initialized. The first will be placed into the data psect, and the second in the bss psect. The bss psect is always cleared to zeros on startup of the program, thus the second variable will be initialized at run time to zero. The first will however occupy space in the program file, and will maintain its initialized value of 1 at startup. It is quite possible to modify the value of a variable in the data psect during execution, however it is better practice not to do so, since this leads to more consistent use of variables, and allows for restartable and romable programs.

For more information on the particular psects used in a specific compiler, refer to the appropriate machine-specific chapter.

8.4 Local Psects

Most psects are *global*, i.e. they are referred to by the same name in all modules, and any reference in any module to a global psect will refer to the same psect as any other reference. Some psects are *local*, which means that they are local to only one module, and will be considered as separate from any other psect even of the same name in another module. Local psects can only be referred to at link time by a class name, which is a name associated with one or more psects via a *CLASS=* directive in assembler code.

8.5 Global Symbols

The linker handles only symbols which have been declared as global to the assembler. From the C source level, this means all names which have storage class external and which are not declared as static. These symbols may be referred to by modules other than the one in which they are defined. It is the linker's job to match up the definition of a global symbol with the references to it. Other symbols (local symbols) are passed through the linker to the symbol file, but are not otherwise processed by the linker.

8.6 Link and load addresses

The linker deals with two kinds of addresses; *link* and *load* addresses. Generally speaking the link address of a psect is the address by which it will be accessed at run time. The load address, which may or may not be the same as the link address, is the address at which the psect will start within the output file (hex

or binary file etc.). In the case of the 8086 processor, the link address roughly corresponds to the offset within a segment, while the load address corresponds to the physical address of a segment. The segment address is the load address divided by 16.

Other examples of link and load addresses being different are; an initalized data psect that is copied from ROM to RAM at startup, so that it may be modified at run time; a banked text psect that is mapped from a physical (== load) address to a virtual (== link) address at run time.

The exact manner in which link and load addresses are used depends very much on the particular compiler and memory model being used.

8.7 Operation

A command to the linker takes the following form:

```
HLINK[1] options files ...
```

Options is zero or more linker options, each of which modifies the behaviour of the linker in some way. Files is one or more object files, and zero or more library names. The options recognized by the linker are listed in table 8-1 and discussed in the following paragraphs.

8.7.1 Numbers in linker options

Several linker options require memory addresses or sizes to be specified. The syntax for all these is similar. By default, the number will be interpreted as a decimal value. To force interpretation as a hex number, a trailing 'H' should be added, e.g. 765FH will be treated as a hex number.

8.7.2 -8

When linking 8086 programs, it is convenient to have addresses in the link map appear as segmented addresses, e.g. 007F:1234 represents a segment number of 7F and an offset of 1234.

8.7.3 -A*class=low-high,...*

Normally psects are linked according to the information given to a -P option (see below) but sometimes it is desired to have a psect or class linked into more than one non-contiguous address range. This option allows a number of address ranges to be specified for a class. For example:

```
-ACODE=1020h-7FFEh,8000h-BFFEh
```

1 In earlier versions of Pacific C the linker was called LINK.EXE

Table 8-1; Linker options

Option	Effect
-8	Use 8086 style segment:offset address form
-A*psect*=*low*-*high*,...	Specify address ranges for a psect
-C*psect=class*	Specify a class name for a global psect
-C*baseaddr*	Produce binary output file based at *baseaddr*
-D*symfile*	Produce old-style symbol file
-F	Produce .OBJ file with only symbol records
-G*spec*	Specify calculation for segment selectors
-H*symfile*	Generate symbol file
-I	Ignore undefined symbols
-L	Preserve relocation items in .OBJ file
-LM	Preserve segment relocation items in .OBJ file
-N	Sort symbol table in map file by address order
-M*mapfile*	Generate a link map in the named file
-O*outfile*	Specify name of output file
-P*spec*	Specify psect addresses and ordering
-S*psect=max*	Specify maximum address for a psect
-U*symbol*	Pre-enter symbol in table as undefined
-V*avmap*	Use file *avmap* to generate an Avocet format symbol file
-W*warnlev*	Set warning level (-10 to 10)
-W*width*	Set map file width (>10)
-X	Remove any local symbols from the symbol file
-Z	Remove trivial local symbols from symbol file

specifies that the class **CODE** is to be linked into the given address ranges. Note that a contribution to a psect from one module cannot be split, but the linker will attempt to pack each block from each module into the address ranges, starting with the first specified.

8.7.4 -C*psect=class*

This option will allow a psect to be associated with a specific class. Normally this is not required on the command line since classes are specified in object files.

8.7.5 -D*symfile*

Use this option to produce an old-style symbol file. An old-style symbol file is an ASCII file, where each line has the link address of the the symbol followed by the symbol name.

8.7.6 -F

Normally the linker will produce an object file that contains both program code and data bytes, and symbol information. Sometimes it is desired to produce a symbol-only object file that can be used again in a subsequent linker run to supply symbol values. The -F option will suppress data and code bytes from the output file, leaving only the symbol records.

8.7.7 -G*spec*

When linking programs using segmented, or bankswitched psects, there are two ways the linker can assign segment addresses, or selectors, to each segment. A segment is defined as a contiguous group of psects where each psect in sequence has both its link and load address concatenated with the previous psect in the group. The segment address or selector for the segment is the value derived when a segment type relocation is processed by the linker (see the description of the object code format in appendix).

By default the segment selector will be generated by dividing the base load address of the segment by the relocation quantum of the segment, which is based on the *RELOC*= value given to psects at the assembler level. This is appropriate for 8086 real mode code, but not for protected mode or some bankswitched arrangements. In this instance the **-G** option is used to specify a method for calculating the segment selector. The argument to -G is a string similar to:

```
A/10h-4h
```

where *A* represents the load address of the segment and represents division. This means “Take the load address of the psect, divide by 10 hex, then subtract 4". This form can be modified by substituting *N* for *A*, *X* for / (to represent multiplication), and adding rather than subtracting a constant. The token *N* is replaced by the ordinal number of the segment, which is allocated by the linker. For example:

```
N*8+4
```

means “take the segment number, multiply by 8 then add 4". The result is the segment selector. This particular example would allocate segment selectors in the sequence 4, 12, 20, ... for the number of segments defined. This would be appropriate when compiling for 80286 protected mode, where these selectors would represent LDT entries.

8.7.8 -H*symfile*

This option will instruct the linker to generate a symbol file (in “new” HI-TECH format - see appendix “File Formats”). The optional argument *symfile* specifies a file to receive the symbol file. The default file name is **l.sym**.

8.7.9 -I

Usually failure to resolve a reference to an undefined symbol is a fatal error. Use of this option will cause undefined symbols to be treated as warnings instead.

8.7.10 -L

When the linker produces an output file it does not usually preserve any relocation information, since the file is now absolute. In some circumstances a further "relocation" of the program will be done at load time, e.g. when runnig a .EXE file under DOS or a .PRG file under TOS. This requires that some information about what addresses require relocation is preserved in the object (and subsequently the executable) file. The -L option will generate in the output file one null relocation record for each relocation record in the input.

8.7.11 -LM

Similar to the above option, this preserves relocation records in the output file, but only segment relocations. This is used particularly for generating .EXE files to run under DOS.

8.7.12 -M*mapfile*

This causes the linker to generate a link map in the named file, or on the standard output if the file name is omitted. The format of the map file is illustrated by the example in figure 8-1.

The sections in the map file are as follows; first is a copy of the command line used to invoke the linker. This is followed by the version number of the object code in the first file linked, and the machine type. Then are listed all object files that were linked, along with their psect information. Libraries are listed, with each module within the library. The TOTALS section summarizes the psects from the object files. The SEGMENTS section summarizes major memory groupings. This will typically show RAM and ROM usage. The segment names are derived from the name of the first psect in the segment.

Lastly (not shown in the example) is a symbol table, where each global symbol is listed with its associated psect and link address.

8.7.13 -N

By default the symbol table in the link map will be sorted by name. This option will cause it to be sorted numerically, based on the value of the symbol.

8.7.14 -O*outfile*

This option allows specification of an output file name for the linker. The default output file name is **L.OBJ**. Use of this option will override the default.

8.7.15 -P*spec*

Psects are linked together and assigned addresses based on information in supplied to the linker via -P options. The argument to the -P option consists basically of comma separated sequences thus:

`-Ppsect=lnkaddr/ldaddr,psect=lnkaddr/ldaddr, ...`

Figure 8-1; Example map file

```
Linker command line:

-z -Mmap -pvectors=00h,text,strings,const,im2vecs -pbaseram=00h \
  -pramstart=08000h,data/im2vecs,bss/.,stack=09000h -pnvram=bss,heap \
  -oC:\TEMP\l.obj C:\HT-Z80\LIB\rtz80-s.obj hello.obj \
  C:\HT-Z80\LIB\z80-sc.lib

Object code version is 2.4
Machine type is Z80

                Name         Link     Load   Length     Selector
C:\HT-Z80\LIB\rtz80-s.obj
                vectors         0        0       71
                bss          8000     8000       24
                const          FB       FB        1        0
                text           72       72       82
hello.obj       text           F4       F4        7

C:\HT-Z80\LIB\z80-sc.lib
powerup.obj     vectors        71       71        1

TOTAL           Name         Link     Load   Length
        CLASS   CODE
                vectors         0        0       72
                const          FB       FB        1
                text           72       72       89

        CLASS   DATA
                bss          8000     8000       24

SEGMENTS        Name            Load    Length   Top    Selector

                vectors         000000  0000FC  0000FC       0
                bss             008000  000024  008024    8000
```

There are several variations, but essentially each psect is listed with its desired link and load addresses. The link and load addresses are either numbers as described above, or the names of other psects or classes, or special tokens. If a link address is omitted, the psect's link address will be derived from the top of the previous psect, e.g.

```
-Ptext=100h,data,bss
```

In this example the text psect is linked at 100 hex (its load address defaults to the same). The data psect will be linked (and loaded) at an address which is 100 hex plus the length of the text psect, rounded up as necessary if the data psect has a *RELOC=* value associated with it. Similarly, the bss psect will concatenate with the data psect.

If the load address is omitted entirely, it defaults to the same as the link address. If the slash (/) character is supplied, but no address is supplied after it, the load address will concatenate with the previous psect, e.g.

```
-Ptext=0,data=0/,bss
```

will cause both text and data to have a link address of zero, text will have a load address of 0, and data will have a load address starting after the end of text. The bss psect will concatenate with data for both link and load addresses.

The load address may be replaced with a dot (.) character. This tells the linker to set the load address of this psect to the same as its link address. The link or load address may also be the name of another (already linked) psect. This will explicitly concatenate the current psect with the previously specified psect, e.g.

```
-Ptext=0,data=8000h/,bss/. -Pnvram=bss,heap
```


This example shows text at zero, data linked at 8000h but loaded after text, bss is linked and loaded at 8000h plus the size of data, and nvram and heap are concatenated with bss. Note here the use of two -P options. Multiple -P options are processed in order.

If -A options have been used to specify address ranges for a class then this class name may be used in place of a link or load address, and space will be found in one of the address ranges. For example:

```
-ACODE=8000h-BFFEh,E000h-FFFEh
-Pdata=C000h/CODE
```

This will link data at C000h, but find space to load it in the address ranges associated with CODE. If no sufficiently large space is available, an error will result. Note that in this case the data psect will still be assembled into one contiguous block, whereas other psects in the class CODE will be distributed into the address ranges wherever they will fit. This means that if there are two or more psects in class CODE, they may be intermixed in the address ranges.

8.7.16 -S*psect=max*

A psect may have a maximum address associated with it. This is normally set in assembler code (with a *SIZE=* flag on a psect directive) but may also be set on the command line to the linker with this option. For example:

```
-Srbss=80h
```

8.7.17 -U*symbol*

This option will enter the specified symbol into the linker's symbol table as an undefined symbol. This is useful for linking entirely from libraries, or for linking a module from a library where the ordering has been arranged so that by default a later module will be linked.

8.7.18 -V*avmap*

To produce an Avocet format symbol file, the linker needs to be given a map file to allow it to map psect names to Avocet memory identifiers. The avmap file will normally be supplied with the compiler, or created automatically by the compiler driver as required. For a description of the format of an avmap file, see appendix "File Formats".

8.7.19 -W*num*

The -W option can be used to set the warning level, in the range -9 to 9, or the width of the map file, for values = 10.

8.7.20 -X

Local symbols can be suppressed from a symbol file with this option. Global symbols will always appear in the symbol file.

8.7.21 -Z

Some local symbols are compiler generated and not of interest in debugging. This option will suppress from the symbol file all local symbols that have the form of a single alphabetic character, followed by a digit string. The set of letters that can start a trivial symbol is currently "klfLSu". The -Z option will strip any local symbols starting with one of these letters, and followed by a digit string.

8.8 Invoking the Linker

The linker is called HLINK, and normally resides in the BIN subdirectory of the compiler installation directory. It may be invoked with no arguments, in which case it will prompt for input from standard input. If the standard input is a file, no prompts will be printed. This manner of invocation is generally useful if the number of arguments to LINK is large. Even if the list of files is too long to fit on one line,

continuation lines may be included by leaving a backslash (‘\’) at the end of the preceding line. In this fashion, LINK commands of almost unlimited length may be issued. For example a link command file called X.LNK and containing the following text:

```
-Z -OX.OBJ -MX.MAP \
-Ptext=0,data=0/,bss,nvram=bss/. \
X.OBJ Y.OBJ Z.OBJ C:\HT-Z80\LIB\Z80-SC.LIB
```

may be passed to the linker by one of the following:

```
HLINK @X.LNK
HLINK <X.LNK
```

Librarian Reference Manual

9.1 Introduction

Pacific C incorporates a librarian, to permit the building and maintaining of object code libraries. A library is a file comprising several object modules. Combining them into one file offers several advantages:

- Fewer files to link; the linker command line can be shortened considerably by using a few libraries rather than several object files.
- Faster linking; because there are fewer files to open, and because the linker reads a directory at the beginning of the library, it will link faster from a library than from multiple object files.
- Saving of disk space; combining object files into a library will reduce disk space because of the granular nature of filesystem allocation.
- Conditional linking; when linking from a library, the linker will link only those modules that define currently undefined but referenced symbols. If separate object files are specified, they will be linked unconditionally.

9.2 Usage

The librarian is called **LIBR** and is invoked as follows:

```
LIBR key file.lib file.obj ...
```

The *key* is a single letter which tells the librarian what action to perform using the single library file, and zero or more object files. The allowable actions are listed in table 9-1.

9.2.1 Creating a library

To create a library, you should use the *r* command. If the specified library file does not exist, it will be created and the listed object files appended to it. For example:

```
LIBR r test.lib file1.obj file2.obj file3.obj
```

will create the library file test.lib (if it does not already exist) and append to it the three object files. The order of the object modules in the library is the same as the order of the files on the command line.

9.2.2 Updating a library

The *r* command is also used when replacing or appending modules to a library. If a listed module already exists in the library, it will be replaced with the new object module, but will retain its position in the library. Any listed object files that do not already exist in the library will be appended to it. These will always appear after any existing modules in the library. Once the library is created, the only way to change the ordering is to delete the module and append it to the end.

9.2.3 Deleting library modules

To delete a module from a library, use the *d* command, e.g.

```
LIBR d test.lib file2.obj
```

This will delete module *file2.obj* from the library.

9.2.4 Extracting a module

Although the main use of libraries is to allow the linker to use them, it is sometimes required to extract a module from a library to an object file. The *x* command will extract modules. If any object modules are listed, they will be extracted. If no object modules are listed, the entire contents of the library will be extracted into invidivual object modules. This can be useful if you want to re-order a library. You can extract all modules and re-create the library with a specified ordering.

9.2.5 Listing a library

The *m* command will list the contents of a library by module name. If any object files are listed on the command line, only those modules will be included in the listing, otherwise the entire contents will be listed.

The *s* command will list all global symbols, defined and referenced, in either specified modules, or all modules if none are specified. In the symbol listing, a letter **D** will indicate that the symbol is defined in that module, while a letter **U** will indicate that it is undefined (referenced only) in that module.

9.3 Library ordering

Earlier versions of the linker would search a library only once, and thus the ordering of modules within the library was important, since if a module in the library referenced another earlier in the same library, the earlier module would not be linked. The current linker version corrects this problem by searching each library encountered repeatedly until no more references can be satisfied from that library. This means that the library ordering is not critical. However, a very badly ordered library can result in additional link time, and the linker will warn about a badly ordered library. In practice it is unlikely that a randomly ordered library will trigger this warning.

9.4 Creating libraries from PPD

It is possible to create a library using PPD. See the PPD manual for more details.

Appendix A
Error Messages

This appendix lists all possible error messages from the Pacific C compiler, with an explanation of each one. This information is also available on-line from within PPD.

#define syntax error

A macro definition has a syntax error. This could be due to a macro or formal parameter name that does not start with a letter or a missing closing parenthesis (‘)’).

#elif may not follow #else

If a #else has been used after #if, you cannot then use a #elif in the same conditional block.

#elif must be in an #if

#elif must be preceded by a matching #if line. If there is an apparently corresponding #if line, check for things like extra #endif’s, or improperly terminated comments.

#else may not follow #else

There can be only one #else corresponding to each #if.

#else must be in an #if

#else can only be used after a matching #if.

#endif must be in an #if

There must be a matching #if for each #endif. Check for the correct number of #ifs.

#error: *

This is a programmer generated error; there is a directive causing a deliberate error. This is normally used to check compile time defines etc.

#if ... sizeof() syntax error

The preprocessor found a syntax error in the argument to sizeof, in a #if expression. Probable causes are mismatched parentheses and similar things.

#if ... sizeof: bug, unknown type code *

The preprocessor has made an internal error in evaluating a sizeof() expression. Check for a malformed type specifier.

#if ... sizeof: illegal type combination

The preprocessor found an illegal type combination in the argument to sizeof() in a #if expression. Illegal combinations include such things as “short long int”.

#if bug, operand = *

The preprocessor has tried to evaluate an expression with an operator it does not understand. This is an internal error.

#if sizeof() error, no type specified

Sizeof() was used in a preprocessor #if expression, but no type was specified. The argument to sizeof() in a preprocessor expression must be a valid simple type, or pointer to a simple type.

#if sizeof, unknown type *

An unknown type was used in a preprocessor sizeof(). The preprocessor can only evaluate sizeof() with basic types, or pointers to basic types.

#if value stack overflow

The preprocessor filled up its expression evaluation stack in a #if expression. Simplify the expression - it probably contains too many parenthesized subexpressions.

#if, #ifdef, or #ifndef without an argument

The preprocessor directives #if, #ifdef and #ifndef must have an argument. The argument to #if should be an expression, while the argument to #ifdef or #ifndef should be a single name.

#include syntax error

The syntax of the filename argument to #include is invalid. The argument to #include must be a valid file name, either enclosed in double quotes (“”) or angle brackets (<>). For example: #include“afile.h” #include<otherfile.h> Spaces should not be included, and the closing quote or bracket must be present. There should be nothing else on the line.

‘.’ expected after ‘..’

The only context in which two successive dots may appear is as part of the ellipsis symbol, which must have 3 dots.

‘case’ not in switch

A case statement has been encountered but there is no enclosing switch statement. A case statement may only appear inside the body of a switch statement.

‘default’ not in switch

A label has been encountered called “default” but it is not enclosed by a switch statement. The label “default” is only legal inside the body of a switch statement.

(expected

An opening parenthesis was expected here. This must be the first token after a while, for, if, do or asm keyword.

) expected

A closing parenthesis was expected here. This may indicate you have left out a parenthesis in an expression, or you have some other syntax error.

, expected

A comma was expected here. This probably means you have left out the comma between two identifiers in a declaration list. It may also mean that the immediately preceding type name is misspelled, and has thus been interpreted as an identifier.

-s, too few values specified in *

The list of values to the preprocessor -S option is incomplete. This should not happen if the preprocessor is being invoked by the compiler driver or HPD.

-s, too many values, * unused

There were too many values supplied to a -S preprocessor option.

... illegal in non-prototype arg list

The ellipsis symbol may only appear as the last item in a prototyped argument list. It may not appear on its own, nor may it appear after argument names that do not have types.

: expected

A colon is missing in a case label, or after the keyword “default”. This often occurs when a semicolon is accidentally typed instead of a colon.

; expected

A semicolon is missing here. The semicolon is used as a terminator in many kinds of statements, e.g. do .. while, return etc.

= expected

An equal sign was expected here.

] expected

A closing square bracket was expected in an array declaration or an expression using an array index.

a parameter may not be a function

A function parameter may not be a function. It may be a pointer to a function, so perhaps a “*” has been omitted from the declaration.

absolute expression required

An absolute expression is required in this context.

add_reloc - bad size

This is an internal error that should never happen. The assembler may be corrupted, and should be re-installed from the original distribution disks.

argument * conflicts with prototype

The argument specified (argument 1 is the left most argument) of this function declaration does not agree with a previous prototype for this function.

argument list conflicts with prototype

The argument list in a function definition is not the same as a previous prototype for that function. Check that the number and types of the arguments are all the same.

argument redeclared: *

The specified argument is declared more than once in the same argument list.

arguments redeclared

The arguments of the function have been declared differently in two or more places.

arithmetic overflow in constant expression

A constant expression has been evaluated by the code generator that has resulted in a value that is too big for the type of the expression, e.g. trying to store the value 256 in a “char”.

array dimension on * ignored

An array dimension on a function parameter is ignored, because the argument is actually converted to a pointer when passed. Thus arrays of any size may be passed.

array dimension redeclared
An array dimension has been declared as a different non-zero value from its previous declaration. It is acceptable to redeclare the size of an array that was previously declared with a zero dimension, but not otherwise.

array index out of bounds
An array is being indexed with a constant value that is less than zero, or greater than or equal to the number of elements in the array.

assertion
An internal error has occured in the compiler. Contact HI-TECH technical support.

assertion failed: *
This is an internal error. Contact HI-TECH Software technical support.

attempt to modify const object
Objects declared “const” may not be assigned to or modified in any other way..

auto variable * should not be qualified
An auto variable should not have qualifiers such as “near” or “far” associated with it. Its storage class is implicitly defined by the stack organization.

bad #if ... defined() syntax
The defined() pseudo-function in a preprocessor expression requires its argument to be a single name. The name must start with a letter. It should be enclosed in parentheses.

bad ‘-p’ format
The “-P” option given to the linker is malformed.

bad -a spec: *
The format of a -A specification, giving address ranges to the linker, is invalid. The correct format is:

```
-Aclass=low-high
```

where class is the name of a psect class, and low and high are hex numbers.

bad -m option: *
The code generator has been passed a -M option that it does not understand. This should not happen if it is being invoked by a standard compiler driver.

bad -q option *
The first pass of the compiler has been invoked with a -Q option, to specify a type qualifier name, that is badly formed.

bad arg * to tysize
This is an internal error that should not happen. If this occurs, re-install the compiler from the original distribution disks, as it could represent a corrupted executable file.

bad arg to im
The opcode “IM” only takes the constants 0, 1 or 2 as an argument.

bad bconfloat - *
This is an internal code generator error. Contact HI-TECH technical support with full details of the code that caused this error.

bad bit number
A bit number must be an absolute expression in the range 0-7.

bad bitfield type
A bitfield may only have a type of int.

bad character const
This character constant is badly formed.

bad character in extended tekhex line *
This is an internal error in objtohex and should never occur.

bad checksum specification
A checksum list supplied to the linker is syntatically incorrect.

bad combination of flags
The combination of options supplied to objtohex is invalid.

bad complex relocation
The linker has been asked to perform complex relocation that is not syntactically correct. Probably means a corrupted object file.

bad confloat - *
This is an internal code generator error. Contact HI-TECH technical support with full details of the code that caused this error.

bad conval - *
This is an internal code generator error. Contact HI-TECH technical support with full details of the code that caused this error.

bad dimensions
The code generator has been passed a declaration that results in an array having a zero dimension.

bad element count expr
There is an error in the intermediate code. Try re-installing the compiler from the distribution disks, as this could be caused by a corrupted file.

bad gn
This is an internal code generator error. Contact HI-TECH technical support with full details of the code that caused this error.

bad high address in -a spec
The high address given in a -A specification is invalid: it should be a valid number, in decimal, octal or hexadecimal radix. The radix is specified by a trailing O (for octal) or H for hex. Decimal is default.

bad int. code
The code generator has been passed input that is not syntatically correct.

bad load address in -a spec
The load address given in a -A specification is invalid: it should be a valid number, in decimal, octal or hexadecimal radix. The radix is specified by a trailing O (for octal) or H for hex. Decimal is default.

bad low address in -a spec
The low address given in a -A specification is invalid: it should be a valid number, in decimal, octal or hexadecimal radix. The radix is specified by a trailing O (for octal) or H for hex. Decimal is default.

bad mod '+' for how = *
Internal error - Contact HI-TECH.

bad non-zero node in call graph
The linker has encountered a top level node in the call graph that is referenced from lower down in the call graph. This probably means the program has indirect recursion, which is not allowed when using a compiled stack.

bad object code format
The object code format of this object file is invalid. This probably means it is either truncated, corrupted, or not a HI-TECH object file.

bad op * to revlog
This is an internal code generator error. Contact HI-TECH technical support with full details of the code that caused this error.

bad op * to swaplog
This is an internal code generator error. Contact HI-TECH technical support with full details of the code that caused this error.

bad op: *
This is caused by an error in the intermediate code file. You may have run out of disk (or RAMdisk) space for temporary files.

bad operand
This operand is invalid. Check the syntax.

bad origin format in spec
The origin format in a -p option is not a validly formed decimal, octal or hex number. A hex number must have a trailing H.

bad pragma *
The code generator has been passed a "pragma" directive that it does not understand.

bad record type *
This indicates that the object file is not a valid HI-TECH object file.

bad relocation type
This is an internal assembler error. Contact HI-TECH technical support with full details of the code that caused this error.is

bad ret_mask
This is an internal code generator error. Contact HI-TECH technical support with full details of the code that caused this error.

bad segment fixups
This is an obscure message from objtohex that is not likely to occur in practice.

bad segspec *

The segspec option (-G) to the linker is invalid. The correct form of a segspec option is along the following lines:

```
-Gnxc+o
```

where n stands for the segment number, x is a multiplier symbol, c is a constant (multiplier) and o is a constant offset. For example the option

```
-Gnx4+16
```

would assign segment selectors starting from 16, and incrementing by 4 for each segment, i.e. in the order 16, 20, 24 etc.

bad size in -s option

The size part of a -S option is not a validly formed number. The number must be a decimal, octal or hex number. A hex number needs a trailing H, and an octal number a trailing O. All others are assumed to be decimal.

bad size list

The first pass of the compiler has been invoked with a -Z option, specifying sizes of types, that is badly formed.

bad storage class

The storage class “auto” may only be used inside a function. A function parameter may not have any storage class specifier other than “register”. If this error is issued by the code generator, it could mean that the intermediate code file is invalid. This could be caused by running out of disk (or RAMdisk) space.

bad string * in psect pragma

The code generator has been passed a “pragma psect” directive that has a badly formed string. “Pragma psect” should be followed by something of the form “oldname=newname”.

bad sx

This is an internal code generator error. Contact HI-TECH technical support with full details of the code that caused this error.

bad u usage

This is an internal code generator error. Contact HI-TECH technical support with full details of the code that caused this error.

bad variable syntax

There is an error in the intermediate code file. This could be caused by running out of disk (or RAMdisk) space for temporary files.

bad which * after i

This is an internal compiler error. Contact HI-TECH Software technical support.

binary digit expected

A binary digit was expected. The format for a binary number is

```
0Bxxx
```

where xxx is a string of zeroes and ones, e.g.

```
0B0110
```

bit field too large * bits)

The maximum number of bits in a bit field is the same as the number of bits in an “int”.

bug: illegal __ macro *

This is an internal error in the preprocessor that should not happen.

can’t allocate space for port variables: *

“Port” variables may only be declared “extern” or have an absolute address associated via the “@ address” construct. They may not be declared in such a way that would require the compiler to allocate space for them.

can’t be both far and near

It is illegal to qualify a type as both far and near.

can’t be long

Only “int” and “float” can be qualified with “long”. Thus combinations like “long char” are illegal.

can’t be register

Only function parameters or auto (local) variables may be declared “register”.

can’t be short

Only “int” can be modified with short. Thus combinations like “short float” are illegal.

can’t be unsigned

There is no such thing as an unsigned floating point number.

can’t call an interrupt function

A function qualified “interrupt” can’t be called from other functions. It can only be called by a hardware (or software) interrupt. This is because an interrupt function has special function entry and exit code that is appropriate only for calling from an interrupt. An “interrupt” function can call other non-interrupt functions.

can’t create *

The named file could not be created. Check that all directories in the path are present.

can’t create cross reference file *

The cross reference file could not be created. Check that all directories are present. This can also be caused by the assembler running out of memory.

can’t create temp file

The compiler was unable to create a temporary file. Check the DOS Environment variable TEMP (and TMP) and verify it points to a directory that exists, and that there is space available on that drive. For example, AUTOEXEC.BAT should have something like:

```
SET TEMP=C:\TEMP
```

where the directory C:\TEMP exists.

can’t create temp file *

The compiler could not create the temporary file named. Check that all the directories in the file path exist.

can’t create xref file *

An output file for the cross reference could not be created.

can’t enter abs psect

This is an internal assembler error. Contact HI-TECH technical support with full details of the code that caused this error.

can’t find op

Internal error - Contact HI-TECH.

can’t find space for psect * in segment *

The named psect cannot be placed in the specified segment. This probably means your code has got too big for the specified ROM space (using -A options).

can’t generate code for this expression

This expression is too difficult for the code generator to handle. Try simplifying the expression, e.g. using a temporary variable to hold an intermediate result.

can’t have ‘port’ variable: *

The qualifier “port” can be used only with pointers or absolute variables. You cannot define a port variable as the compiler does not allocate space for port variables. You can declare an external port variable.

can’t have ‘signed’ and ‘unsigned’ together

The type modifiers signed and unsigned cannot be used together in the same declaration, as they have opposite meaning.

can’t have an array of bits or a pointer to bit

IT is not legal to have an array of bits, or a pointer to bit.

can’t have array of functions

You can’t have an array of functions. You can however have an array of pointers to functions. The correct syntax for an array of pointers to functions is “int (* arrayname[])();”. Note that parentheses are used to associate the star (*) with the array name before the parentheses denoting a function.

can’t initialise auto aggregates

You can’t initialise structures or arrays local to a function unless they are declared “static”.

can’t initialize arg

A function argument can’t have an initialiser. The initialisation of the argument happens when the function is called and a value is provided for the argument by the calling function.

can’t mix proto and non-proto args

A function declaration can only have all prototyped arguments (i.e. with types inside the parentheses) or all K&R style args (i.e. only names inside the parentheses and the argument types in a declaration list before the start of the function body).

can’t open

A file can’t be opened - check spelling.

can't open *
The named file could not be opened. Check the spelling and the directory path. This can also be caused by running out of memory.

can't open avmap file *
A file required for producing Avocet format symbol files is missing. Try re-installing the compiler.

can't open checksum file *
The checksum file specified to objtohex could not be opened. Check spelling etc.

can't open command file *
The command file specified could not be opened for reading. Check spelling!

can't open include file *
The named include file could not be opened. Check spelling. This can also be caused by running out of memory, or running out of file handles.

can't open input file *
The specified input file could not be opened. Check the spelling of the file name.

can't open output file *
The specified output file could not be created. This could be because a directory in the path name does not exist.

can't reopen *
The compiler could not reopen a temporary file it had just created.

can't seek in *
The linker can't seek in the specified file. Make sure the output file is a valid filename.

can't take address of register variable
A variable declared "register" may not have storage allocated for it in memory, and thus it is illegal to attempt to take the address of it by applying the "&" operator.

can't take sizeof func
Functions don't have sizes, so you can't take use the "sizeof" operator on a function.

can't take sizeof(bit)
You can't take sizeof a bit value, since it is smaller than a byte.

can't take this address
The expression which was the object of the "&" operator is not one that denotes memory storage ("an lvalue") and therefore its address can not be defined.

can't use a string in an #if
The preprocessor does not allow the use of strings in #if expressions.

cannot get memory
The linker is out of memory! This is unlikely to happen, but removing TSR's etc is the cure.

cannot open
A file cannot be opened - check spelling.

cannot open include file *

The named include file could not be opened for reading by the preprocessor. Check the spelling of the filename. If it is a standard header file, not in the current directory, then the name should be enclosed in angle brackets (<>) not quotes.

cast type must be scalar or void

A typecast (an abstract type declarator enclosed in parentheses) must denote a type which is either scalar (i.e. not an array or a structure) or the type “void”.

char const too long

A character constant enclosed in single quotes may not contain more than one character.

character not valid at this point in format specifier

The printf() style format specifier has an illegal character.

close error (disk space?)

When the compiler closed a temporary file, an error was reported. The most likely cause of this is that there was insufficient space on disk for the file. Note that temporary files may be created on a RAM disk, so even if your hard disk has ample space it is still possible to get this error.

common symbol psect conflict: *

A common symbol has been defined to be in more than one psect.

complex relocation not supported for -r or -l options yet

The linker was given a -R or -L option with file that contain complex relocation. This is not yet supported.

conflicting fnconf records

This is probably caused by multiple run-time startoff module. Check the linker arguments, or “Object Files...” in HPD.

constant conditional branch

A conditional branch (generated by an “if” statement etc.) always follows the same path. This may indicate an expression with missing or badly placed parentheses, causing the evaluation to yield a value different to what you expected, or it may be because you have written something like “while(1)”. To produce an infinite loop, use “for(;;)”.

constant conditional branch: possible use of = instead of ==

There is an expression inside an if or other conditional construct, where a constant is being assigned to a variable. This may mean you have inadvertentley used an assignment (=) instead of a compare (==).

constant expression required

In this context an expression is required that can be evaluated to a constant at compile time.

constant left operand to ?

The left operand to a conditional operator (?) is constant, thus the result of the tertiary operator ?: will always be the same.

constant operand to || or &&

One operand to the logical operators || or && is a constant. Check the expression for missing or badly placed parentheses.

constant relational expression

There is a relational expression that will always be true or false. This may be because e.g. you are comparing an unsigned number with a negative value, or comparing a variable with a value greater than the largest number it can represent.

control line * within macro expansion

A preprocessor control line (one starting with a #) has been encountered while expanding a macro. This should not happen.

declaration of * hides outer declaration

An object has been declared that has the same name as an outer declaration (i.e. one outside and preceding the current function or block). This is legal, but can lead to accidental use of one variable when the outer one was intended.

declarator too complex

This declarator is too complex for the compiler to handle. Examine the declaration and find a way to simplify it. If the compiler finds it too complex, so will anybody maintaining the code.

def[bmsf] in text psect

The assembler file supplied to the optimizer is invalid.

default case redefined

There is only allowed to be one “default” label in a switch statement. You have more than one.

deff not supported in cp/m version

The CP/M hosted assembler does not support floating point.

degenerate signed comparison

There is a comparision of a signed value with the most negative value possible for this type, such that the comparision will always be true or false. E.g.

```
char     c;

if(c >= -128)
```

will always be true, because an 8 bit signed char has a maximum negative value of -128.

degenerate unsigned comparison

There is a comparision of an unsigned value with zero, which will always be true or false. E.g.

```
unsigned char    c;

if(c >= 0)
```

will always be true, because an unsigned value can never be less than zero.

digit out of range

A digit in this number is out of range of the radix for the number, e.g. using the digit 8 in an octal number, or hex digits A-F in a decimal number. An octal number is denoted by the digit string commencing with a zero, while a hex number starts with “0X” or “0x”.

dimension required
Only the most significant (i.e. the first) dimension in a multi-dimension array may not be assigned a value. All succeeding dimensions must be present.

directive not recognized
An assembler directive is unrecognized. Check spelling.

divide by zero in #if, zero result assumed
Inside a #if expression, there is a division by zero which has been treated as yielding zero.

division by zero
A constant expression that was being evaluated involved a division by zero.

double float argument required
The printf format specifier corresponding to this argument is %f or similar, and requires a floating point expression. Check for missing or extra format specifiers or arguments to printf.

duplicate -m flag
The linker only likes to see one -m flag, unless one of them does not specify a file name. Two map file names are more than it can handle!

duplicate case label
There are two case labels with the same value in this switch statement.

duplicate label *
The same name is used for a label more than once in this function. Note that the scope of labels is the entire function, not just the block that encloses a label.

duplicate qualifier
There are two occurences of the same qualifier in this type specification. This can occur either directly or through the use of a typedef. Remove the redundant qualifier.

duplicate qualifier key *
This qualifier key (given via a -Q option, has been used twice.

duplicate qualifier name *
A duplicate qualifier name has been specified to P1 via a -Q option. This should not occur if the standard compiler drivers are used.

end of file within macro argument from line *
A macro argument has not been terminated. This probably means the closing parenthesis has been omitted from a macro invocation. The line number given is the line where the macro argument started.

end of string in format specifier
The format specifier for the printf() style function is malformed.

entry point multiply defined
There is more than one entry point defined in the object files given the linker.

enum tag or { expected
After the keyword “enum” must come either an identifier that is or will be defined as an enum tag, or an opening brace.

eof in #asm
An end of file has been encountered inside a #asm block. This probably means the #endasm is missing or misspelt.

eof in comment
End of file was encountered inside a comment. Check for a missing closing comment flag.

eof inside conditional
END-of-FILE was encountered while scanning for an “endif” to match a previous “if”.

eof inside macro def’n
End-of-file was encountered while processing a macro definition. This means there is a missing “endm” directive.

eof on string file
P1 has encountered an unexpected end-of-file while re-reading its file used to store constant strings before sorting and merging. This is most probably due to running out of disk space. Check free disk space, OR RAM disk size. The RAM disk may be too small, if it is used for temporary files.

error closing output file
The compiler detected an error when closing a file. This most probably means there is insufficient disk space, but note that the file could be on a RAM disk, so even if you have ample space on your hard disk, this error can still occur. If this is the case, increase the size of your ram disk or move your temporary file area onto the hard disk.

error in format string
There is an error in the format string here. The string has been interpreted as a printf() style format string, and it is not syntactically correct. If not corrected, this will cause unexpected behaviour at run time.

evaluation period has expired
The evaluation period for this compiler has expired. Contact HI-TECH to purchase a full licence.

expand - bad how
This is an internal code generator error. Contact HI-TECH technical support with full details of the code that caused this error.

expand - bad which
This is an internal code generator error. Contact HI-TECH technical support with full details of the code that caused this error.

expected ‘-’ in -a spec
There should be a minus sign (-) between the high and low addresses in a -A spec, e.g.

```
-AROM=1000h-1FFFh
```

exponent expected
A floating point constant must have at least one digit after the “e” or “E”.

expression error
There is a syntax error in this expression, OR there is an error in the intermediate code file. This could be caused by running of disk (or RAMdisk) space.

expression generates no code

This expression generates no code. Check for things like leaving off the parentheses in a function call.

expression stack overflow at op *

Expressions in #if lines are evaluated using a stack with a size of 128. It is possible for very complex expressions to overflow this. Simplify the expression.

expression syntax

This expression is badly formed and cannot be parsed by the compiler.

expression too complex

This expression has caused overflow of the compiler's internal stack and should be re-arranged or split into two expressions.

external declaration inside function

A function contains an "extern" declaration. This is legal but is invariably A Bad Thing as it restricts the scope of the function declaration to the function body. This means that if the compiler encounters another declaration, use or definition of the extern object later in the same file, it will no longer have the earlier declaration and thus will be unable to check that the declarations are consistent. This can lead to strange behaviour of your program or signature errors at link time. It will also hide any previous declarations of the same thing, again subverting the compiler's type checking. As a general rule, always declare "extern" variables and functions outside any other functions.

fast interrupt can't be used in large model

The large model (bank switched) does not support fast interrupts, as the alternate register set is used for bank switching.

field width not valid at this point

A field width may not appear at this point in a printf() type format specifier.

filename work buffer overflow

A filename constructed while looking for an include file has exceeded the length of an internal buffer. Since this buffer is 4096 bytes long, this is unlikely to happen.

fixup overflow in expression *

The linker was asked to relocate (fixup) an item that would not fit back into the space after relocation. For example this will occur if a byte size object is initialized with an address that is bigger than 255. This error occurred in a complex expression.

fixup overflow referencing *

The linker was asked to relocate (fixup) an item that would not fit back into the space after relocation. For example this will occur if a byte size object is initialized with an address that is bigger than 255.

flag * unknown

This option used on a "PSECT" directive is unknown to the assembler.

float param coerced to double

Where a non-prototyped function has a parameter declared as "float", the compiler convert this into a "double float". This is because the default C type conversion conventions provide that when a floating point number is passed to a non-prototyped function, it will be converted to double. It is important that the function declaration be consistent with this convention.

floating exponent too large

The exponent of the floating point number is too large. For the Z80 the largest floating point exponent is decimal 19.

floating number expected

The arguments to the "DEFF" pseudo-op must be valid floating point numbers.

formal parameter expected after #

The stringization operator # (not to be confused with the leading # used for preprocessor control lines) must be followed by a formal macro parameter. If you need to stringize a token, you will need to define a special macro to do it, e.g.

```
#define  __mkstr__(x)      #x
```

then use __mkstr__(token) wherever you need to convert a token into a string.

function * appears in multiple call graphs: rooted at *

This function can be called from both main line code and interrupt code. Use the reentrant keyword, if this compiler supports it, or recode to avoid using local variables or parameters, or duplicate the function.

function * is never called

This function is never called. This may not represent a problem, but space could be saved by removing it. If you believe this function should be called, check your source code.

function body expected

Where a function declaration is encountered with K&R style arguments (i.e. argument names but no types inside the parentheses) a function body is expected to follow.

function declared implicit int

Where the compiler encounters a function call of a function whose name is presently undefined, the compiler will automatically declare the function to be of type "int", with unspecified (K&R style) parameters. If a definition of the function is subsequently encountered, it is possible that its type and arguments will be different from the earlier implicit declaration, causing a compiler error. The solution is to ensure that all functions are defined or at least declared before use, preferably with prototyped parameters. If it is necessary to make a forward declaration of a function, it should be preceded with the keywords "extern" or "static" as appropriate.

function does not take arguments

This function has no parameters, but it is called here with one or more arguments.

function is already 'extern'; can't be 'static'

This function was already declared extern, possibly through an implicit declaration. It has now been redeclared static, but this redeclaration is invalid. If the problem has arisen because of use before definition, either move the definition earlier in the file, or place a static forward definition earlier in the file, e.g.

```
static int fred(void);
```

function or function pointer required

Only a function or function pointer can be the subject of a function call. This error can be produced when an expression has a syntax error resulting in a variable or expression being followed by an opening parenthesis ("(") which denotes a function call.

functions can't return arrays

A function can return only a scalar (simple) type or a structure. It cannot return an array.

functions can't return functions

A function cannot return a function. It can return a function pointer. A function returning a pointer to a function could be declared like this: int (* (name()))(). Note the many parentheses that are necessary to make the parts of the declaration bind correctly.

functions nested too deep

This error is unlikely to happen with C code, since C cannot have nested functions!

garbage after operands

There is something on this line after the operands other than a comment. This could indicate an operand error.

garbage on end of line

There were non-blank and non-comment characters after the end of the operands for this instruction. Note that a comment must be started with a semicolon.

hex digit expected

After "0x" should follow at least one of the hex digits 0-9 and A-F or a-f.

ident records do not match

The object files passed to the linker do not have matching ident records. This means they are for different processor types.

identifier expected

Inside the braces of an "enum" declaration should be a comma-separated list of identifiers.

identifier redefined: *

This identifier has already been defined. It cannot be defined again.

identifier redefined: * (from line *)

This identifier has been defined twice. The 'from line ' value is the line number of the first declaration.

illegal # command *

The preprocessor has encountered a line starting with #, but which is not followed by a recognized control keyword. This probably means the keyword has been misspelt. Legal control keywords are:

```
assert  asm     define  elif    else    endasm endif   error   if
ifdef   ifndef  include line    pragma  undef
```

illegal #if line

There is a syntax error in the expression following #if. Check the expression to ensure it is properly constructed.

illegal #undef argument

The argument to #undef must be a valid name. It must start with a letter.

illegal '#' directive
The compiler does not understand the "#" directive. It is probably a misspelling of a pre-processor "#" directive.

illegal -o flag
This -o flag is illegal. A -o option to the linker must have a filename. There should be no space between the filenam and the -o, e.g.

```
-ofile.obj
```

illegal -p flag
The -p flag needs stuff after it. This is used for defining psect mapping to the linker.

illegal character *
This character is illegal.

illegal character * decimal) in #if
The #if expression had an illegal character. Check the line for correct syntax.

illegal character * in #if
There is a character in a #if expression that has no business being there. Valid characters are the letters, digits and those comprising the acceptable operators.

illegal conversion
This expression implies a conversion between imcompatible types, e.g. a conversion of a structure type into an integer.

illegal conversion between pointer types
A pointer of one type (i.e. pointing to a particular kind of object) has been converted into a pointer of a different type. This will usually mean you have used the wrong variable, but if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed.

illegal conversion of integer to pointer
An integer has been assigned to or otherwise converted to a pointer type. This will usually mean you have used the wrong variable, but if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed.

illegal conversion of pointer to integer
A pointer has been assigned to or otherwise converted to a integral type. This will usually mean you have used the wrong variable, but if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed.

illegal flag *
This flag is unrecognized.

illegal function qualifier(s)
A qualifier such as "const" or "volatile" has been applied to a function. These qualifiers only make sense when used with an lvalue (i.e. an expression denoting memory storage). Perhaps you left out a star ("*") indicating that the function should return a pointer to a qualified object.

illegal initialisation
You can't initialise a "typedef" declaration, because it does not reserve any storage that could be initialised.

illegal operation on a bit variable
Not all operations on bit variables are supported. This operation is one of those.

illegal operator in #if
A #if expression has an illegal operator. Check for correct syntax.

illegal or too many -p flags
There are too many -p options to the linker. Combine some of them.

illegal record type
There is an error in an object file. This is either an invalid object file, or an internal error in the linker. Try recreating the object file.

illegal relocation size: *
There is an error in the object code format read by the linker. This either means you are using a linker that is out of date, or that there is an internale error in the assembler or linker.

illegal relocation type: *
An object file contained a relocation record with an illegal relocation type. This probably means the file is corrupted or not an object file.

illegal switch *
This command line option was not understood.

illegal type for array dimension
An array dimension must be either an integral type or an enumerated value.

illegal type for index expression
An index expression must be either integral or an enumerated value.

illegal type for switch expression
A "switch" operation must have an expression that is either an integral type or an enumerated value.

illegal use of void expression
A void expression has no value and therefore you can't use it anywhere an expression with a value is required, e.g. as an operand to an arithmetic operator.

image too big
The program image being constructed by objtohex is too big for its virtual memory system.

implicit conversion of float to integer
A floating point value has been assigned or otherwise converted to an integral type. This could result in truncation of the floating point value. A typecast will make this warning go away.

implicit return at end of non-void function
A function which has been declared to return a value has an execution path that will allow it to reach the end of the function body, thus returning without a value. Either insert a return statement with a value, or if the function is not to return a value, declare it "void".

implict signed to unsigned conversion

A signed number is being assigned or otherwise converted to a larger unsigned type. Under the ANSI "value preserving" rules, this will result in the signed value being first sign-extended to a signed number the size of the target type, then converted to unsigned (which involves no change in bit pattern). Thus an unexpected sign extension can occur. To ensure this does not happen, first convert the signed value to an unsigned equivalent, e.g. if you want to assign a signed char to an unsigned int, first typecast the char value to "unsigned char".

inappropriate 'else'

An "else" keyword has been encountered that cannot be associated with an "if" statement. This may mean there is a missing brace or other syntactic error.

inappropriate break/continue

A "break" or "continue" statement has been found that is not enclosed in an appropriate control structure. "continue" can only be used inside a "while", "for" or "do while" loop, while "break" can only be used inside those loops or a "switch" statement.

include files nested too deep

Macro expansions and include file handling have filled up the assembler's internal stack. The maximum number of open macros and include files is 30.

incompatible intermediate code version; should be *

The intermediate code file produced by P1 is not the correct version for use with this code generator. This is either that incompatible versions of one or more compilers have been installed in the same directory, or a temporary file error has occured leading to corruption of a temporary file. Check the setting of the TEMP environment variable. If it refers to a long path name, change it to something shorter.

incomplete * record body: length = *

An object file contained a record with an illegal size. This probably means the file is truncated or not an object file.

incomplete record

The object file passed to objtohex is corrupted.

incomplete record: *

An object code record is incomplete. This is probably due to a corrupted or invalid object module. Re-compile the source file, watching for out of disk space errors etc.

incomplete record: type = * length = *

This indicates that the object file is not a valid HI-TECH object file, or that it has been truncated, possibly due to running out of disk or RAMdisk space.

inconsistent storage class

A declaration has conflicting storage classes. Only one storage class should appear in a declaration.

inconsistent type

Only one basic type may appear in a declaration, thus combinations like "int float" are illegal.

index offset too large

An offset on a Z80 indexed addressing form must lie in the range -128 to 127.

initialisation syntax

The initialisation of this object is syntactically incorrect. Check for the correct placement and number of braces and commas.

initializer in 'extern' declaration

A declaration containing the keyword "extern" has an initialiser. This overrides the "extern" storage class, since to initialise an object it is necessary to define (i.e. allocate storage for) it.

insufficient memory for macro def'n

There is not sufficient memory to store a macro definition.

integer constant expected

A colon appearing after a member name in a structure declaration indicates that the member is a bitfield. An integral constant must appear after the colon to define the number of bits in the bitfield.

integer expression required

In an "enum" declaration, values may be assigned to the members, but the expression must evaluate to a constant of type "int".

integral argument required

An integral argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments.

integral type required

This operator requires operands that are of integral type only.

invalid disable: *

This is an internal preprocessor error that should not occur.

invalid format specifier or type modifier

The format specifier or modifier in the printf() style string is illegal for this particular format.

invalid number syntax

The syntax of a number is invalid. This can be, e.g. use of 8 or 9 in an octal number, or other malformed numbers.

jump out of range

A short jump ("JR") instruction has been given an address that is more than 128 bytes away from the present location. Use the "JP" opcode instead.

label identifier expected

An identifier denoting a label must appear after "goto".

lexical error

An unrecognized character or token has been seen in the input.

library * is badly ordered

This library is badly ordered. IT will still link correctly, but it will link faster if better ordered.

line does not have a newline on the end

The last line in the file is missing the newline (linefeed, hex 0A) from the end. Some editors will create such files, which can cause problems for include files. The ANSI C standard requires all source files to consist of complete lines only.

line too long
This line is too long. It will not fit into the compiler's internal buffers. It would require a line over 1000 characters long to do this, so it would normally only occur as a result of macro expansion.

local illegal outside macros
The "LOCAL" directive is only legal inside macros. It defines local labels that will be unique for each invocation of the macro.

local psect '*' conflicts with global psect of same name
A local psect may not have the same name as a global psect.

logical type required
The expression used as an operand to "if", "while" statements or to boolean operators like ! and && must be a scalar integral type.

long argument required
A long argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments.

macro * wasn't defined
A macro name specified in a -U option to the preprocessor was not initially defined, and thus cannot be undefined.

macro argument after % must be absolute
The argument after % in a macro call must be absolute, as it must be evaluated at macro call time.

macro argument may not appear after local
The list of labels after the directive "LOCAL" may not include any of the formal parameters to the macro.

macro expansions nested too deep
Macro expansions in the assembler are nested too deep. The limit is 30 macros and include files nested at one time.

macro work area overflow
The total length of a macro expansion has exceeded the size of an internal table. This table is normally 8192 bytes long. Thus any macro expansion must not expand into a total of more than 8K bytes.

member * redefined
This name of this member of the struct or union has already been used in this struct or union.

members cannot be functions
A member of a structure or a union may not be a function. It may be a pointer to a function. The correct syntax for a function pointer requires the use of parentheses to bind the star ("*") to the pointer name, e.g. "int (*name)();".

metaregister * can't be used directly
This is an internal compiler error. Contact HI-TECH Software technical support.

mismatched comparision

A comparison is being made between a variable or expression and a constant value which is not in the range of possible values for that expression, e.g. if you compare an unsigned character to the constant value 300, the result will always be false (not equal) since an unsigned character can NEVER equal 300. As an 8 bit value it can represent only 0-255.

misplaced '?' or ':', previous operator is *

A colon operator has been encountered in a #if expression that does not match up with a corresponding ? operator. Check parentheses etc.

misplaced constant in #if

A constant in a #if expression should only occur in syntactically correct places. This error is most probably caused by omission of an operator.

missing ')'

A closing parenthesis was missing from this expression.

missing '=' in class spec

A class spec needs an = sign, e.g.

```
-Ctext=ROM
```

missing ']'

A closing square bracket was missing from this expression.

missing arg to -a

The -a option requires the name of a qualifier as an argument.

missing arg to -u

The -U (undefine) option needs an argument, e.g.

```
-U_symbol
```

missing arg to -w

The -W option (listing width) needs a numeric argument.

missing argument to 'pragma psect'

The pragma 'psect' requires an argument of the form oldname=newname where oldname is an existing psect name known to the compiler, and newname is the desired new name. Example:

```
#pragma psect     bss=battery
```

missing basic type: int assumed

This declaration does not include a basic type, so int has been assumed. This declaration is not illegal, but it is preferable to include a basic type to make it clear what is intended.

missing key in avmap file

A file required for producing Avocet format symbol files is corrupted. Try re-installing the compiler.

missing memory key in avmap file

A file required for producing Avocet format symbol files is corrupted. Try re-installing the compiler.

missing name after pragma 'inline'

The 'inline' pragma has the syntax:

```
#pragma inline func_name
```

where func_name is the name of a function which is to be expanded to inline code. This pragma has no effect except on functions specially recognized by the code generator.

missing name after pragma 'printf_check'

The pragma 'printf_check', which enable printf style format string checking for a function, requires a function name, e.g.

```
#pragma printf_check sprintf
```

missing number after % in -p option

The % operator in a -p option (for rounding boundaries) must have a number after it.

missing number after * in -p option

The -p option has a missing number after a % sign.

missing number after pragma 'pack'

The pragma 'pack' requires a decimal number as argument. For example

```
#pragma pack(1)
```

will prevent the compiler aligning structure members onto anything other than one byte boundaries. Use this with caution as some processors enforce alignment and will not operate correctly if word fetches are made on odd boundaries (e.g. 68000, 8096).

mod by zero in #if, zero result assumed

A modulus operation in a #if expression has a zero divisor. The result has been assumed to be zero.

module has code below file base of *

This module has code below the address given, but the -C option has been used to specify that a binary output file is to be created that is mapped to this address. This would mean code from this module would have to be placed before the beginning of the file! Check for missing psect directives in assembler files.

multi-byte constant * isn't portable

Multi-byte constants are not portable, and in fact will be rejected by later passes of the compiler (this error comes from the preprocessor).

multiple free: *

This is an internal compiler error. Contact HI-TECH Software technical support.

multiply defined symbol *

This symbol has been defined in more than one place in this module.

near function should be static

A near function in the bank switched model should be static, as it cannot be called from another module.

nested #asm directive

It is not legal to nest #asm directives. Check for a missing or misspelt #endasm directive.

nested comments

This warning is issued when nested comments are found. A nested comment may indicate that a previous closing comment marker is missing or malformed.

no #asm before #endasm

A #endasm operator has been encountered, but there was no previous matching #asm.

no arg to -o

The assembler requires that an output file name argument be supplied after the “-O” option. No space should be left between the -O and the filename.

no case labels

There are no case labels in this switch statement.

no end record

This object file has no end record. This probably means it is not an object file.

no end record found

An object file did not contain an end record. This probably means the file is corrupted or not an object file.

no file arguments

The assembler has been invoked without any file arguments. It cannot assemble anything.

no identifier in declaration

The identifier is missing in this declaration. This error can also occur where the compiler has been confused by such things as missing closing braces.

no memory for string buffer

P1 was unable to allocate memory for the longest string encountered, as it attempts to sort and merge strings. Try reducing the number or length of strings in this module.

no psect specified for function variable/argument allocation

This is probably caused by omission of correct run-time startoff module. Check the linker arguments, or “Object Files...” in HPD.

no space for macro def’n

The assembler has run out of memory.

no start record: entry point defaults to zero

None of the object files passed to the linker contained a start record. The start address of the program has been set to zero. This may be harmless, but it is recommended that you define a start address in your startup module by using the “END” directive.

no. of arguments redeclared

The number of arguments in this function declaration does not agree with a previous declaration of the same function.

nodecount = *

This is an internal code generator error. Contact HI-TECH technical support with full details of the code that caused this error.

non-constant case label

A case label in this switch statement has a value which is not a constant.

non-prototyped function declaration: *

A function has been declared using old-style (K&R) arguments. It is preferable to use prototype declarations for all functions. If the function has no arguments, declare it as e.g. “int func(void)”.

non-scalar types can’t be converted

You can’t convert a structure, union or array to anything else. You can convert a pointer to one of those things, so perhaps you left out an ampersand (“&”).

non-void function returns no value

A function that is declared as returning a value has a “return” statement that does not specify a return value.

not a member of the struct/union *

This identifier is not a member of the structure or union type with which it used here.

not a variable identifier: *

This identifier is not a variable; it may be some other kind of object, e.g. a label.

not an argument: *

This identifier that has appeared in a K&R stype argument declarator is not listed inside the parentheses after the function name. Check spelling.

object code version is greater than *

The object code version of an object module is higher than the highest version the linker is known to work with. Check that you are using the correct linker.

object file is not absolute

The object file passed to objtohex has relocation items in it. This may indicate it is the wrong object file, or that the linker or objtohex have been given invalid options.

only functions may be qualified interrupt

The qualifier “interrupt” may not be applied to anything except a function.

only functions may be void

A variable may not be “void”. Only a function can be “void”.

only lvalues may be assigned to or modified

Only an lvalue (i.e. an identifier or expression directly denoting addressable storage) can be assigned to or otherwise modified. A typecast does not yield an lvalue. To store a value of different type into a variable, take the address of the variable, convert it to a pointer to the desired type, then dereference that pointer, e.g. “*(int *)&x = 1" is legal whereas ”(int)x = 1" is not.

only modifier l valid with this format

The only modifier that is legal with this format is l (for long).

only modifiers h and l valid with this format

Only modifiers h (short) and l (long) are legal with this printf() format specifier.

only register storage class allowed

The only storage class allowed for a function parameter is “register”.

oops! -ve number of nops required!

An internal error has occured. Contact HI-TECH.

operand error

The operand to this opcode is invalid. Check you assembler reference manual for the proper form of operands for this instruction.

operands of * not same pointer type

The operands of this operator are of different pointer types. This probably means you have used the wrong pointer, but if the code is actually what you intended, use a typecast to suppress the error message.

operands of * not same type

The operands of this operator are of different pointer. This probably means you have used the wrong variable, but if the code is actually what you intended, use a typecast to suppress the error message.

operator * in incorrect context

An operator has been encountered in a #if expression that is incorrectly placed, e.g. two binary operators are not separated by a value.

out of far memory

The compiler has run out of far memory. Try removing TSR's etc. If your system supports EMS memory, the compiler will be able to use up to 64K of this, so if it is not enable, try enabling EMS.

out of far memory (wanted * bytes)

The code generator could not allocate any more far memory. Try reducing the memory used by TSR's etc.

out of memory

The compiler has run out of memory. If you have unnecessary TSRs loaded, remove them. If you are running the compiler from inside another program, try running it directly from the command prompt. Similarly, if you are using HPD, try using the command line compiler driver instead.

out of memory for assembler lines

The optimizer has run out of memory to store assembler lines, e.g. #asm lines from the C source, or C source comment lines. Reduce the size of the function.

out of near memory

The compiler has run out of near memory. This is probably due to too many symbol names. Try splitting the program up, or reducing the number of unused symbols in header files etc.

out of space in macro * arg expansion

A macro argument has exceeded the length of an internal buffer. This buffer is normally 4096 bytes long.

output file cannot be also an input file

The linker has detected an attempt to write its output file over one of its input files. This cannot be done, because it needs to simultaneously read and write input and output files.

page width must be >= 41

The listing page widthe must be at least 41 characters. Any less will not allow a properly formatted listing to be produced.

phase error
The assembler has calculated a different value for a symbol on two different passes. This is probably due to bizarre use of macros or conditional assembly.

phase error in macro args
The assembler has detected a difference in the definition of a symbol on the first and a subsequent pass.

phase error on temporary label
The assembler has detected a difference in the definition of a symbol on the first and a subsequent pass.

pointer required
A pointer is required here. This often means you have used "->" with a structure rather than a structure pointer.

pointer to * argument required
A pointer argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments.

pointer to non-static object returned
This function returns a pointer to a non-static (e.g. automatic) variable. This is likely to be an error, since the storage associated with automatic variables becomes invalid when the function returns.

portion of expression has no effect
Part of this expression has no side effects, and no effect on the value of the expression.

possible pointer truncation
A pointer qualifed "far" has been assigned to a default pointer or a pointer qualified "near", or a default pointer has been assigned to a pointer qualified "near". This may result in truncation of the pointer and loss of information, depending on the memory model in use.

preprocessor assertion failure
The argument to a preprocessor #assert directive has evaluated to zero. This is a programmer induced error.

probable missing '}' in previous block
The compiler has encountered what looks like a function or other declaration, but the preceding function has not been ended with a closing brace. This probably means that a closing brace has been omitted from somewhere in the previous function, although it may well not be the last one.

psect '*' re-orged
This psect has its address multiply defined. Check the linker options.

psect * cannot be in classes *
A psect cannot be in more than one class. This is either due to assembler modules with conflicting class= options, or use of the -C option to the linker.

psect * cannot be in classes * and *
This psect has been declared as being in two different classes, which is not allowed.

psect * in more than one group
This psect has been defined to be in more than one group.

psect * not loaded on * boundary

This psect has a relocatability requirement that is not met by the load address given in a -P option. For example if a psect must be on a 4K byte boundary, you could not start it at 100H.

psect * not relocated on * boundary

This psect is not relocated on the required boundary. Check the relocatability of the psect and correct the -p option. if necessary.

psect * not specified in -p option

This psect was not specified in a "-P" option to the linker. It has been linked at the end of the program, which is probably not where you wanted it.

psect * not specified in -p option (first appears in *

This psect is missing from any -p or -A option. It will be linked at a default address, which is almost certainly not what you want.

psect * re-orged

This psect has had its start address specified more than once.

psect * selector value redefined

The selector value for this psect has been defined more than once.

psect * type redefined: *

This psect has had its type defined differently by different modules. This probably means you are trying to link imcompatible object modules, e.g. linking 386 flat model code with 8086 real mode code.

psect exceeds max size: *

The psect has more bytes in it than the maximum allowed.

psect is absolute: *

This psect is absolute and should not have an address specified in a -P option.

psect may not be local and global

A psect may not be declared to be local if it has already been declared to be (default) global.

psect origin multiply defined: *

The origin of this psect is defined more than once.

psect property redefined

A property of a psect has been defined in more than place to be different.

psect reloc redefined

The relocatability of this psect has been defined differently in two or more places.

psect selector redefined

The selector associated with this psect has been defined differently in two or more places.

psect size redefined

The maximum size of this psect has been defined differently in two or more places.

qualifiers redeclared

This function has different qualifiers in different declarations.

read error on *

The linker encountered an error trying to read this file.

record too long
This indicates that the object file is not a valid HI-TECH object file.

record too long: *
An object file contained a record with an illegal size. This probably means the file is corrupted or not an object file.

recursive function calls:
These functions (or function) call each other recursively. One or more of these functions has statically allocated local variables (compiled stack). Either use the reentrant keyword (if supported with this compiler) or recode to avoid recursion.

recursive macro definition of *
The named macro has been defined in such a manner that expanding it causes a recursive expansion of itself!

redefining macro *
The macro specified is being redefined, to something different to the original definition. If you want to deliberately redefine a macro, use #undef first to remove the original definition.

redundant & applied to array
The address operator "&" has been applied to an array. Since using the name of an array gives its address anyway, this is unnecessary and has been ignored.

refc == 0
Internal error - Contact HI-TECH.

regused - bad arg to g
Internal error - Contact HI-TECH.

relocation error
It is not possible to add together two relocatable quantities. A constant may be added to a relocatable value, and two relocatable addresses in the same psect may be subtracted. An absolute value must be used in various places where the assembler must know a value at assembly time.

relocation offset * out of range *
An object file contained a relocation record with a relocation offset outside the range of the preceding text record. This means the object file is probably corrupted.

relocation too complex
The complex relocation in this expression is too big to be inserted into the object file.

remsym error
Internal error. Contact HI-TECH technical support.

rept argument must be >= 0
The argument to a "REPT" directive must be greater than zero.

seek error: *
The linker could not seek when writing an output file.

segment * overlaps segment *
The named segments have overlapping code or data. Check the addresses being assigned by the "-P" option.

signatures do not match: *

The specified function has different signatures in different modules. This means it has been declared differently, e.g. it may have been prototyped in one module and not another. Check what declarations for the function are visible in the two modules specified and make sure they are compatible.

signed bitfields not supported

Only unsigned bitfields are supported. If a bitfield is declared to be type “int”, the compiler still treats it as unsigned.

simple integer expression required

A simple integral expression is required after the operator “@”, used to associate an absolute address with a variable.

simple type required for *

A simple type (i.e. not an array or structure)is required as an operand to this operator.

sizeof external array * is zero

The sizeof an external array evaluates to zero. This is probably due to the array not having an explicit dimension in the extern declaration.

sizeof yields 0

The code generator has taken the size of an object and found it to be zero. This almost certainly indicates an error in your declaration of a pointer, e.g. you may have declared a pointer to a zero length array. In general, pointers to arrays are of little use. If you require a pointer to an array of objects of unknown length, you only need a pointer to a single object that can then be indexed or incremented.

storage class illegal

A structure or union member may not be given a storage class. Its storage class is determined by the storage class of the structure.

storage class redeclared

A variable or function has been re-declared with a different storage class. This can occur where there are two conflicting declarations, or where an implicit declaration is followed by an actual declaration.

strange character * after ##

A character has been seen after the token catenation operator ## that is neither a letter nor a digit. Since the result of this operator must be a legal token, the operands must be tokens containing only letters and digits.

strange character after # *

There is an unexpected character after #.

string expected

The operand to an “asm” statement must be a string enclosed in parentheses.

string too long

This string is too long. Shorten it.

struct/union member expected

A structure or union member name must follow a dot (“.”) or arrow (“->”).

struct/union redefined: *

A structure or union has been defined more than once.

struct/union required
A structure or union identifier is required before a dot (“.”).

struct/union tag or ‘{‘ expected
An identifier denoting a structure or union or an opening brace must follow a “struct” or “union” keyword.

symbol * cannot be global
There is an error in an object file, where a local symbol has been declared global. This is either an invalid object file, or an internal error in the linker. Try recreating the object file.

symbol * has erroneous psect: *
There is an error in an object file, where a symbol has an invalid psect. This is either an invalid object file, or an internal error in the linker. Try recreating the object file.

symbol * not defined in #undef
The symbol supplied as argument to #undef was not already defined. This is a warning only, but could be avoided by including the #undef in a #ifdef ... #endif block.

syntax error
A syntax error has been detexted. This could be caused a number of things.

syntax error in -a spec
The -A spec is invalid. A valid -A spec should be something like:

```
-AROM=1000h-1FFFh
```

syntax error in checksum list
There is a syntax error in a checksum list read by the linker. The checksum list is read from standard input by the linker, in response to an option. Re-read the manual on checksum listt.

syntax error in local argument
There is a syntax error in a local argument.

text does not start at 0
Code in some things must start at zero. Here it doesn’t.

text offset too low
You aren’t likely to see this error. Rhubarb!

text record has bad length: *
There is an error in an object file. This is either an invalid object file, or an internal error in the linker. Try recreating the object file.

text record has length too small: *
This indicates that the object file is not a valid HI-TECH object file.

this function too large - try reducing level of optimization
A large function has been encountered when using a -Og (global optimization) switch. Try re-compiling without the global optimization, or reduce the size of the function.

this is a struct
This identifier following a “union” or “enum” keyword is already the tag for a structure, and thus should only follow the keyword “struct”.

this is a union
This identifier following a “struct” or “enum” keyword is already the tag for a union, and thus should only follow the keyword “union”.

this is an enum
This identifier following a “struct” or “union” keyword is already the tag for an enumerated type, and thus should only follow the keyword “enum”.

too few arguments
This function requires more arguments than are provided in this call.

too few arguments for format string
There are too few arguments for this format string. This would result in a garbage value being printed or converted at run time.

too many (*) enumeration constants
There are too many enumeration constants in an enumerated type. The maximum number of enumerated constants allowed in an enumerated type is 512.

too many (*) structure members
There are too many members in a structure or union. The maximum number of members allowed in one structure or union is 512.

too many arguments
This function does not accept as many arguments as there are here.

too many arguments for format string
There are too many arguments for this format string. This is harmless, but may represent an incorrect format string.

too many arguments for macro
A macro may only have up to 31 parameters, as per the C Standard.

too many arguments in macro expansion
There were too many arguments supplied in a macro invocation. The maximum number allowed is 31.

too many cases in switch
There are too many case labels in this switch statement. The maximum allowable number of case labels in any one switch statement is 511.

too many comment lines - discarding
The compiler is generating assembler code with embedded comments, but this function is so large that an excessive number of source line comments are being generated. This has been suppressed so that the optimizer will not run out of memory loading comment lines.

too many errors
There were so many errors that the compiler has given up. Correct the first few errors and many of the later ones will probably go away.

too many file arguments. usage: cpp [input [output]]
CPP should be invoked with at most two file arguments.

too many include directories
A maximum of 7 directories may be specified for the preprocessor to search for include files.

too many initializers
There are too many initializers for this object. Check the number of initializers against the object definition (array or structure).

too many macro parameters
There are too many macro parameters on this macro definition.

too many nested #* statements
#if, #ifdef etc. blocks may only be nested to a maximum of 32.

too many nested #if statements
#if, #ifdef etc. blocks may only be nested to a maximum of 32.

too many psect class specifications
There are too many psect class specifications (-C options)

too many psect pragmas
Too many “pragma psect” directives have been used.

too many psects
There are too many psects! Boy, what a program!

too many qualifier names
There are too many qualifier names specified.

too many relocation items
Objtohex filled up a table. This program is just way too complex!

too many segment fixups
There are too many segment fixups in the object file given to objtohex.

too many segments
There are too many segments in the object file given to objtohex.

too many symbols
There are too many symbols for the assemblers symbol table. Reduce the number of symbols in your program. If it is the linker producing this error, suggest changing some global to local symbols.

too many symbols in *
There are too many symbols in the specified function. Reduce the size of the function.

too many temporary labels
There are too many temporary labels in this assembler file. The assembler allows a maximum of 2000 temporary labels.

too much indirection
A pointer declaration may only have 16 levels of indirection.

too much pushback
This error should not occur, and represents an internal error in the preprocessor.

type conflict
The operands of this operator are of incompatible types.

type modifier already specified
This type modifier has already be specified in this type.

type modifiers not valid with this format
Type modifiers may not be used with this format.

type redeclared
The type of this function or object has been redeclared. This can occur because of two incompatible declarations, or because an implicit declaration is followed by an incompatible declaration.

type specifier reqd. for proto arg
A type specifier is required for a prototyped argument. It is not acceptable to just have an identifier.

unbalanced paren's, op is *
The evaluation fo a #if expression found mismatched parentheses. Check the expression for correct parenthesisation.

undefined enum tag: *
This enum tag has not been defined.

undefined identifier: *
This symbol has been used in the program, but has not been defined or declared. Check for spelling errors.

undefined shift * bits)
An attempt has been made to shift a value by a number of bits equal to or greater than the number of bits in the data type, e.g. shifting a long by 32 bits. This will produce an undefined result on many processors. This is non-portable code and is flagged as having undefined results by the C Standard.

undefined struct/union
This structure or union tag is undefined. Check spelling etc.

undefined struct/union: *
The specified structure or union tag is undefined. Check spelling etc.

undefined symbol *
The named symbol is not defined, and has not been specified "GLOBAL".

undefined symbol * in #if, 0 used
A symbol on a #if expression was not a defined preprocessor macro. For the purposes of this expression, its value has been taken as zero.

undefined symbol:
The symbol following is undefined at link time. This could be due to spelling error, or failure to link an appropriate module.

undefined symbols:
A list of symbols follows that were undefined at link time.

undefined temporary label
A temporary label has been referenced that is not defined. Note that a temporary label must have a number >= 0.

undefined variable: *
This variable has been used but not defined at this point

unexpected \ in #if
The backslash is incorrect in the #if statement.

unexpected end of file
This probably means an object file has been truncated because of a lack of disk (or RAMdisk) space.

unexpected eof
An end-of-file was encountered unexpectedly. Check syntax.

unexpected text in #control line ignored
This warning occurs when extra characters appear on the end of a control line, e.g.

```
#endif something
```

The “something” will be ignored, but a warning is issued. It is preferable (and in accordance with Standard C) to enclose the “something” as a comment, e.g.

```
#endif   /* something */
```

unknown complex operator *
There is an error in an object file. This is either an invalid object file, or an internal error in the linker. Try recreating the object file.

unknown fnrec type *
This indicates that the object file is not a valid HI-TECH object file.

unknown option *
This option to the preprocessor is not recognized.

unknown pragma *
An unknow pragma directive was encountered.

unknown predicate *
Internal error - Contact HI-TECH.

unknown psect
The assembler file read by the optimizer has an unknown psect.

unknown psect: *
This psect has been listed in a -P option, but is not defined in any module within the program.

unknown qualifier * given to -a
The -a option to P1 should have as its argument a defined qualifier name,

unknown record type: *
An invalid object module has been read by the linker. It is either corrupted or not an object file.

unknown symbol type *
The symbol type encountered is unknown to this linker. Check that the correct linker is being used.

unreachable code
This section of code will never be executed, because there is no execution path by which it could be reached. Look for missing “break” statements inside a control structure like “while” or “for”.

unreasonable matching depth
This is an internal code generator error. Contact HI-TECH technical support with full details of the code that caused this error.

unrecognized option to -z: *
The code generator has been passed a -Z option it does not understand. This should not happen if it is invoked with the standard driver.

unrecognized qualifer name after ‘strings’
The pragma ‘strings’ requires a list of valid qualifer names. For example

```
#pragma  strings const code
```

would add const and code to the current string qualifiers. If no qualifiers are specified, all qualification will be removed from subsequent strings. The qualifer names must be recognized by the compiler.

unterminated #if[n][def] block from line *
A #if or similar block was not terminated with a matching #endif. The line number is the line on which the #if block began.

unterminated macro arg
An argument to a macro is not terminated. Note that angle brackets (“<>”) are used to quote macro arguments.

unterminated string
A string constant appears not to have a closing quote missing.

unterminated string in macro body
A macro definition contains a string that lacks a closing quote.

unused constant: *
This enumerated constant is never used. Maybe it isn’t needed at all.

unused enum: *
This enumerated type is never used. Maybe it isn’t needed at all.

unused label: *
This label is never used. Maybe it isn’t needed at all.

unused member: *
This structure member is never used. Maybe it isn’t needed at all.

unused structure: *
This structure tag is never used. Maybe it isn’t needed at all.

unused typedef: *
This typedef is never used. Maybe it isn’t needed at all.

unused union: *
This union type is never used. Maybe it isn’t needed at all.

unused variable declaration: *
This variable is never used. Maybe it isn’t needed at all.

unused variable definition: *
This variable is never used. Maybe it isn’t needed at all.

variable may be used before set: *
This variable may be used before it has been assigned a value. Since it is an auto variable, this will result in it having a random value.

void function cannot return value
A void function cannot return a value. Any “return” statement should not be followed by an expression.

while expected
The keyword “while” is expected at the end of a “do” statement.

work buffer overflow *
An internal preprocessor buffer has been filled. This buffer has a size of 4096 bytes.

write error (out of disk space?) *
Probably means that the hard disk or RAM disk is full.

write error on *
A write error occured on the named file. This probably means you have run out of disk space.

write error on object file
An error was reported when the assembler was attempting to write an object file. This probably means there is not enough disk space.

wrong number of macro arguments for * - * instead of *
A macro has been invoked with the wrong number of arguments.

{ expected
An opening brace was expected here.

} expected
A closing brace was expected here.

Appendix B

Library Functions

The functions within the Pacific C library are listed in this appendix. Each entry begins with the name of the function, followed by information under the following headings:

SYNOPSIS

Under this heading is given the C definition of the function, and the header file in which it is declared.

DESCRIPTION

This is a narrative description of the function and its purpose.

EXAMPLE

An example is given here of the use of the function. This usually consists of a complete small program that illutstrates the function.

DATA TYPES

If any special data types (structures etc.) are defined for use with the function, they are listed here with their C definition. These data types will be defined in the header file given under SYNOPSIS.

RETURN VALUE

The type and nature of the return value of the function (if any) is given. Information on error returns is also included.

SEE ALSO

Where there are related functions that may be of interest in connection with the current function, they are listed here.

NOTE

Sometimes special information will be provided under this heading.

ACOS

SYNOPSIS

```
#include <math.h>
double   acos(double f)
```

DESCRIPTION

Acos() implements the converse of *cos()*, i.e. it is passed a value in the range -1 to +1, and returns an angle in radians whose cosine is equal to that value.

EXAMPLE

```
#include        <math.h>
#include        <stdio.h>

/* print acos() values for -1 to 1 in degrees */

main()
{
        float   i, a;

        for(i = -1.0; i < 1.0 ; i += 0.1) {
                a = acos(i)*180.0/3.141592;
                printf("acos(%f) = %f degrees\n", i, a);
        }
}
```

RETURN VALUE

An angle in radians, in the range 0 to 2π. Where the argument value is outside the domain -1 to 1, the return value will be zero.

SEE ALSO

sin, cos, tan, acos, atan, atan2

ASCTIME

SYNOPSIS

```
#include <time.h>
char *   asctime(struct tm * t)
```

DESCRIPTION

Asctime() takes the broken down time pointed to by its argument, and returns a 26 character string describing the current date and time in the format

```
Sun Sep 16 01:03:52 1973\n\0
```

Note the newline at the end of the string. The width of each field in the string is fixed. The example gets the current time, converts it to a struct tm pointer with localtime, then converts this to ASCII and prints it.

EXAMPLE

```
#include        <stdio.h>
#include        <time.h>

main()
{
        time_t  clock;
        struct tm *     tp;

        time(&clock);
        tp = localtime(&clock);
        printf("%s", asctime(tp));
}
```

DATA TYPES

```
struct tm {       int    tm_sec;
      int    tm_min;
      int    tm_hour;
      int    tm_mday;
      int    tm_mon;
      int    tm_year;
      int    tm_wday;
      int    tm_yday;
```

```
    int     tm_isdst;
};
```

SEE ALSO

ctime, time, gmtime, localtime, time

ASIN

SYNOPSIS

```
#include  <math.h>
double    asin(double f)
```

DESCRIPTION

Asin() implements the converse of *sin()*, i.e. it is passed a value in the range -1 to +1, and returns an angle in radians whose sine is equal to that value.

EXAMPLE

```
#include        <math.h>
#include        <stdio.h>

main()
{
        float   i, a;

        for(i = -1.0; i < 1.0 ; i += 0.1) {
                a = asin(i)*180.0/3.141592;
                printf("asin(%f) = %f degrees\n", i, a);
        }
}
```

RETURN VALUE

An angle in radians, in the range $-\pi/2$ to $+\pi/2$. Where the argument value is outside the domain -1 to 1, the return value will be zero.

SEE ALSO

sin, cos, tan

ASSERT

SYNOPSIS

```
#include <assert.h>
void     assert(int e)
```

DESCRIPTION

This macro is used for debugging purposes; the basic method of usage is to place assertions liberally throughout your code at points where correct operation of the code depends upon certain conditions being true initially. An *assert()* may be used to ensure at run time that that assumption holds. For example, the following statement asserts that the pointer **tp** is non-null:

```
assert(tp);
```

If at run time the expression evaluates to false, the program will abort with a message identifying the source file and line number of the assertion, and the expression used as an argument to it. A fuller discussion of the uses of *assert* is impossible in limited space, but it is closely linked to methods of proving program correctness.

EXAMPLE

```
ptrfunc(struct xyz * tp)
{
        assert(tp != 0);
}
```

ATAN

SYNOPSIS

```
#include <math.h>
double atan(double x);
```

DESCRIPTION

This function returns the arc tangent of its argument, i.e. it returns an angle **e**in the range $\pi/2$ to $-pi/2$ such that $sin(e) \equiv x$.

SEE ALSO

tan, asin, acos, atan2

ATEXIT

SYNOPSIS

```
#include <stdlib.h>
int      atexit(void (*func)(void));
```

DESCRIPTION

The *atexit()* function registers the function pointed to by **func**, to be called without arguments at normal program termination. On program termination, all functions registered by *atexit()* are called, in the reverse order of their registration.

EXAMPLE

```
#include        <stdio.h>
#include        <stdlib.h>

char *  fname;
FILE *  fp;

void
rmfile(void)
{
        if(fp)
                fclose(fp);
        if(fname)
                remove(fname);
}

main()
{        /* create a file; on exit, close and remove it */

        if(!(fp = fopen((fname = “test.fil”), “w”)))
                exit(1);
        atexit(rmfile);
}
```

RETURN VALUE

Atexit() returns zero if the registration succeeds, nonzero if it fails.

SEE ALSO

exit

ATOF

SYNOPSIS

```
#include <stdlib.h>
double   atof(char * s)
```

DESCRIPTION

Atof scans the character string passed to it, skipping leading blanks, and converts an ASCII representation of a number to a double. The number may be in decimal, normal floating point or scientific notation.

EXAMPLE

```
#include        <stdlib.h>
#include        <stdio.h>

main()
{
        char    buf[80];
        double  i;

        gets(buf);
        i = atof(buf);
        printf("Read %s: converted to %f\n", buf, i);
}
```

RETURN VALUE

A double precision floating point number. If no number is found in the string, 0.0 will be returned.

SEE ALSO

atoi, atol

ATOI

SYNOPSIS

```
#include <math.h>
double   atoi(char * s)
```

DESCRIPTION

Atoi scans the character string passed to it, skipping leading blanks, and converts an ASCII representation of a decimal number to an integer.

EXAMPLE

```
#include        <stdlib.h>
#include        <stdio.h>

main()
{
        char    buf[80];
        int  i;

        gets(buf);
        i = atoi(buf);
        printf("Read %s: converted to %d\n", buf, i);
}
```

RETURN VALUE

A signed integer. If no number is found in the string, 0 will be returned.

SEE ALSO

atof, atol

ATOL

SYNOPSIS

```
#include  <math.h>
double    atol(char * s)
```

DESCRIPTION

Atoi scans the character string passed to it, skipping leading blanks, and converts an ASCII representation of a decimal number to a long integer.

EXAMPLE

```
#include        <stdlib.h>
#include        <stdio.h>

main()
{
        char    buf[80];
        long  i;

        gets(buf);
        i = atol(buf);
        printf("Read %s: converted to %ld\n", buf, i);
}
```

RETURN VALUE

A long integer. If no number is found in the string, 0 will be returned.

SEE ALSO

atoi, atof

BSEARCH

SYNOPSIS

```
#include <stdlib.h>
void * bsearch(const void * key, const void * base, size_t n_memb,
            size_t size, int (*compar)(const void *, const void
*));
```

DESCRIPTION

The *bsearch()* function searches a sorted array for an element matching a particular key. It uses a binary search algorithm, calling the function pointed to by **compar** to compare elements in the array.

EXAMPLE

```
/* sample bsearch program */

#include    <stdlib.h>
#include    <stdio.h>
#include    <string.h>

struct value {
    char    name[40];
    int value;
}   values[100];

int
val_cmp(const void * p1, const void * p2)
{
    return strcmp(((const struct value *)p1)-name,
        ((const struct value *)p2)-name);
}

main()
{
    char    inbuf[80];
    int i;
    struct value *  vp;

    i = 0;
    while(gets(inbuf)) {
        sscanf(inbuf,"%s %d", values[i].name, &values[i].value);
        i++;
    }
    qsort(values, i, sizeof values[0], val_cmp);
```

```
    vp = bsearch("fred", values, i, sizeof values[0],
        val_cmp);
    if(!vp)
        printf("Item 'fred' was not found\n");
    else
        printf("Item 'fred' has value %d\n", vp-value);
}
```

RETURN VALUE

A pointer to the matched array element (if there is more than one matching element, any of these may be returned). If no match is found, a null pointer is returned.

NOTE

The comparison function must have the correct prototype.

SEE ALSO

qsort

CALLOC

SYNOPSIS

```
#include <stdlib.h>
void * calloc(size_t cnt, size_t size)
```

DESCRIPTION

Calloc() attempts to obtain a contiguous block of dynamic memory which will hold **cnt** objects, each of length **size**. The block is filled with zeros.

EXAMPLE

```
#include        <stdlib.h>
#include        <stdio.h>

struct test {
        int     a[20];
} *     ptr;

main()
{       /* allocate space for 20 structs */

        ptr = calloc(20, sizeof(struct test));
        if(!ptr)
                printf("Failed\n");
        else
                free(ptr);
}
```

RETURN VALUE

A pointer to the block is returned, or 0 if the memory could not be allocated.

SEE ALSO

brk, sbrk, malloc, free

CEIL

SYNOPSIS

```
#include <math.h>
double   ceil(double f)
```

DESCRIPTION

This routine returns the smallest whole number not less than **f**.

CGETS

SYNOPSIS

```
#include <conio.h>
char *   cgets(char * s)
```

DESCRIPTION

Cgets() will read one line of input from the console into the buffer passed as an argument. It does so by repeated calls to *getche()*. As characters are read, they are buffered, with BACKSPACE deleting the previously typed character, and ctrl-U deleting the entire line typed so far. Other characters are placed in the buffer, with a carriage return or line feed (newline) terminating the function. The collected string is null terminated.

EXAMPLE

```
#include        <conio.h>
#include        <string.h>

char    buffer[80];

main()
{
        for(;;) {
                cgets(buffer);
                if(strcmp(buffer, “exit”) == 0)
                        break;
                cputs(“Type ‘exit’ to finish\n”);
        }
}
```

RETURN VALUE

The return value is the character pointer passed as the sole argument.

SEE ALSO

getch, getche, putch, cputs

CHDIR

SYNOPSIS

```
#include <sys.h>
int      chdir(char * s)
```

DESCRIPTION

This function is availble only under MS-DOS. It changes the current working directory to the path name supplied as argument. This path name be be absolute, as in C:\FRED, or relative, as in ..\SOURCES.

EXAMPLE

```
#include        <sys.h>

main()
{
        chdir("C:\\");
}
```

RETURN VALUE

A return value of -1 indicates that the requested change could not be performed. This usually indicates the directory or some component of the path name does not exist. On success the return value will be zero.

NOTE

Under MS-DOS, there is a "current directory" for each drive. If there is a drive specification in the path, this function will change the current directory for that drive, but will NOT change the current drive.

SEE ALSO

mkdir, rmdir, getcwd

CHDRV

SYNOPSIS

```
#include <sys.h>
int chdrv(char * drvname);
```

DESCRIPTION

To set the current DOS drive, invoke this function with its argument pointing to a string where the first character is a drive letter.

EXAMPLE

```
#include    <sys.h>
main()
{
    chdrv(“C:”);
}
```

RETURN VALUE

If the selected drive is invalid, a value of -1 is returned.

SEE ALSO

chmod

CHMOD

SYNOPSIS

```
#include <stat.h> #include       <unixio.h>

int      chmod(char * name, int mode)
```

DESCRIPTION

This function changes the file attributes (or modes) of the named file. The argument **name** may be any valid file name. The **mode** argument may include all bits defined in *stat.h* except those relating to the type of the file, e.g. S_IFDIR.

EXAMPLE

```
#include        <stat.h>
#include        <stdio.h>
#include        <unixio.h>

/* make a file read-only */

main(argc, argv)
char ** argv;
{
        if(argc > 1)
                if(chmod(argv[1], S_IREAD) < 0)
                        perror(argv[1]);
}
```

RETURN VALUE

Zero is returned on success, -1 on failure.

NOTE

Not all bits may be changed under all operating systems, e.g. neither DOS nor CP/M permit a file to be made unreadable, thus even if **mode** does not include S_IREAD the file will still be readable (and *stat()* will still return S_IREAD in flags).

SEE ALSO

stat, creat

CLOSE

SYNOPSIS

```
#include <unixio.h>
int close(int fd)
```

DESCRIPTION

This routine closes the file associated with the file descriptor **fd**, which will have been previously obtained from a call to *open()* or *creat()*.

EXAMPLE

```
#include        <unixio.h>
#include        <stdio.h>

/* create an empty file */

main(argc, argv)
char ** argv;
{
        int     fd;

        if(argc > 1) {
                if((fd = creat(argv[1], 0600)) < 0)
                        perror(argv[1]);
                else
                        close(fd);

        }
}
```

RETURN VALUE

Close() returns 0 for a successful close, or -1 otherwise.

SEE ALSO

open, read, write, seek

CLRERR, CLREOF

SYNOPSIS

```
#include <stdio.h> void clrerr(FILE * stream)
void clreof(FILE * stream)
```

DESCRIPTION

These are macros, defined in stdio.h, which reset the error and end of file flags respectively for the specified stream. They should be used with care; the major valid use is for clearing an EOF status on input from a terminal-like device, where it may be valid to continue to read after having seen an end-of-file indication. If a *clreof()* is not done, then repeated reads will continue to return EOF.

EXAMPLE

```
#include        <stdio.h>
#include        <string.h>

main()
{
        char    buf[80];

        for(;;) {
                if(!gets(buf)) {
                        printf("EOF seen\n");
                        clreof(stdin);
                } else if(strcmp(buf, "quit") == 0)
                        break;
        }
}
```

SEE ALSO

fopen, fclose

COS

SYNOPSIS

```
#include <math.h>
double   cos(double f)
```

DESCRIPTION

This function yields the cosine of its argument. The cosine is calculated by expansion of a polynomial series approximation.

EXAMPLE

```
#include        <math.h>
#include        <stdio.h>

#define C       3.141592/180.0
main()
{
        double  i;

        for(i = 0 ; i <= 180.0 ; i += 10)
                printf("sin(%3.0f) = %f, cos = %f\n",
                        i, sin(i*C), cos(i*C));
}
```

RETURN VALUE

A double in the range -1 to +1.

SEE ALSO

sin, tan, asin, acos, atan

COSH, SINH, TANH

SYNOPSIS

```
#include <math.h>
double   cosh(double f)
double   sinh(double f)
double   tanh(double f)
```

DESCRIPTION

These functions implement the hyperbolic trig functions.

CPUTS

SYNOPSIS

```
#include <conio.h>
void     cputs(const char * s)
```

DESCRIPTION

Cputs() writes its argument string to the console, outputting carriage returns before each newline in the string. It calls *putch()* repeatedly.On a hosted system *cputs()* differs from *puts()* in that it reads the console directly, rather than using file I/O. In an embedded system *cputs* and *puts* are equivalent.

EXAMPLE

```
#include        <conio.h>
#include        <string.h>

char    buffer[80];

main()
{
        for(;;) {
                cgets(buffer);
                if(strcmp(buffer, “exit”) == 0)
                        break;
                cputs(“Type ‘exit’ to finish\n”);
        }
}
```

SEE ALSO

cputs, puts, putch

CREAT

SYNOPSIS

```
#include <stat.h>
int      creat(char * name, int mode)
```

DESCRIPTION

This routine attempts to create the file named by **name**. If the file exists and is writeable, it will be removed and re-created. The return value is -1 if the create failed, or a small non-negative number if it succeeded. This number is a valuable token which must be used to write to or close the file subsequently. **Mode** is used to initialize the attributes of the created file. The allowable bits are the same as for *chmod()*, but for Unix compatibility it is recommended that a mode of 0666 or 0600 be used. Under CP/M the mode is ignored - the only way to set a file's attributes is via the *chmod()* function.

EXAMPLE

```
#include        <unixio.h>
#include        <stdio.h>

main(argc, argv)
char ** argv;
{
        int     fd;

        if(argc > 1) {
                if((fd = creat(argv[1], 0600)) < 0)
                        perror(argv[1]);
                else
                        close(fd);

        }
}
```

SEE ALSO

open, close, read, write, seek, stat, chmod

CTIME

SYNOPSIS

```
#include <time.h>
char *   ctime(time_t t)
```

DESCRIPTION

Ctime() converts the time in seconds pointed to by its argument to a string of the same form as described for asctime. Thus the example program prints the current time and date:

EXAMPLE

```
#include        <stdio.h>
#include        <time.h>

main()
{
        time_t  clock;

        time(&clock);
        printf("%s", ctime(&clock));
}
```

DATA TYPES

typedef long time_t;

SEE ALSO

gmtime, localtime, asctime, time

DI, EI

SYNOPSIS

```
#include <intrpt.h>
void ei(void);
void di(void);
```

DESCRIPTION

Ei() and *di()* enable and disable interrupts respectively. These are implemented as macros defined in *intrpt.h*. On most processors they will expand to an in-line assembler instruction that sets or clears the interrupt enable or mask bit. The example shows the use of *ei()* and *di* around access to a long variable that is modified during an interrupt. If this was not done, it would be possible to return an incorrect value, if the interrupt occurred between accesses to successive words of the count value.

EXAMPLE

```
#include        <intrpt.h>

long    count;

void interrupt
tick(void)
{
        count++;
}

long
getticks(void)
{
        long    val;

        /* disable interrupts around access
           to count, to ensure consistency */

        di();
        val = count;
        ei();
        return val;
}
```

DIV

SYNOPSIS

```
#include <stdlib.h>
div_t div(int numer, int demon);
```

DESCRIPTION

The *div()* function computes the quotient and remainder of the numerator divided by the denominator.

EXAMPLE

```
#include    <stdlib.h>
#include    <stdio.h>
main()
{
    div_t   x;

    x = div(12345, 66);
    printf("quotient = %d, remainder = %d\n", x.quot, x.rem);
}
```

DATA TYPES

```
typedef struct {
    int   quot;
    int   rem;
}  div_t;
```

DRIVER

SYNOPSIS

```
#include <dos.h>
int     driverx(far void * entry, union REGS * regs, struct SREGS
sregs);
int     driver(far void * entry, union REGS * regs);
```

DESCRIPTION

These routines allow entry points into external code to be called, with registers set up to specific values. They are used for accessing DOS features where a far call, rather than an interrupt, is used, for example the XMS routines.

DATA TYPES

```
struct WORDREGS {       unsigned int ax;
      unsigned int bx;
      unsigned int cx;
      unsigned int dx;
      unsigned int si;
      unsigned int di;
      unsigned int cflag;
      unsigned int psw;
};

struct BYTEREGS {
      unsigned char al, ah;
      unsigned char bl, bh;
      unsigned char cl, ch;
      unsigned char dl, dh;
};

union REGS {
      struct WORDREGS x;
      struct BYTEREGS h;
};

struct SREGS {
      unsigned int es;
      unsigned int cs;
      unsigned int ss;
```

```
    unsigned int ds;
};
```

SEE ALSO

int86, int86x, intdos, intdosx

DUP

SYNOPSIS

```
#include <unixio.h>
int dup(int fd)
```

DESCRIPTION

Given a file descriptor, such as returned by *open()*, this routine will return another file descriptor which will refer to the same open file.

EXAMPLE

```
#include        <stdio.h>
#include        <unixio.h>
#include        <stdlib.h>
#include        <sys.h>

main(argc, argv)
char ** argv;
{
        FILE *  fp;

        if(argc < 3) {
                fprintf(stderr, “Usage: fd2 stderr_file command ...\n”);
                exit(1);
        }
        if(!(fp = fopen(argv[1], “w”))) {
                perror(argv[1]);
                exit(1);
        }
        close(2);
        dup(fileno(fp));        /* make stderr reference file */
        fclose(fp);
        spawnvp(argv[2], argv+2);
        close(2);
}
```

RETURN VALUE

-1 is returned if the **fd** argument is a bad descriptor or does not refer to an open file.

SEE ALSO

open, close, creat, read, write

DUP2

SYNOPSIS

```
#include <unixio.h>
int dup2(int original, int new)
```

DESCRIPTION

Dup2() duplicates a file handle, so that the open file may be referred to by the new handle as well as the old handle. Unlike *dup()* this function allows the new file handle to be specified, rather than the next free handle being allocated. If the new handle **new** is already in use before the *dup2()* call, it will be closed first.

EXAMPLE

```
#include    <stdio.h>
#include    <unixio.h>
#include    <stdlib.h>
#include    <sys.h>

main(argc, argv)
char ** argv;
{
    FILE *  fp;

    if(argc < 3) {
        fprintf(stderr, “Usage: fd2 stderr_file command ...\n”);
        exit(1);
    }
    if(!(fp = fopen(argv[1], “w”))) {
        perror(argv[1]);
        exit(1);
    }
    dup2(fileno(fp), 2);    /* make stderr reference file */
    fclose(fp);
    spawnvp(argv[2], argv+2);
    close(2);
}
```

RETURN VALUE

If either handle is invalid, or the original handle does not refer to an open file, -1 will be returned.

SEE ALSO

dup

EXIT

SYNOPSIS

```
#include <stdlib.h>
void     exit(int status)
```

DESCRIPTION

This call will close all open files and exit from the program. On CP/M, this means a return to CCP level, under MS-DOS a return to the DOS prompt or the program which spawned the current program. **Status** is a value that is used as the exit value of the program. This is recovered under DOS with the wait for status DOS call. The status value will be stored on CP/M at 80H. In an embedded system *exit()* normally restarts the program as though a hardware reset had occured. This call will never return.

EXAMPLE

```
#include <stdlib.h>

main()
{
       exit(0);
}
```

RETURN VALUE

Never returns.

SEE ALSO

atexit

EXP

SYNOPSIS

```
#include <math.h>
double   exp(double f)
```

DESCRIPTION

Exp() returns the exponential function of its argument, i.e. e sup f .

EXAMPLE

```
#include        <math.h>
#include        <stdio.h>

main()
{
        double  f;

        for(f = 0.0 ; f <= 5 ; f += 1.0)
                printf("e to %1.0f = %f\n", f, exp(f));
}
```

SEE ALSO

log, log10, pow

FABS

SYNOPSIS

```
#include <math.h>
double   fabs(double f)
```

DESCRIPTION

This routine returns the absolute value of its double argument.

SEE ALSO

abs

FARMALLOC, FARFREE, FARREALLOC

SYNOPSIS

```
#include <stdlib.h>
far void * farmalloc(size_t size);
void farfree(far void * vp);
```

DESCRIPTION

In small and medium models, pointers are by default 16 bit near pointers that refer to the DS register. The maximum amount of memory that can be accessed with default pointers is 64K. To allocate more memory, the function *farmalloc()* can be used. It returns a far pointer (32 bits in segment:offset form) to memory allocated at the end of the current program's memory. Memory allocated with *farmalloc()* should be freed with *farfree()* when no longer required.

Farrealloc() is the far equivalent of *realloc()*, i.e. it will alter the size of a block, copying it if necessary, and returning a pointer to the new block.

Memory allocated with these routines can not be used where near memory is required, i.e. it cannot be used as an argument to a library routine which expects a default pointer. There are however far versions of some useful routines in the library.

The maximum size of a single block that can be allocated with *farmalloc()* is 64K, since it must lie within one segment.

EXAMPLE

```
#include    <stdlib.h>
#include    <stdio.h>
main()
{
    far char *      x;
    int             i;

    for(i = 0 ; i != 10 ; i++) {
        if(!(x = farmalloc(32768)))
            break;
        i++;
    }
    printf("allocated %ld Kbytes in 32K blocks\n", i*32768L);
}
```

NOTE

In large and compact models farmalloc and farfree are the same functions as malloc and free. Freed memory is not returned to DOS until the program exits, but will be reallocated by farmalloc.

SEE ALSO

farstrcpy, farmemcpy, malloc, free, realloc

FARMEMCPY, FARMEMSET

SYNOPSIS

```
#include <stdlib.h>
far void * farmemcpy(far void * to, far void * from, size_t
count);
far void * farmemset(far void * buf, int val, size_t count);
```

DESCRIPTION

Farmemcpy() will copy **count** bytes from one far memory address to another. IT is used in small and compact models for copying between memory blocks where either the source or destination block is not in the default data segment.

Similarly *farmemset()* will set a far block of memory to a specified value.

EXAMPLE

```
#include    <string.h>
#include    <stdio.h>
#include    <stdlib.h>

far char    buf[80];
far char    buf2[80];

main()
{
    int i;

    farmemset(buf2, 0x7F, 80);
    for(i = 0 ; i != 80 ; i++)
        buf[i] = i;
    farmemcpy(buf2, buf, 80);
    printf("buf2[10] = %d\n", buf2[10]);
}
```

RETURN VALUE

The destination buffer pointer is returned.

SEE ALSO

farmalloc, memcpy, memset

FCLOSE

SYNOPSIS

```
#include <stdio.h>
int      fclose(FILE * stream)
```

DESCRIPTION

This routine closes the specified i/o stream. **Stream** should be a token returned by a previous call to *fopen()*.

EXAMPLE

```
#include        <stdio.h>

main(argc, argv)
char ** argv;
{
        FILE *  fp;

        if(argc > 1) {
                if(!(fp = fopen(argv[1], “r”)))
                        perror(argv[1]);
                else {
                        fprintf(stderr, “Opened %s\n”, argv[1]);
                        fclose(fp);
                }

        }
}
```

RETURN VALUE

Zero is returned on a successful close, EOF otherwise.

SEE ALSO

fopen, fread, fwrite

FDOPEN

SYNOPSIS

```
#include <stdio.h>
FILE * fdopen(int fd, char * mode);
```

DESCRIPTION

Where it is desired to associate a STDIO stream with a low-level file descriptor that already refers to an open file this function may be used. It will return a pointer to a FILE structure which references the specified low-level file descriptor, as though the function *fopen()* had been called. The **mode** argument is the same as for *fopen()*.

EXAMPLE

```
#include    <stdio.h>

main()
{
    FILE *  fp;

    /* 3 is the device AUX */

    fp = fdopen(3, “w”);
    fprintf(fp, “AUX test line\n”);
}
```

RETURN VALUE

NULL if a FILE structure could not be allocated

SEE ALSO

fopen, freopen, close

FEOF, FERROR

SYNOPSIS

```
#include <stdio.h>
feof(FILE * stream)
ferror(FILE * stream)
```

DESCRIPTION

These macros test the status of the EOF and ERROR bits respectively for the specified stream. Each will be true if the corresponding flag is set. The macros are defined in stdio.h. **Stream** must be a token returned by a previous *fopen()* call.

EXAMPLE

```
#include        <stdio.h>

main()
{
        while(!feof(stdin))
                getchar();
}
```

SEE ALSO

fopen, fclose

FFIRST, FNEXT

SYNOPSIS

```
#include <stat.h>
struct dirbuf *  ffirst(char *);
struct dirbuf *  fnext(void);
```

DESCRIPTION

These two functions provide a means of finding DOS files that match a particular filename pattern. *Ffirst()* is called first, and is passed a filename that may include the special pattern matching characters **?** and *****.

DATA TYPES

```
struct stat {
      unsigned long   st_ino;
      unsigned short  st_dev;
      unsigned short  st_mode;       /* flags */
      long            st_atime;      /* access time */
      long            st_mtime;      /* modification time */
      long            st_size;       /* file size in bytes */
};
struct dirbuf
{
      struct stat     di_stat;              /* info on the file */
      char            di_name[MAXNAMLEN+1];   /* the file name */
};
```

FFLUSH

SYNOPSIS

```
#include <stdio.h>
int fflush(FILE * stream)
```

DESCRIPTION

Fflush() will output to the disk file or other device currently open on the specified **stream** the contents of the associated buffer. This is typically used for flushing buffered standard output in interactive applications. Normally stdout is opened in line buffered mode, and is flushed before any input is done on a stdio stream, but if input is to be done via console I/O routines, it may be necessary to call *fflush()* first.

EXAMPLE

```
#include        <stdio.h>
#include        <conio.h>
main()
{
        printf("press a key: ");
        fflush(stdout);
        getch();
}
```

SEE ALSO

fopen, fclose

FGETC

SYNOPSIS

```
#include <stdio.h>
int      fgetc(FILE * stream)
```

DESCRIPTION

Fgetc() returns the next character from the input stream. If end-of-file is encountered EOF will be returned instead. It is for this reason that the function is declared as int. The integer EOF is not a valid byte, thus end-of-file is distinguishable from reading a byte of all 1 bits from the file. *Fgetc()* is the non-macro version of *getc()*.

RETURN VALUE

A character from the input stream, or EOF on end of file.

SEE ALSO

fopen, fclose, fputc, getc, putc

FGETS

SYNOPSIS

```
#include <stdio.h>
char * fgets(char * s, size_t n, char * stream)
```

DESCRIPTION

Fgets() places in the buffer **s** up to **n**-1 characters from the input **stream**. If a newline is seen in the input before the correct number of characters is read, then *fgets()* will return immediately. The newline will be left in the buffer. The buffer will be null terminated in any case.

EXAMPLE

```
#include        <stdio.h>

main()
{
        char    buffer[128];

        while(fgets(buffer, sizeof buffer, stdin))
                fputs(buffer, stdout);
}
```

RETURN VALUE

A successful *fgets()* will return its first argument; NULL is returned on end-of-file or error.

FILENO

SYNOPSIS

```
#include <stdio.h>
fileno(FILE * stream)
```

DESCRIPTION

Fileno() is a macro from stdio.h which yields the file descriptor associated with **stream**. It is mainly used when it is desired to perform some low-level operation on a file opened as a stdio stream.

EXAMPLE

```
#include        <stdio.h>
#include        <unixio.h>
#include        <stdlib.h>
#include        <sys.h>

main(argc, argv)
char ** argv;
{
        FILE *  fp;

        if(argc < 3) {
                fprintf(stderr, “Usage: fd2 stderr_file command ...\n”);
                exit(1);
        }
        if(!(fp = fopen(argv[1], “w”))) {
                perror(argv[1]);
                exit(1);
        }
        close(2);
        dup(fileno(fp));        /* make stderr reference file */
        fclose(fp);
        spawnvp(argv[2], argv+2);
        close(2);
}
```

SEE ALSO

fopen, fclose, open, close, dup

FLOOR

SYNOPSIS

```
#include <math.h>
double   floor(double f)
```

DESCRIPTION

This routine returns the largest whole number not greater than **f**.

FOPEN

SYNOPSIS

```
#include <stdio.h>
FILE * fopen(char * name, char * mode);
```

DESCRIPTION

Fopen() attempts to open file for reading or writing (or both) according to the **mode** string supplied. The mode string is interpreted as follows:

rThe file is opened for reading if it exists. If the file does not exist the call fails.

r+If the file exists it is opened for reading and writing. If the file does not already exist the call fails.

wThe file is created if it does not exist, or truncated if it does. It is then opened for writing.

w+ The file is created if it does not already exist, or truncated if it does. The file is opened for reading and writing.

a The file is created if it does not already exist, and opened for writing. All writes will be dynamically forced to the end of file, thus this mode is known as *append* mode.

a+ The file is created if it does not already exist, and opened for reading and writing. All writes to the file will be dynamically forced to the end of the file, i.e. while any portion of the file may be read, all writes will take place at the end of the file and will not overwrite any existing data. Calling *fseek()* in an attempt to write at any other place in the file will not be effective.

The “b” modifier may be appended to any of the above modes, e.g. “r+b” or “rb+” are equivalent. Adding the “b” modifier will cause the file to be opened in binary rather than ASCII mode. Opening in ASCII mode ensures that text files are read in a manner compatible with the Unix-derived conventions for C programs, i.e. that text files contain lines delimited by newline characters. The special treatment of read or written characters varies with the operating system, but includes some or all of the following:

NEWLINE (LINE FEED) Converted to carriage return, line feed on output.

RETURN Ignored on input, inserted before NEWLINE on output.

CTRL-Z Signals EOF on input, appended on fclose on output if necessary on CP/M.

Opening a file in binary mode will allow each character to be read just as written, but because the exact size of a file is not known to CP/M, the file may contain more bytes than were written to it. See *open()* for a description of what constitutes a file name.

When using one of the read/write modes (with a '+' character in the string), although they permits reading and writing on the same stream, it is not possible to arbitrarily mix input and output calls to the same stream. At any given time a stream opened with a "+" mode will be in either an input or output state. The state may only be changed when the associated buffer is empty, which is only guaranteed immediately after a call to *fflush()* or one of the file positioning functions *fseek()* or *rewind()*. The buffer will also be empty after encountering EOF while reading a binary stream, It is recommended that an explicit call to *fflush()* be used to ensure this situation. Thus after reading from a stream you should call *fflush()* or *fseek()* before attempting to write on that stream, and vice versa.

EXAMPLE

```
#include        <stdio.h>


main(argc, argv)
char ** argv;
{
        FILE *  fp;

        if(argc > 1) {
                if(!(fp = fopen(argv[1], "r")))
                        perror(argv[1]);
                else {
                        fprintf(stderr, "Opened %s\n", argv[1]);
                        fclose(fp);
                }

        }
}
```

SEE ALSO

fclose, fgetc, fputc, freopen, fread, fflush, fwrite, fseek

FPRINTF

SYNOPSIS

```
#include <stdio.h>
int fprintf(FILE * stream, char * fmt, ...);
```

DESCRIPTION

Fprintf() performs formatted printing on the specified **stream**. Refer to *printf()* for the details of the available formats.

EXAMPLE

```
#include        <stdio.h>

main(argc, argv)
char ** argv;
{
        FILE *  fp;

        if(argc > 1) {
                if(!(fp = fopen(argv[1], “r”)))
                        perror(argv[1]);
                else {
                        fprintf(stderr, “Opened %s\n”, argv[1]);
                        fclose(fp);
                }

        }
}
```

RETURN VALUE

The number of characters written.

SEE ALSO

printf, fscanf, sscanf, sprintf, vfprintf

FPRINTF

SYNOPSIS

```
#include <stdio.h>
fprintf(FILE * stream, char * fmt, ...);
vfprintf(FILE * stream, va_list va_arg);
```

DESCRIPTION

Fprintf() performs formatted printing on the specified **stream**. Refer to *printf()* for the details of the available formats. *Vfprintf()* is similar to *fprintf()* but takes a variable argument list pointer rather than a list of arguments. See the description of *va_start()* for more information on variable argument lists.

SEE ALSO

printf, fscanf, sscanf, sprintf

FPUTC

SYNOPSIS

```
#include <stdio.h>
int      fputc(int c, FILE * stream)
```

DESCRIPTION

The character **c** is written to the supplied **stream**. This is the non-macro version of *putc()*.

RETURN VALUE

The character is returned if it was successfully written, EOF is returned otherwise. Note that “written to the stream” may mean only placing the character in the buffer associated with the stream.

SEE ALSO

putc, fgetc, fopen, fflush

FPUTS

SYNOPSIS

```
#include <stdio.h>
int      fputs(char * s, FILE * stream)
```

DESCRIPTION

The null-terminated string **s** is written to the **stream**. No newline is appended (cf. *puts()*).

EXAMPLE

```
#include        <stdio.h>

main()
{
        fputs("this is a line\n", stdout);
}
```

RETURN VALUE

The return value is 0 for success, EOF for error.

SEE ALSO

puts, fgets, fopen, fclose

FREAD

SYNOPSIS

```
#include <stdio.h>
int      fread(void * buf, size_t size, size_t cnt, FILE * stream)
```

DESCRIPTION

Up to **cnt** objects, each of length **size**, are read into memory at **buf** from the **stream**. No word alignment in the stream is assumed or necessary. The read is done via successive *getc()*'s.

EXAMPLE

```
#include        <stdio.h>

main()
{
        char    buf[80];
        int     i;

        i = fread(buf, 1, sizeof(buf), stdin);
        printf("Read %d bytes\n", i);
}
```

RETURN VALUE

The return value is the number of objects read. If none is read, 0 will be returned. Note that a return value less than **cnt**, but greater than 0, may not represent an error (cf. *fwrite()*).

SEE ALSO

fwrite, fopen, fclose, getc

FREE

SYNOPSIS

```
#include <stdlib.h>
void     free(void * ptr)
```

DESCRIPTION

Free() deallocates the block of memory at **ptr**, which must have been obtained from a call to *malloc()* or *calloc()*.

EXAMPLE

```
#include        <stdlib.h>
#include        <stdio.h>

struct test {
        int     a[20];
} *     ptr;

main()
{       /* allocate space for 20 structs */

        ptr = calloc(20, sizeof(struct test));
        if(!ptr)
                printf("Failed\n");
        else
                free(ptr);
}
```

SEE ALSO

malloc, calloc

FREOPEN

SYNOPSIS

```
#include <stdio.h>
FILE * freopen(char * name, char * mode, FILE * stream)
```

DESCRIPTION

Freopen() closes the given **stream** (if open) then re-opens the stream attached to the file described by **name**. The mode of opening is given by **mode**. This function is commonly used to attach *stdin* or *stdout* to a file, as in the example.

EXAMPLE

```
#include        <stdio.h>

main()
{
        char    buf[80];

        if(!freopen("test.fil", "r", stdin))
                perror("test.fil");
        if(gets(buf))
                puts(buf);
}
```

RETURN VALUE

It either returns the **stream** argument, if successful, or NULL if not. See *fopen()* for more information.

SEE ALSO

fopen, fclose

FREXP

SYNOPSIS

```
#include <math.h>
double   frexp(double f, int * p)
```

DESCRIPTION

Frexp() breaks a floating point number into a normalized fraction and an integral power of 2. The integer is stored into the int object pointed to by **p**. Its return value **x** is in the interval [0.5, 1.0) or zero, and **f** equals **x** times 2 raised to the power stored in ***p**. If **f** is zero, both parts of the result are zero.

EXAMPLE

```
#include        <math.h>
#include        <stdio.h>

main()
{
        double  f;
        int     i;

        f = frexp(23456.34, &i);
        printf("23456.34 = %f * 2^%d\n", f, i);
}
```

SEE ALSO

ldexp

FSCANF

SYNOPSIS

```
#include <stdio.h>
int      fscanf(FILE * stream, char * fmt, ...)
```

DESCRIPTION

This routine performs formatted input from the specified **stream**. See *scanf()* for a full description of the behaviour of the routine.

EXAMPLE

```
#include        <stdio.h>

main()
{
        int     i;

        printf("Enter a number: ");
        fscanf(stdin, "%d", &i);
        printf("Read %d\n", i);
}
```

RETURN VALUE

The number of values assigned, or EOF if an error occurred and no items were converted.

SEE ALSO

scanf, sscanf, fopen, fclose

FSEEK

SYNOPSIS

```
#include <stdio.h>
int      fseek(FILE * stream, long offs, int wh)
```

DESCRIPTION

Fseek() positions the "file pointer" (i.e. a pointer to the next character to be read or written) of the specified **stream** as follows:

wh	Resultant location
0	offs
1	offs+previous location
2	offs+length of file

It should be noted that **offs** is a signed value. Thus the 3 allowed modes give postioning relative to the beginning of the file, the current file pointer and the end of the file respectively. Note however that positioning beyond the end of the file is legal, but will result in an EOF indication if an attempt is made to read data there. It is quite in order to write data beyond the previous end of file. *Fseek()* correctly accounts for any buffered data. The current file position can be determined with the function *ftell()*.

EXAMPLE

```
#include        <stdio.h>
#include        <stdlib.h>

main()
{
        FILE *  fp;

         /* open file for read/write */
        fp = fopen("test.fil", "r+");
        if(!fp)
                exit(1);
        fseek(fp,0L, 2);        /* seek to end */
        fputs("Another line!\n", fp);
        fclose(fp);
}
```

RETURN VALUE

EOF is returned if the positioning request could not be satisfied, otherwise zero.

SEE ALSO

lseek, fopen, fclose, ftell

FTELL

SYNOPSIS

```
#include <stdio.h>
long     ftell(FILE * stream)
```

DESCRIPTION

This function returns the current position of the conceptual read/write pointer associated with **stream**. This is the position relative to the beginning of the file of the next byte to be read from or written to the file.

EXAMPLE

```
#include        <stdio.h>
#include        <stdlib.h>

main()
{
        FILE *  fp;

        fp = fopen("test.fil", "r");
        if(!fp)
                exit(1);
        fseek(fp, 0L, 2);       /* seek to end */
        printf("size = %ld\n", ftell(fp));
}
```

SEE ALSO

fseek

FWRITE

SYNOPSIS

```
#include <stdio.h>
int      fwrite(void * buf, size_t size, size_t cnt,
             FILE * stream)
```

DESCRIPTION

Cnt objects of length **size** bytes will be written from memory at **buf**, to the specified **stream**.

EXAMPLE

```
#include        <stdio.h>

main()
{
        if(fwrite("a test string\n", 1, 14, stdout) != 14)
                fprintf(stderr, "Fwrite failed\n");
}
```

RETURN VALUE

The number of whole objects written will be returned, or 0 if none could be written. Any return value not equal to **cnt** should be treated as an error (cf. *fread()*).

SEE ALSO

fread, fopen, fclose

GETC

SYNOPSIS

```
#include <stdio.h>
int      getc(FILE * stream)
FILE * stream;
```

DESCRIPTION

One character is read from the specified stream and returned. This is the macro version of *fgetc()*, and is defined in stdio.h.

EXAMPLE

```
#include        <stdio.h>

main()
{
        int     i;

        while((i = getc(stdin)) != EOF)
                putchar(i);
}
```

RETURN VALUE

EOF will be returned on end-of-file or error.

GETCH, GETCHE, UNGETCH

SYNOPSIS

```
#include <conio.h>
char     getch(void)
char     getche(void)
void     ungetch(char)
```

DESCRIPTION

Getch() reads a single character from the console keyboard and returns it without echoing. *Getche()* is similar but does echo the character typed. *Ungetch()* will push back one character such that the next call to *getch()* or *getche()* will return that character. to the console screen, prepending a carriage return if the character is a newline. In an embedded system, the source of characters is defined by the particular routines supplied. By default, the library contains a version of *getch()* that will interface to the Lucifer debugger. The user should supply an appropriate routine if another source is desired, e.g. a serial port.

The module **getch.c** in the SOURCES directory contains model versions of all the console I/O routines. Other modules may also be supplied, e.g. **ser180.c** has routines for the serial port in a Z180.

EXAMPLE

```
#include        <conio.h>

main()
{
        char    c;

        while((c = getche()) != '\n')
                continue;
}
```

SEE ALSO

cgets, cputs

GETCHAR

SYNOPSIS

```
#include <stdio.h>
int      getchar(void)
```

DESCRIPTION

Getchar() is a *getc(stdin)* operation. It is a macro defined in stdio.h. Note that under normal circumstances getchar() will NOT return unless a carriage return has been typed on the console. To get a single character immediately from the console, use the routine *getch()*.

SEE ALSO

getc, fgetc, freopen, fclose

GETCWD

SYNOPSIS

```
#include <sys.h>
char *   getcwd(char * drive)
```

DESCRIPTION

Getcwd() returns the path name of the current working directory on the specified **drive**, where **drive** is a string, the first character of which is taken as a drive letter. If the string is null (“”) then the current drive working directory will be returned.

EXAMPLE

```
#include        <sys.h>
#include        <stdio.h>

main()
{
        char *  cp;

        cp = getcwd(“C”);
        printf(“cwd = %s\n”, cp);
}
```

RETURN VALUE

The return value is a pointer to a static area of memory which will be overwritten on the next call to *getcwd()*.

SEE ALSO

chdir, getdrv

GETDRV

SYNOPSIS

```
#include <sys.h>
char * getdrv(void)
```

DESCRIPTION

Calling this function will return a pointer to a static buffer containing a string of the form "**X:**", where **X** is the letter representing the current drive, e.g. "C:".

EXAMPLE

```
#include    <sys.h>
#include    <stdio.h>

main()
{
    char *  cp;

    cp = getdrv();
    printf("Current drive is %s\n", cp);
}
```

SEE ALSO

getcwd, setdrv

GETENV

SYNOPSIS

```
#include <stdlib.h>
char *   getenv(char * s)
extern char **       environ;
```

DESCRIPTION

Getenv() will search the vector of environment strings for one matching the argument supplied, and return the value part of that environment string. For example, if the environment contains the string

COMSPEC=C:\COMMAND.COM

then *getenv("COMSPEC")* will return **C:\COMMAND.COM**. The global variable **environ** is a pointer to an array of pointers to environment strings, terminated by a null pointer. This array is initialized at startup time under MS-DOS from the environment pointer supplied when the program was executed. Under CP/M no such environment is supplied, so the first call to *getenv()* will attempt to open a file in the current user number on the current drive called **ENVIRON**. This file should contain definitions for any environment variables desired to be accessible to the program, e.g.

HITECH=0:C:

Each variable definition should be on a separate line, consisting of the variable name (conventionally all in upper case) followed without intervening white space by an equal sign ('=') then the value to be assigned to that variable.

EXAMPLE

```
#include        <stdlib.h>
#include        <stdio.h>

main()
{
        printf("comspec = %s\n", getenv("COMSPEC"));
}
```

RETURN VALUE

NULL if the specified variable could not be found.

GETFREEMEM

SYNOPSIS

```
#include <sys.h>
void     getfreemem(struct freemem *);
```

DESCRIPTION

To find the free DOS memory, both conventional and extended, use this function. It fills in a buffer whose address is passed with the free memory in bytes.

EXAMPLE

```
#include    <sys.h>
#include    <stdio.h>

main()
{
    struct freemem  fm;

    getfreemem(&fm);
    printf("conventional memory: %lu bytes, extended memory: %lu bytes\n",
        fm.fr_dosmem, fm.fr_extmem);
}
```

DATA TYPES

```
struct freemem {       unsigned long   fr_dosmem;
    unsigned long   fr_extmem;
};
```

NOTE

The "extended" memory value is the amount of extended memory available via the BIOS. For XMS memory, see the _xms routines.

GETIVA

SYNOPSIS

```
#include <intrpt.h>
extern isr getiva(int intnum);
```

DESCRIPTION

Getiva() returns a pointer to a far function that is currently pointed to by the nominated interrupt vector. It uses the DOS int 21h/35h call.

EXAMPLE

```
#include        <intrpt.h>
#include        <stdio.h>

main()
{
        isr     fp;

        fp = getiva(0x23);
        printf("int 23h points to address %4.4X:%4.4X\n",
                (unsigned)((unsigned long)fp > 16), (unsigned short)fp);
}
```

DATA TYPES

typedef far interrupt void (*isr)(void);

SEE ALSO

setiva

GETS

SYNOPSIS

```
#include <stdio.h>
char * gets(char * s)
```

DESCRIPTION

Gets() reads a line from standard input into the buffer at **s**, deleting the newline (cf. *fgets()*). The buffer is null terminated. In an embedded system, gets is equivalent to *cgets()*, and results in *getche()* being called repeatedly to getch characters. Editing (with backspace) is available.

EXAMPLE

```
#include        <stdio.h>

main()
{
        char    buf[80];

        printf("Type a line: ");
        if(gets(buf))
                puts(buf);
}
```

RETURN VALUE

It returns its argument, or NULL on end-of-file.

SEE ALSO

fgets, freopen, puts

GETW

SYNOPSIS

```
#include <stdio.h>
int      getw(FILE * stream)
```

DESCRIPTION

Getw() returns one word (16 bits for the Z80 and 8086) from the nominated **stream**. EOF is returned on end-of-file, but since this is a perfectly good word, the *feof()* macro should be used for testing for end-of-file. When reading the word, no special alignment in the file is necessary, as the read is done by two consecutive *getc()*'s. The byte ordering is however undefined. The word read should in general have been written by *putw()*. Do not rely on this function to read binary data written by another program.

SEE ALSO

putw, getc, fopen, fclose

GMTIME

SYNOPSIS

```
#include <time.h>
struct tm *          gmtime(time_t * t)
```

DESCRIPTION

This function converts the time pointed to by **t** which is in seconds since 00:00:00 on Jan 1, 1970, into a broken down time stored in a structure as defined in *time.h*. The structure is as follows:

```
struct tm {        int     tm_sec;
        int     tm_min;
        int     tm_hour;
        int     tm_mday;   /* day of month */
        int     tm_mon;    /* month; 0-11
        int     tm_year;   /* year -1900 */
        int     tm_wday;   /* weekday - sunday = 0 */
        int     tm_yday;   /* day of year - 0-365 */
        int     tm_isdst;
};
```

EXAMPLE

```
#include        <stdio.h>
#include        <time.h>

main()
{
        time_t  clock;
        struct tm *     tp;

        time(&clock);
        tp = gmtime(&clock);
        printf("It's %d in London\n",
                 tp-tm_year+1900);
}
```


DATA TYPES

typedef long time_t; struct tm {
    int   tm_sec;
    int   tm_min;

```
    int     tm_hour;
    int     tm_mday;
    int     tm_mon;
    int     tm_year;
    int     tm_wday;
    int     tm_yday;
    int     tm_isdst;
};
```

SEE ALSO

ctime, asctime, time, localtime

INT86, INT86X

SYNOPSIS

```
#include <dos.h>
int      int86(int intno, union REGS * inregs,
         union REGS * outregs)
int      int86x(int intno, union REGS inregs,
         union REGS outregs, struct SREGS * segregs)
```

DESCRIPTION

These functions allow calling of software interrupts from C programs. *Int86()* and *int86x()* execute the software interrupt specified by *intno*. The **inregs** pointer should point to a union containing values for each of the general purpose registers to be set when executing the interrupt, and the values of the registers on return are copied into the union pointed to by **outregs**. The *x* versions of the calls also take a pointer to a union defining the segment register values to be set on execution of the interrupt, though only ES and DS are actually set from this structure.

EXAMPLE

```
#include        <stdio.h>
#include        <dos.h>

/*      Check available BIOS extended memory */

main()
{
        union REGS      rbuf;

        rbuf.x.ax = 0x8800;
        int86(0x15, &rbuf, &rbuf);
        printf("Free extended memory = %uK\n",
                rbuf.x.ax);
}
```

RETURN VALUE

The return value is of no significance. Test the appropriate register values.

DATA TYPES

```
struct WORDREGS {       unsigned int ax;
    unsigned int bx;
    unsigned int cx;
```

```
        unsigned int dx;
        unsigned int si;
        unsigned int di;
        unsigned int cflag;
        unsigned int psw;
};

struct BYTEREGS {
        unsigned char al, ah;
        unsigned char bl, bh;
        unsigned char cl, ch;
        unsigned char dl, dh;
};

union REGS {
        struct WORDREGS x;
        struct BYTEREGS h;
};

struct SREGS {
        unsigned int es;
        unsigned int cs;
        unsigned int ss;
        unsigned int ds;
};
```

SEE ALSO

segread, intdos, intdosx

INTDOS, INTDOSX

SYNOPSIS

```
#include <dos.h>
int      intdos(union REGS * inregs, union REGS * outregs)
int      intdosx(union REGS * inregs, union REGS * outregs,
         struct SREGS * segregs)
```

DESCRIPTION

These functions allow calling of DOS 0x21 interrupts from C programs. The **inregs** pointer should point to a union containing values for each of the general purpose registers to be set when executing the interrupt, and the values of the registers on return are copied into the union pointed to by **outregs**. The *x* version of the call also takes a pointer to a union defining the segment register values to be set on execution of the interrupt, though only ES and DS are actually set from this structure.

EXAMPLE

```
#include        <stdio.h>
#include        <dos.h>

main()
{
        /* determine DOS free memory */
        union REGS rbuf;

        rbuf.x.ax = 0x4800;
        rbuf.x.bx = 0xFFFF;
        intdos(&rbuf, &rbuf);
        printf("DOS has %lu bytes free\n", rbuf.x.bx*16l);
}
```

DATA TYPES

```
struct WORDREGS {       unsigned int ax;
    unsigned int bx;
    unsigned int cx;
    unsigned int dx;
    unsigned int si;
    unsigned int di;
    unsigned int cflag;
    unsigned int psw;
};
```

```
struct BYTEREGS {
        unsigned char al, ah;
        unsigned char bl, bh;
        unsigned char cl, ch;
        unsigned char dl, dh;
};

union REGS {
        struct WORDREGS x;
        struct BYTEREGS h;
};

struct SREGS {
        unsigned int es;
        unsigned int cs;
        unsigned int ss;
        unsigned int ds;
};
```

SEE ALSO

segread

ISALNUM, ISALPHA, ISDIGIT, ISLOWER et. al.

SYNOPSIS

```
#include <ctype.h>
isalnum(char c)
isalpha(char c)
isascii(char c)
iscntrl(char c)
isdigit(char c)
islower(char c)
isprint(char c)
isgraph(char c)
ispunct(char c)
isspace(char c)
isupper(char c)
isalnum(char c)
```

DESCRIPTION

These macros, defined in ctype.h, test the supplied character for membership in one of several overlapping groups of characters. Note that all except isascii are defined for *c* if *isascii(c)* is true or if c = EOF.

```
isalnum(c)        c is alphanumeric isalpha(c)  c is in A-Z or a-z
isascii(c)        c is a 7 bit ascii character
iscntrl(c)        c is a control character
isdigit(c)        c is a decimal digit
islower(c)        c is in a-z
isprint(c)        c is a printing char
isgraph(c)        c is a non-space printable character
ispunct(c)        c is not alphanumeric
isspace(c)        c is a space, tab or newline
isupper(c)        c is in A-Z
isxdigit(c)       c is in 0-9 or a-f or A-F
isalnum(c)        c is in 0-9 or a-z or A-Z
```


EXAMPLE

```
#include       <ctype.h>
#include       <stdio.h>

main()
```

```
{
        char    buf[80];
        int     i;

        gets(buf);
        i = 0;
        while(isalnum(buf[i]))
                i++;
        buf[i] = 0;
        printf("'%s' is the word\n", buf);
}
```

SEE ALSO

toupper, tolower, toascii

ISATTY

SYNOPSIS

```
#include <unixio.h>
int isatty(int fd)
```

DESCRIPTION

This tests the type of the file associated with **fd**. It returns true if the file is attached to a tty-like device. This would normally be used for testing if standard input is coming from a file or the console. For testing STDIO streams, use *isatty(fileno(stream))*.

EXAMPLE

```
#include        <unixio.h>
#include        <stdio.h>

main()
{
        if(isatty(fileno(stdin)))
                printf("input not redirected\n");
        else
                printf("Input is redirected\n");
}
```

RETURN VALUE

Zero if the stream is associated with a file; 1 if it is associated with the console or other keyboard type device.

ISNEC98

SYNOPSIS

```
#include <dos.h>
int isnec98(void);
```

DESCRIPTION

This function returns TRUE if executed on a NEC 98xx series computer, false otherwise. If you don't know what a NEC 98 series machine is, you don't need this function. If you're curious, it is a Japanese PC that runs MS-DOS but is NOT and IBM compatible.

EXAMPLE

```
#include <dos.h>
#include <stdio.h>

main()
{
    printf("This is%s a NEC 98xx machine\n",
        isnec98() ? "" : " not");
}
```

KBHIT

SYNOPSIS

```
#include <conio.h>
int      kbhit(void)
```

DESCRIPTION

This function returns 1 if a character has been pressed on the console keyboard, 0 otherwise. Normally the character would then be read via *getch()*.

EXAMPLE

```
#include        <conio.h>

main()
{
        int     i;

        while(!kbhit()) {
                cputs("I'm waiting..");
                for(i = 0 ; i != 1000 ; i++)
                        continue;
        }
}
```

SEE ALSO

getch, getche

LDEXP

SYNOPSIS

```
#include <math.h>
double   ldexp(double f, int i)
```

DESCRIPTION

Ldexp() performs the inverse of *frexp()*. operation; the integer **i** is added to the exponent of the floating point **f** and the resultant value returned.

EXAMPLE

```
#include        <math.h>
#include        <stdio.h>

main()
{
        double  f;

        f = ldexp(1.0, 10);
        printf("1.0 * 2^10 = %f\n", f);
}
```

SEE ALSO

frexp

LDIV

SYNOPSIS

```
#include <stdlib.h>
ldiv_t          ldiv(long number, long denom);
```

DESCRIPTION

The *ldiv()* routine divides the numerator by the denominator, computing the quotient and the remainder. The sign of the quotient is the same as that of the mathematical quotient. Its absolute value is the largest integer which is less than the absolute value of the mathematical quotient.

The *ldiv()* function is similar to the *div()* function, the difference being that the arguments and the members of the returned structure are all of type long int.

EXAMPLE

```
#include    <stdlib.h>
#include    <stdio.h>

main()
{
    ldiv_t  lt;

    lt = ldiv(1234567, 12345);
    printf("Quotient = %ld, remainder = %ld\n",
        lt.quot, lt.rem);
}
```

RETURN VALUE

A structure of type **ldiv_t**

DATA TYPES

```
typedef struct {       long    quot;   /* quotient */
     long    rem;    /* remainder */
} ldiv_t;
```

SEE ALSO

div

LOCALTIME

SYNOPSIS

```
#include <time.h>
struct tm *          localtime(time_t * t)
```

DESCRIPTION

Localtime converts the time pointed to by **t** which is in seconds since 00:00:00 on Jan 1, 1970, into a broken down time stored in a structure as defined in *time.h*. *Localtime()* takes into account the contents of the global integer **time_zone**. This should contain the number of minutes that the local time zone is **WESTWARD** of Greenwich. Since there is no way under MS-DOS of actually pre-determining this value, by default *localtime()* will return the same result as *gmtime()*.

EXAMPLE

```
#include        <stdio.h>
#include        <time.h>

char *  wday[] = {
“Sunday”, “Monday”, “Tuesday”, “Wednesday”,
“Thursday”, “Friday”, “Saturday”
};

main()
{
        time_t  clock;
        struct tm *     tp;

        time(&clock);
        tp = localtime(&clock);
        printf(“Today is %s\n”, wday[tp-tm_wday]);
}
```

DATA TYPES

typedef long time_t; struct tm {

```
    int    tm_sec;
    int    tm_min;
    int    tm_hour;
    int    tm_mday;
    int    tm_mon;
    int    tm_year;
    int    tm_wday;
```

```
    int     tm_yday;
    int     tm_isdst;
};
```

SEE ALSO

ctime, asctime, time

LOG, LOG10

SYNOPSIS

```
#include <math.h>
double   log(double f)
double   log10(double f)
```

DESCRIPTION

Log() returns the natural logarithm of **f**, i.e. the number *x* such that e e sup x = f . *Log10()* returns the logarithm to base 10 of **f**, i.e the number *x* such that e 10 sup x = f .

EXAMPLE

```
#include        <math.h>
#include        <stdio.h>

main()
{
        double  f;

        for(f = 1.0 ; f <= 10.0 ; f += 1.0)
                printf("log(%1.0f) = %f\n", f, log(f));
}
```

RETURN VALUE

Zero if the argument is negative.

SEE ALSO

exp, pow

LONGJMP

SYNOPSIS

```
#include <setjmp.h>
void     longjmp(jmp_buf buf, int val)
```

DESCRIPTION

Longjmp(), in conjunction with *setjmp()*, provides a mechanism for non-local gotos. To use this facility, *setjmp()* should be called with a **jmp_buf** argument in some outer level function. The call from *setjmp()* will return 0. To return to this level of execution, *lonjmp()* may be called with the same **jmp_buf** argument from an inner level of execution. Note however that the function which called *setjmp()* must still be active when *longjmp()* is called. Breach of this rule will cause disaster, due to the use of a stack containing invalid data. The **val** argument to *longjmp()* will be the value apparently returned from the *setjmp()*. This should normally be non-zero, to distinguish it from the genuine *setjmp()* call.

EXAMPLE

```
#include        <stdio.h>
#include        <setjmp.h>
#include        <stdlib.h>

jmp_buf jb;

inner(void)
{
        longjmp(jb, 5);
}

main()
{
        int     i;

        if(i = setjmp(jb)) {
                printf("setjmp returned %d\n", i);
                exit(0);
        }
        printf("setjmp returned 0 - good\n");
        printf("calling inner...\n");
        inner();
        printf("inner returned - bad!\n");
}
```

RETURN VALUE

Longjmp never returns.

SEE ALSO

setjmp

LSEEK

SYNOPSIS

```
#include <unixio.h>
long lseek(int fd, long offs, int wh)
```

DESCRIPTION

This function operates in an analogous manner to *fseek()*, however it does so on unbuffered low-level i/o file descriptors, rather than on STDIO streams. It also returns the resulting pointer location. Thus *lseek(fd, 0L, 1)* returns the current pointer location without moving it.

EXAMPLE

```
#include        <stdio.h>
#include        <stdlib.h>
#include        <unixio.h>

main()
{
        int     fd;

        fd = open("test.fil", 1); /* open for write */
        if(fd < 0)
                exit(1);
        lseek(fd,0L, 2);        /* seek to end */
        write(fd, "more stuff\r\n", 12);
        close(fd);
}
```

RETURN VALUE

-1 is returned on error, the resulting location otherwise.

SEE ALSO

open, close, read, write

MALLOC

SYNOPSIS

```
#include <stdlib.h>
void * malloc(size_t cnt)
```

DESCRIPTION

Malloc() attempts to allocate **cnt** bytes of memory from the “heap”, the dynamic memory allocation area. If successful, it returns a pointer to the block, otherwise 0 is returned. The memory so allocated may be freed with *free(),* or changed in size via *realloc()*. *Malloc()* calls *sbrk()* to obtain memory, and is in turn called by *calloc()*. *Malloc()* does not clear the memory it obtains, unlike *calloc()*..

EXAMPLE

```
#include        <stdlib.h>
#include        <stdio.h>
#include        <string.h>


main()
{
        char *  cp;

        cp = malloc(80);
        if(!cp)
                printf(“Malloc failed\n”);
        else {
                strcpy(cp, “a string”);
                printf(“block = ‘%s’\n”, cp);
                free(cp);
        }
}
```

RETURN VALUE

A pointer to the memory if it succeeded; NULL otherwise.

SEE ALSO

calloc, free, realloc

MEMCHR

SYNOPSIS

```
#include <string.h>
void * memchr(const void * block, int val, size_t length);
```

DESCRIPTION

Memchr() is similar to *strchr()* except that instead of searching null-terminated strings, it searches a block of memory specified by length for a particular byte. Its arguments are a pointer to the memory to be searched, the value of the byte to be searched for, and the length of the block. A pointer to the first occurrence of that byte in the block is returned.

EXAMPLE

```
#include    <string.h>
#include    <stdio.h>

unsigned int    ary[] = {
    1, 5, 0x6789, 0x23
};

main()
{
    char *  cp;

    cp = memchr(ary, 0x89, sizeof ary);
    if(!cp)
        printf("not found\n");
    else
        printf("Found at offset %u\n", cp - (char *)ary);
}
```

RETURN VALUE

A pointer to the first byte matching the argument if one exists; NULL otherwise.

SEE ALSO

strchr

MEMCMP

SYNOPSIS

```
#include <string.h>
int      memcmp(void * s1, void * s2, size_t n)
```

DESCRIPTION

Memcmp() compares two blocks of memory, of length **n**, and returns a signed value similar to *strncmp()*. Unlike *strncmp()* the comparision does not stop on a null character. The ASCII collating sequence is used for the comparision, but the effect of including non-ASCII characters in the memory blocks on the sense of the return value is indeterminate. Testing for equality is always reliable.

EXAMPLE

```
#include        <stdio.h>
#include        <string.h>

main()
{
        int     buf[10], cow[10], i;

        buf[0] = 1;
        buf[2] = 4;
        cow[0] = 1;
        cow[2] = 5;
        buf[1] = 3;
        cow[1] = 3;
        i = memcmp(buf, cow, 3*sizeof(int));
        if(i < 0)
                printf("less than\n");
        else if(i > 0)
                printf("Greater than\n");
        else
                printf("Equal\n");
}
```

RETURN VALUE

Memcmp() returns -1, 0 or 1, depending on whether **s1** points to string which is less than, equal to or greater than the string pointed to by **s2** in the collating sequence.

SEE ALSO

strncpy, strncmp, strchr, memset, memchr

MEMCPY

SYNOPSIS

```
#include <string.h>
void *   memcpy(void * d, void * s, size_t n)
```

DESCRIPTION

Memcpy() copies **n** bytes of memory starting from the location pointed to by **s** to the block of memory pointed to by **d**. The result of copying overlapping blocks is undefined. *Memcpy()* differs from *strcpy()* in that it copies a specified number of bytes, rather than all bytes up to a null terminator.

EXAMPLE

```
#include        <string.h>
#include        <stdio.h>

main()
{
        char    buf[80];

        memset(buf, 0, sizeof buf);
        memcpy(buf, “a partial string”, 10);
        printf(“buf = ‘%s’\n”, buf);
}
```

RETURN VALUE

Memcpy() returns its first argument.

SEE ALSO

strncpy, strncmp, strchr, memset

MEMMOVE

SYNOPSIS

```
#include <string.h>
void *   memmove(void * s1, void * s2, size_t n)
```

DESCRIPTION

Memmove() is similar to *memcpy()* except copying of overlapping blocks is handled correctly. That is, it will copy forwards or backwards as appropriate to correctly copy one block to another that overlaps it.

RETURN VALUE

Memmove() returns its first argument.

SEE ALSO

strncpy, strncmp, strchr

MEMSET

SYNOPSIS

```
#include <string.h>
void     memset(void * s, char c, size_t n)
```

DESCRIPTION

Memset() fills **n** bytes of memory starting at the location pointed to by **s** with the character **c**.

EXAMPLE

```
#include        <string.h>
#include        <stdio.h>

main()
{
        char    abuf[20];

        strcpy(abuf, “This is a string”);
        memset(abuf, ‘x’, 5);
        printf(“buf = ‘%s’\n”, abuf);
}
```

SEE ALSO

strncpy, strncmp, strchr, memcpy, memchr

MKDIR

SYNOPSIS

```
#include <sys.h>
int      mkdir(char * s)
```

DESCRIPTION

This function creates a directory under MS-DOS. The argument is a path name, and if successful the directory specified by the path name will be created. All interemediate directories in the path name must already exist.

EXAMPLE

```
#include        <sys.h>
#include        <stdio.h>

main()
{
        if(mkdir("test.dir") < 0)
                perror("test.dir");
        else
                printf("Succeeded\n");
}
```

RETURN VALUE

Success returns 0, failure -1.

SEE ALSO

chdir, rmdir

OPEN

SYNOPSIS

```
#include <unixio.h>
int      open(char * name, int mode)
```

DESCRIPTION

Open() is the fundamental means of opening files for reading and writing. The file specified by **name** is sought, and if found is opened for reading, writing or both. **Mode** is encoded as follows:

```
Mode     Meaning

0        Open for reading only
1        Open for writing only
2        Open for both reading and writing
```

The file must already exist - if it does not, *creat()* should be used to create it. On a successful open, a file descriptor is returned. This is a non-negative integer which may be used to refer to the open file subsequently. If the open fails, -1 is returned.Under MS-DOS the syntax of filenames are standard MS-DOS. The syntax of a CP/M filename is:

```
[uid:][drive:]name.type
```

where uid is a decimal number 0 to 15, drive is a letter A to P or a to p, name is 1 to 8 characters and type is 0 to 3 characters. Though there are few inherent restrictions on the characters in the name and type, it is recommended that they be restricted to the alphanumerics and standard printing characters. Use of strange characters may cause problems in accessing and/or deleting the file.

One or both of uid: and drive: may be omitted; if both are supplied, the uid: must come first. Note that the [and] are meta-symbols only. Some examples are:

```
fred.dat file.c
0:xyz.com
0:a:file1.p
a:file2.
```


If the uid: is omitted, the file will be sought with uid equal to the current user number, as returned by getuid(). If drive: is omitted, the file will be sought on the currently selected drive. The following special file names are recognized:

```
lst:     Accesses the list device - write only
```

```
pun:     Accesses the punch device - write only
rdr:     Accesses the reader device - read only
con:     Accesses the system console - read/write
```

File names may be in any case - they are converted to upper case during processing of the name.

MS-DOS filenames may be any valid MS-DOS 2.xx filename, e.g.

```
fred.nrk A:\HITECH\STDIO.H
```

The special device names (e.g. CON, LST) are also recognized. These do not require (and should not have) a trailing colon.

EXAMPLE

```
#include        <stdio.h>
#include        <stdlib.h>
#include        <unixio.h>

main()
{
        int     fd;

        fd = open(“test.fil”, 1); /* open for write */
        if(fd < 0)
                exit(1);
        lseek(fd,0L, 2);        /* seek to end */
        write(fd, “more stuff\r\n”, 12);
        close(fd);
}
```

SEE ALSO

close, fopen, fclose, read, write, creat

PERROR

SYNOPSIS

```
#include <stdio.h>
void     perror(char * s)
```

DESCRIPTION

This routine will print on the stderr stream the argument **s**, followed by a descriptive message detailing the last error returned from a DOS system call. The error number is retrieved from the global variable **errno**. *Perror()* is of limited usefulness under CP/M as it does not give as much error information as MS-DOS.

EXAMPLE

```
#include        <stdio.h>
#include        <stdlib.h>

main(argc, argv)
char ** argv;
{
        if(argc < 2)
                exit(0);
        if(!freopen(argv[1], “r”, stdin))
                perror(argv[1]);
}
```

SEE ALSO

strerror

POW

SYNOPSIS

```
#include <math.h>
double   pow(double f, double p)
```

DESCRIPTION

Pow() raises its first argument, **f**, to the power **p**.

EXAMPLE

```
#include        <math.h>
#include        <stdio.h>

main()
{
        double  f;

        for(f = 1.0 ; f <= 10.0 ; f += 1.0)
                printf("pow(2, %1.0f) = %f\n", f, pow(2, f));
}
```

SEE ALSO

log, log10, exp

PRINTF, VPRINTF

SYNOPSIS

```
#include <stdio.h>
int      printf(char * fmt, ...)
int      vprintf(char * fmt, va_list va_arg)
```

DESCRIPTION

Printf() is a formatted output routine, operating on stdout. There are corresponding routines operating on a given stream (*fprintf()*) or into a string buffer (*sprintf()*). *Printf()* is passed a format string, followed by a list of zero or more arguments. In the format string are conversion specifications, each of which is used to print out one of the argument list values. Each conversion specification is of the form **%m.nc** where the percent symbol **%** introduces a conversion, followed by an optional width specification **m. n** is an optional precision specification (introduced by the dot) and **c** is a letter specifying the type of the conversion. A minus sign ('-') preceding **m** indicates left rather than right adjustment of the converted value in the field. Where the field width is larger than required for the conversion, blank padding is performed at the left or right as specified. Where right adjustment of a numeric conversion is specified, and the first digit of **m** is 0, then padding will be performed with zeroes rather than blanks. For integer formats, the precision indicates a minimum number of digits to be output, with leading zeros inserted to make up this number if required.

A hash character (#) preceding the width indicates that an alternate format is to be used. The nature of the alternate format is discussed below. Not all formats have alternates. In those cases, the presence of the hash character has no effect.

The floating point formats require that the appropriate floating point library is linked. From within HPD this can be forced by selecting the "Float formats in printf" selection in the options menu. From the command line driver, use the option **-LF**.

If the character ***** is used in place of a decimal constant, e.g. in the format **%*d**, then one integer argument will be taken from the list to provide that value. The types of conversion are:

f Floating point - m is the total width and n is the number of digits after the decimal point. If n is omitted it defaults to 6. If the precision is zero, the decimal point will be omitted unless the alternate format is specified.

e Print the corresponding argument in scientific notation. Otherwise similar to **f**.

g Use **e** or **f** format, whichever gives maximum precision in minimum width. Any trailing zeros after the decimal point will be removed, and if no digits remain after the decimal point, it will also be removed.

o x X u d Integer conversion - in radices 8, 16, 16, 10 and 10 respectively. The conversion is signed in the case of d, unsigned otherwise. The precision value is the total number of digits to print, and may be used to force leading zeroes. E.g. **%8.4x** will print at least 4 hex digits in an 8 wide field. Preceding the key letter with an **l** indicates that the value argument is a long integer or unsigned value. The letter **X** prints out hexadecimal numbers using the upper case letters *A-F* rather than *a-f* as would be printed when using **x**. When the alternate format is specified, a leading zero will be supplied for the octal format, and al leading 0x or 0X for the hex format.

s Print a string - the value argument is assumed to be a character pointer. At most **n** characters from the string will be printed, in a field **m** characters wide.

c The argument is assumed to be a single character and is printed literally.

Any other characters used as conversion specifications will be printed. Thus **%%** will produce a single percent sign.

Vprintf() is similar to *printf()* but takes a variable argument list pointer rather than a list of arguments. See the description of *va_start()* for more information on variable argument lists. An example of using vprintf is given below.

EXAMPLE

```
printf(“Total = %4d%%”, 23)
       yields ‘Total =     23%’
printf(“Size is %lx” , size)
     where size is a long, prints size
     as hexadecimal.
printf(“Name = %.8s”, “a1234567890")
      yields ‘Name = a1234567’
printf(”xx%*d", 3, 4)
      yields ‘xx  4’

/* vprintf example */

#include        <stdio.h>

int
error(char * s, ...)
{
        va_list ap;

        va_start(ap, s);
        printf(“Error: ”);
        vprintf(s, ap);
        putchar(‘\n’);
        va_end(ap);
```

```
}

main()
{
        int     i;

        i = 3;
        error("testing 1 2 %d", i);
}
```

RETURN VALUE

Printf() returns the number of characters written to stdout.

SEE ALSO

fprintf, sprintf

PUTC

SYNOPSIS

```
#include <stdio.h>
int      putc(int c, FILE * stream)
```

DESCRIPTION

Putc() is the macro version of *fputc()* and is defined in stdio.h. It places the supplied character onto the specified I/O stream.

EXAMPLE

```
#include        <stdio.h>

char *  x = “this is a string”;

main()
{
        char *  cp;

        cp = x;
        while(*x)
                putc(*x++, stdout);
        putc(‘\n’, stdout);
}
```

RETURN VALUE

EOF on error, the character passed as argument otherwise.

SEE ALSO

fputc, getc, fopen, fclose, putch

PUTCH

SYNOPSIS

```
#include <conio.h>
void     putch(int c)
```

DESCRIPTION

Putch() outputs the character **c** to the console screen, prepending a carriage return if the character is a newline. In a CP/M or MS-DOS system this will ue one of the system I/O calls. In an embedded system this routine, and associated others, will be defined in a hardware dependent way. The standard *putch()* routines in the embedded library interface either to a serial port or to the Lucifer debugger.

EXAMPLE

```
#include        <conio.h>

char *  x = “this is a string”;

main()
{
        char *  cp;

        cp = x;
        while(*x)
                putch(*x++);
        putch(‘\n’);
}
```

SEE ALSO

cgets, cputs, getch, getche

PUTCHAR

SYNOPSIS

```
#include <stdio.h>
int      putchar(int c)
```

DESCRIPTION

Putchar() is a *putc()* operation on stdout, defined in stdio.h.

EXAMPLE

```
#include        <stdio.h>

char *  x = “this is a string”;

main()
{
        char *  cp;

        cp = x;
        while(*x)
                putchar(*x++);
        putchar(‘\n’);
}
```

RETURN VALUE

The character, or EOF if an error occurred.

SEE ALSO

putc, getc, freopen, fclose

PUTS

SYNOPSIS

```
#include <stdio.h>
int      puts(char * s)
```

DESCRIPTION

Puts() writes the string **s** to the stdout stream, appending a newline. Thenull terminating the string is not copied.

EXAMPLE

```
#include        <stdio.h>

main()
{
        puts(“Hello, world!”);
}
```

RETURN VALUE

EOF is returned on error; zero otherwise.

SEE ALSO

fputs, gets, freopen, fclose

PUTW

SYNOPSIS

```
#include <stdio.h>
int      putw(int w, FILE * stream)
```

DESCRIPTION

Putw() copies the word **w** to the given **stream**. It returns **w**, except on error, in which case EOF is returned. Since this is a good integer, *ferror()* should be used to check for errors. The routine *getw()* may be used to read in integer written by *putw()*.

SEE ALSO

getw, fopen, fclose

QSORT

SYNOPSIS

```
#include <stdlib.h>
void     qsort(void * base, size_t nel,
         size_t width, int (*func)(void *, void *))
```

DESCRIPTION

Qsort() is an implementation of the quicksort algorithm. It sorts an array of **nel** items, each of length **width** bytes, located contiguously in memory at **base**. **Func** is a pointer to a function used by *qsort()* to compare items. It calls **func** with pointers to two items to be compared. If the first item is considered to be greater than, equal to or less than the second then *func()* should return a value greater than zero, equal to zero or less than zero respectively.

EXAMPLE

```
#include        <stdio.h>
#include        <stdlib.h>

int array[] = {
        567, 23, 456, 1024, 17, 567, 66
};

int
sortem(const void * p1, const void * p2)
{
        return *(int *)p1 - *(int *)p2;
}

main()
{
        register int    i;

        qsort(aray, sizeof aray/sizeof aray[0],
                sizeof aray[0], sortem);
        for(i = 0 ; i != sizeof aray/sizeof aray[0] ; i++)
                printf("%d\t", aray[i]);
        putchar('\n');
}
```

NOTE

The function parameter must be a pointer to a function of type similar to:

int func(const void *, const void *)

i.e. it must accept two const void * parameters, and must be prototyped.

RAND

SYNOPSIS

```
#include <stdlib.h>
int rand(void)
```

DESCRIPTION

Rand() is a pseudo-random number generator. It returns an integer in the range 0 to 32767, which changes in a pseudo-random fashion on each call. The algorithm will produce a deterministic sequence if started from the same point. The starting point is set using the *srand()* call. The example shows use of the *time()* function to generate a different starting point for the sequence each time.

EXAMPLE

```
#include        <stdlib.h>
#include        <stdio.h>
#include        <time.h>

main()
{
        time_t  toc;
        int     i;

        time(&toc);
        srand((int)toc);
        for(i = 0 ; i != 10 ; i++)
                printf("%d\t", rand());
        putchar('\n');
}
```

SEE ALSO

srand

READ

SYNOPSIS

```
#include <unixio.h>
int      read(int fd, void * buf, size_t cnt)
```

DESCRIPTION

Read() will read from the file associated with **fd** up to **cnt** bytes into a buffer located at **buf**. It returns the number of bytes actually read. A zero return indicates end-of-file. A negative return indicates error. **Fd** should have been obtained from a previous call to *open()*. It is possible for *read()* to return less bytes than requested, e.g. when reading from the console, in which case *read()* will read one line of input.

EXAMPLE

```
#include        <stdio.h>
#include        <unixio.h>

main()
{
        char    buf[80];
        int     i;

        /* read from stdin */

        i = read(0, buf, sizeof(buf));
        printf("Read %d bytes\n", i);
}
```

RETURN VALUE

The number of bytes read; zero on EOF, -1 on error. Be careful not to misinterpret a read of > 32767 as a negative return value.

SEE ALSO

open, close, write

REALLOC

SYNOPSIS

```
#include <stdlib.h>
void * realloc(void * ptr, size_t cnt)
```

DESCRIPTION

Realloc() frees the block of memory at **ptr**, which should have been obtained by a previous call to *malloc(), calloc()* or *realloc()*, then attempts to allocate **cnt** bytes of dynamic memory, and if successful copies the contents of the block of memory located at **ptr** into the new block. At most, *realloc()* will copy the number of bytes which were in the old block, but if the new block is smaller, will only copy **cnt** bytes.

EXAMPLE

```
#include        <stdlib.h>
#include        <stdio.h>
#include        <string.h>

main()
{
        char *  cp;

        cp = malloc(255);
        if(gets(cp))
                cp = realloc(cp, strlen(cp)+1);
        printf("buffer now %d bytes long\n",
                strlen(cp)+1);
}
```

RETURN VALUE

A pointer to the new (or resized) block. NULL if the block could not be expanded. A request to shrink a block will never fail.

SEE ALSO

malloc, calloc, realloc

REMOVE

SYNOPSIS

```
#include <stdio.h>
int      remove(char * s)
```

DESCRIPTION

Remove() will attempt to remove the file named by the argument **s** from the directory.

EXAMPLE

```
#include        <stdio.h>

main()
{
        if(remove(“test.fil”) < 0)
                perror(“test.fil”);
}
```

RETURN VALUE

Zero on success, -1 on error.

SEE ALSO

unlink

RENAME

SYNOPSIS

```
#include <stdio.h>
int      rename(char * name1, char * name2)
```

DESCRIPTION

The file named by **name1** will be renamed to **name2**.

EXAMPLE

```
#include        <stdio.h>

main()
{
        if(rename("test.fil", "test1.fil"))
                perror("Rename");
}
```

RETURN VALUE

-1 will be returned if the rename was not successful, zero if the rename was performed.

NOTE

Rename is not permitted across drives or directories.

SEE ALSO

open, close, unlink

REWIND

SYNOPSIS

```
#include <stdio.h>
int      rewind(FILE * stream)
```

DESCRIPTION

This function will attempt to re-position the read/write pointer of the nominated **stream** to the beginning of the file. This call is equivalent to *fseek(stream, 0L, 0)*.

EXAMPLE

```
#include        <stdio.h>
#include        <stdlib.h>

main()
{
char buf[80];

        if(!freopen("test.fil", "r", stdin))
                exit(1);
        gets(buf);
        printf("got '%s'\n", buf);
        rewind(stdin);
        gets(buf);
        printf("Got '%s' again\n", buf);
}
```

RETURN VALUE

A return value of -1 indicates that the attempt was not successful, perhaps because the stream is associated with a non-random access file such as a character device.

SEE ALSO

fseek, ftell

RMDIR

SYNOPSIS

```
#include <sys.h>
int rmdir(char *)
```

DESCRIPTION

Rmdir() will remove the directory specified by the path name. The directory cannot be removed unless it is empty.

EXAMPLE

```
#include        <sys.h>
#include        <stdio.h>

main()
{
        if(rmdir("test.dir") < 0)
                perror("test.dir");
        else
                printf("Succeeded\n");
}
```

RETURN VALUE

Zero on success, -1 on failure

SBRK

SYNOPSIS

```
#include <stdlib.h>
char *   sbrk(int incr)
```

DESCRIPTION

Sbrk() increments the current highest memory location allocated to the program by **incr** bytes. It returns a pointer to the previous highest location. Thus *sbrk(0)* returns a pointer to the current highest location, without altering its value. This is a low-level routine not intended to be called by user code. Use *malloc()* instead.

RETURN VALUE

If there is insufficient memory to satisfy the request, (char *)-1 is returned.

SEE ALSO

brk, malloc, calloc, realloc, free

SCANF, VSCANF

SYNOPSIS

```
#include <stdio.h>
int      scanf(char * fmt, ...)
int      vscanf(char *, va_list ap);
```

DESCRIPTION

Scanf() performs formatted input ("de-editing") from the stdin stream. Similar functions are available for streams in general, and for strings. The function *vscanf()* is similar, but takes a pointer to an argument list rather than a series of additional arguments. This pointer should have been initialized with *va_start()*. The input conversions are performed according to the **fmt** string; in general a character in the format string must match a character in the input; however a space character in the format string will match zero or more "white space" characters in the input, i.e. spaces, tabs or newlines. A conversion specification takes the form of the character **%**, optionally followed by an assignment suppression character ('*'), optionally followed by a numerical maximum field width, followed by a conversion specification character. Each conversion specification, unless it incorporates the assignment suppression character, will assign a value to the variable pointed at by the next argument. Thus if there are two conversion specifications in the fmt string, there should be two additional pointer arguments. The conversion characters are as follows:

o x d Skip white space, then convert a number in base 8, 16 or 10 radix respectively. If a field width was supplied, take at most that many characters from the input. A leading minus sign will be recognized.

f Skip white space, then convert a floating number in either conventional or scientific notation. The field width applies as above.

s Skip white space, then copy a maximal length sequence of non-white-space characters. The pointer argument must be a pointer to char. The field width will limit the number of characters copied. The resultant string will be null-terminated.

c Copy the next character from the input. The pointer argument is assumed to be a pointer to char. If a field width is specified, then copy that many characters. This differs from the s format in that white space does not terminate the character sequence.

The conversion characters **o, x, u, d** and **f** may be preceded by an **l** to indicate that the corresponding pointer argument is a pointer to long or double as appropriate. A preceding **h** will indicate that the pointer argument is a pointer to short rather than int.

EXAMPLE

```
scanf(“%d %s”, &a, &s)
      with input “          12s”
will assign 12 to a, and “s” to s.

scanf(“%3cd %lf”, &c, &f)
     with input “ abcd -3.5"
     will assign ” abc" to c, and -3.5 to f.
```

RETURN VALUE

Scanf() returns the number of successful conversions; EOF is returned if end-of-file was seen before any conversions were performed.

SEE ALSO

fscanf, sscanf, printf, va_arg

SEGREAD

SYNOPSIS

```
#include <dos.h> int segread(struct SREGS * segregs)
```

DESCRIPTION

Segread() copies the values of the segment registers into the structure pointed to by **segregs**. This usually used to initialize a **struct SREGS** before calling *int86x()* or *intdosx()*. See these functions for more information.

DATA TYPES

```
struct SREGS {      unsigned int es;
      unsigned int cs;
      unsigned int ss;
      unsigned int ds;
};
```

SEE ALSO

int86, int86x, intdos, intdosx

SETJMP

SYNOPSIS

```
#include <setjmp.h> int          setjmp(jmp_buf buf)
```

DESCRIPTION

Setjmp() is used with *longjmp()* for non-local gotos. See *longjmp()* for further information.

EXAMPLE

```
#include        <stdio.h>
#include        <setjmp.h>
#include        <stdlib.h>

jmp_buf jb;

inner(void)
{
        longjmp(jb, 5);
}

main()
{
        int     i;

        if(i = setjmp(jb)) {
                printf("setjmp returned %d\n", i);
                exit(0);
        }
        printf("setjmp returned 0 - good\n");
        printf("calling inner...\n");
        inner();
        printf("inner returned - bad!\n");
}
```

RETURN VALUE

Setjmp() returns 0 after the real call, and non-zero if it apparently returns after a call to *longjmp()*.

SEE ALSO

longjmp

SETVBUF, SETBUF

SYNOPSIS

```
#include <stdio.h>
int      setvbuf(FILE * stream, char * buf,
         int mode, size_t size);
void     setbuf(FILE * stream, char * buf)
```

DESCRIPTION

The *setvbuf()* function allows the buffering behaviour of a STDIO stream to be altered. It supersedes the function *setbuf()* which is retained for backwards compatibility. The arguments to *setvbuf()* are as follows: **stream** designates the STDIO stream to be affected; **buf** is a pointer to a buffer which will be used for all subsequent I/O operations on this stream. If **buf** is null, then the routine will allocate a buffer from the heap if necessary, of size BUFSIZ as defined in *h*. **mode** may take the values _IONBF, to turn buffering off completely, _IOFBF, for full buffering, or _IOLBF for line buffering. Full buffering means that the associated buffer will only be flushed when full, while line buffering means that the buffer will be flushed at the end of each line or when input is requested from another STDIO stream. **size** is the size of the buffer supplied. By default, **stdout** and **stdin** are line buffered when associated with a terminal-like device, and full buffered when associated with a file.

If a buffer is supplied by the caller, that buffer will remain associated with that stream even over*fclose()*, *fopen()* calls until another *setvbuf()* changes it.

EXAMPLE

```
#include        <stdio.h>

char    buffer[8192];

main()
{
        int     i, j;

        /* set a large buffer for stdout */

        setvbuf(stdout, buffer, _IOFBF, sizeof buffer);
        for(i = 0 ; i != 2000 ; i++)
                if((i % 100) == 0)
                        printf("i = %4d\n", i);
                else
                        for(j = 0 ; j != 1000 ; j++)
```

```
                                continue;
}
```

NOTE

If the **buf** argument is null, then the size is ignored.

SEE ALSO

fopen, freopen, fclose

SET_VECTOR

SYNOPSIS

```
#include <intrpt.h> typedef interrupt void (*isr)();
isr set_vector(isr * vector, isr func);
```

DESCRIPTION

This routine allows an interrupt vector to be initialized. The first argument should be the address of the interrupt vector (not the vector number but the actual address) cast to a pointer to *isr*, which is a typedef'd pointer to an interrupt function. The second argument should be the function which you want the interrupt vector to point to. This must be declared using the *interrupt* type qualifier.

Not all compilers support this routine; the macros ROM_VECTOR, RAM_VECTOR and CHANGE_VECTOR are used with some processors, and are to be preferred even where set_vector is supported. See
h or the processor specific manual section to determine what is supported for a particular compiler.

The example shown sets up a vector for the DOS ctrl-BREAK interrupt.

EXAMPLE

```
#include        <signal.h>
#include        <stdlib.h>
#include        <intrpt.h>

static far interrupt void
brkintr(void)
{
        exit(-1);
}

#define BRKINT  0x23
#define BRKINTV ((far isr *)(BRKINT * 4))

void
set_trap(void)
{
        set_vector(BRKINTV, brkintr);
}
```

RETURN VALUE

The return value of *set_vector()* is the previous contents of the vector, if *set_vector()* is implemented as a function. If it is implemented as a macro, it has no return value.

SEE ALSO

di(), ei(), ROM_VECTOR, RAM_VECTOR, CHANGE_VECTOR

SIGNAL

SYNOPSIS

```
#include <signal.h> void (* signal)(int sig, void (*func)(int));
```

DESCRIPTION

Signal() provides a mechanism for catching control-C's (ctrl-BREAK for MS-DOS) typed on the console during I/O. Under CP/M the console is polled whenever an I/O call is performed, while for MS-DOS the polling depends on the setting of the BREAK command. If a control-C is detected certain action will be performed. The default action is to exit summarily; this may be modified with *signal()*. The **sig** argument to signal may at the present time be only SIGINT, signifying an interrupt condition. The func argument may be one of SIG_DFL, representing the default action i.e. to exit immediately, SIG_IGN, to ignore control-C's completely, or the address of a function which will be called with one argument, the number of the signal caught, when a control-C is seen. As the only signal supported is SIGINT, this will always be the value of the argument to the called function.

EXAMPLE

```
#include        <signal.h>
#include        <stdio.h>
#include        <setjmp.h>
#include        <stdlib.h>

jmp_buf jb;

void
catch(int c)
{
        longjmp(jb, 1);
}

main()
{
        int     i;

        if(setjmp(jb)) {
                printf("\n\nCaught signal\n");
                exit(0);
        }
        signal(SIGINT, catch);
        for(i = 0 ;; i++ ) {
                printf("%6d\r", i);
```

```
        }
}
```

SEE ALSO

exit

SIN

SYNOPSIS

```
#include  <math.h>
double    sin(double f);
```

DESCRIPTION

This function returns the sine function of its argument.

EXAMPLE

```
#include        <math.h>
#include        <stdio.h>

#define C       3.141592/180.0
main()
{
        double  i;

        for(i = 0 ; i <= 180.0 ; i += 10)
                printf("sin(%3.0f) = %f, cos = %f\n",
                        i, sin(i*C), cos(i*C));
}
```

SEE ALSO

cos, tan, asin, acos, atan

SPAWNL, SPAWNV, SPAWNVE

SYNOPSIS

```
#include <sys.h>
int      spawnl(char * n, char * argv0, ...);
int      spawnle(char * n, char * argv0, ...);
int      spawnv(char * n, char ** v)
int      spawnve(char * n, char ** v, char ** e)
int      spawnvp(char * n, char ** v)
int      spawnlp(char * n, char * argv0, ...);
```

DESCRIPTION

These function execute sub-programs under MS-DOS. The first argument is the program to execute. In the case of *spawnlp()* and *spawnvp()* this may be simply a name, without path or file type. The function will search the current PATH to locate the corresponding program, just as the DOS command line interpreter does. The other functions require a path name with file type, e.g. **C:\BIN\TEST.EXE**.

The remaining arguments provide the command line arguments. These are a list of character pointers (strings) terminated by a null pointer. This list can be explicit in the call (the *l* functions) or be a separate array pointed to by a char ** pointer (the *p* functions). In either case the first entry in the list is the nominal **argv[0]** which should appear in the called programs arguments. However since DOS does not pass this to the called program, it is in fact an unused placeholder.

The functions ending in *e* also take an environment pointer. This is a pointer to an array of char * strings, which will be used as the environment for the called program. The other functions use the global variable **environ** to provide the environment. See *getenv()* for more information on environment variables.

EXAMPLE

```
#include        <sys.h>

char *  args[] =
{
        “mem”,  /* this is required */
        “/c”,
        0
};

main()
{
        spawnlp(“mem”, “mem”, NULL);
```

```
        spawnv("c:\\dos\\mem.exe", args);
}
```

RETURN VALUE

Failure to execute the program returns -1; otherwise the return value is the exit value of the spawned program. By convention an exit value of zero means no error, and non-zero indicates some kind of error.

SEE ALSO

execl, execv

SPRINTF

SYNOPSIS

```
#include <stdio.h>
int      sprintf(char * buf, char * fmt, ...);
int      vsprintf(char * buf, char * fmt, va_list ap);
```

DESCRIPTION

Sprintf() operates in a similar fashion to *printf()*, except that instead of placing the converted output on the stdout stream, the characters are placed in the buffer at **buf**. The resultant string will be null-terminated, and the number of characters in the buffer will be returned. *Vsprintf* takes an argument pointer rather than a list of arguments.

RETURN VALUE

Sprintf() returns the number of characters placed into the buffer.

SEE ALSO

printf, fprintf, sscanf

SQRT

SYNOPSIS

```
#include  <math.h>
double    sqrt(double f)
```

DESCRIPTION

Sqrt() implements a square root function using Newton's approximation.

EXAMPLE

```
#include        <math.h>
#include        <stdio.h>

main()
{
        double  i;

        for(i = 0 ; i <= 20.0 ; i += 1.0)
                printf("square root of %.1f = %f\n", i, sqrt(i));
}
```

SEE ALSO

exp

SRAND

SYNOPSIS

```
#include <stdlib.h>
void srand(int seed)
```

DESCRIPTION

Srand() initializes the random number generator accessed by *rand()* with the given **seed**. This provides a mechanism for varying the starting point of the pseudo-random sequence yielded by *rand()*. On the z80, a good place to get a truly random seed is from the refresh register. Otherwise timing a response from the console will do, or just using the system time.

EXAMPLE

```
#include        <stdlib.h>
#include        <stdio.h>
#include        <time.h>

main()
{
        time_t  toc;
        int     i;

        time(&toc);
        srand((int)toc);
        for(i = 0 ; i != 10 ; i++)
                printf("%d\t", rand());
        putchar('\n');
}
```

SEE ALSO

rand

SSCANF

SYNOPSIS

```
#include <stdio.h>
int      sscanf(char * buf, char * fmt, ...);
int      vsscanf(char * buf, char * fmt, va_list ap);
```

DESCRIPTION

Sscanf() operates in a similar manner to *scanf()*, except that instead of the conversions being taken from stdin, they are taken from the string at **buf**.

SEE ALSO

scanf, fscanf, sprintf

STAT

SYNOPSIS

```
#include <stat.h>
int      stat(char * name, struct stat * statbuf)
```

DESCRIPTION

This routine returns information about the file by **name**. The information returned is operating system dependent, but may include file attributes (e.g. read only), file size in bytes, and file modification and/or access times. The argument **name** should be the name of the file, and may include path names under DOS, user numbers under CP/M, etc. The argument **statbuf** should be the address of a structure as defined in *stat.h* which will be filled in with the information about the file. The structure of *struct stat* is as follows:

```
{ short  st_mode; /* flags */
 long    st_atime;         /* access time */
 long    st_mtime;         /* modification time */
 long    st_size; /* file size */
};
```

The access and modification times (under DOS these are both set to the modification time) are in seconds since 00:00:00 Jan 1 1970. The function *ctime()* may be used to convert this to a readable value. The file size is self explanatory. The flag bits are as follows:

```
Flag     Meaning
```

S_IFMTmask for file type S_IFDIRfile is a directory S_IFREGfile is a regular file S_IREADfile is readable S_IWRITEfile is writeable S_IEXECfile is executable S_HIDDENfile is hidden S_SYSTEMfile is marked system S_ARCHIVEfile has been written to

EXAMPLE

```
#include        <stdio.h>
#include        <stat.h>
#include        <time.h>
#include        <stdlib.h>

main(argc, argv)
char ** argv;
{
        struct stat     sb;

        if(argc > 1) {
```

```
                if(stat(argv[1], &sb)) {
                        perror(argv[1]);
                        exit(1);
                }
                printf("%s: %ld bytes, modified %s", argv[1],
                        sb.st_size, ctime(&sb.st_mtime));
        }
        exit(0);
}
```

RETURN VALUE

Stat returns 0 on success, -1 on failure, e.g. if the file could not be found.

SEE ALSO

ctime, creat, chmod

STRCAT

SYNOPSIS

```
#include <string.h>
char *   strcat(char * s1, char * s2);
```

DESCRIPTION

This function appends (catenates) string **s2** to the end of string **s1**. The result will be null terminated. **S1** must point to a character array big enough to hold the resultant string.

EXAMPLE

```
#include        <string.h>
#include        <stdio.h>

main()
{
        char    buffer[256];
        char *  s1, * s2;

        strcpy(buffer, “Start of line”);
        s1 = buffer;
        s2 = “ ... end of line”;
        strcat(s1, s2);
        printf(“Length = %d\n”, strlen(buffer));
        printf(“string = \”%s\"\n", buffer);
}
```

RETURN VALUE

The value of **s1** is returned.

SEE ALSO

strcpy, strcmp, strncat, strlen

STRCHR

SYNOPSIS

```
#include <string.h>
char *   strchr(char * s, int c)
```

DESCRIPTION

Strchr() searches the string **s** for an occurence of the character **c**. If one is found, a pointer to that character is returned, otherwise NULL is returned.

RETURN VALUE

A pointer to the first match found, or NULL if the character does not exist in the string.

NOTE

Although the function takes an *int* argument for the character, only the lower 8 bits of the value are used.

SEE ALSO

strrchr, strlen, strcmp

STRCMP

SYNOPSIS

```
#include <string.h>
int      strcmp(char * s1, char * s2);
```

DESCRIPTION

Strcmp() compares its two string (null terminated) arguments and returns a signed integer to indicate whether **s1** is less than, equal to or greater than **s2**. The comparison is done with the standard collating sequence, which is that of the ASCII character set.

EXAMPLE

```
#include        <string.h>
#include        <stdio.h>

main()
{
        int     i;

        if((i = strcmp(“ABC”, “ABc”)) < 0)
                printf(“ABC is less than ABc\n”);
        else if(i > 0)
                printf(“ABC is greater than ABc\n”);
        else
                printf(“ABC is equal to ABc\n”);
}
```

RETURN VALUE

A signed integer less than, equal to or greater than zero.

NOTE

Other C implementations may use a different collating sequence; the return value is negative, zero or positive, i.e. do not test explicitly for -1 or 1.

SEE ALSO

strlen, strncmp, strcpy, strcat

STRCPY

SYNOPSIS

```
#include <string.h>
int      strcpy(char * s1, char * s2);
```

DESCRIPTION

This function copies a null-terminated string **s2** to a character array pointed to by **s1**. The destination array must be large enough to hold the entire string, including the null terminator.

EXAMPLE

```
#include        <string.h>
#include        <stdio.h>

main()
{
        char    buffer[256];
        char *  s1, * s2;

        strcpy(buffer, “Start of line”);
        s1 = buffer;
        s2 = “ ... end of line”;
        strcat(s1, s2);
        printf(“Length = %d\n”, strlen(buffer));
        printf(“string = \”%s\"\n", buffer);
}
```

RETURN VALUE

The destination buffer pointer **s1** is returned.

SEE ALSO

strncpy, strlen, strcat, strlen

STRLEN

SYNOPSIS

```
#include <string.h>
int      strlen(char * s);
```

DESCRIPTION

Strlen() returns the number of characters in the string **s**, not including the null terminator.

EXAMPLE

```
#include        <string.h>
#include        <stdio.h>

main()
{
        char    buffer[256];
        char *  s1, * s2;

        strcpy(buffer, “Start of line”);
        s1 = buffer;
        s2 = “ ... end of line”;
        strcat(s1, s2);
        printf(“Length = %d\n”, strlen(buffer));
        printf(“string = \”%s\"\n", buffer);
}
```

STRNCAT

SYNOPSIS

```
#include <string.h>
char *   strncat(char * s1, char * s2, sizt_t n);
```

DESCRIPTION

This function appends (catenates) string **s2** to the end of string **s1**. At most **n** characters will be copied, and the result will be null terminated. **S1** must point to a character array big enough to hold the resultant string.

EXAMPLE

```
#include        <string.h>
#include        <stdio.h>

main()
{
        char    buffer[256];
        char *  s1, * s2;

        strcpy(buffer, “Start of line”);
        s1 = buffer;
        s2 = “ ... end of line”;
        strncat(s1, s2, 5);
        printf(“Length = %d\n”, strlen(buffer));
        printf(“string = \”%s\"\n", buffer);
}
```

RETURN VALUE

The value of **s1** is returned.

SEE ALSO

strcpy, strcmp, strcat, strlen

STRNCMP

SYNOPSIS

```
#include <string.h>
int      strncmp(char * s1, char * s2, size_t n);
```

DESCRIPTION

Strcmp() compares its two string (null terminated) arguments, up to a maximum of **n** characters, and returns a signed integer to indicate whether **s1** is less than, equal to or greater than **s2**. The comparison is done with the standard collating sequence, which is that of the ASCII character set.

RETURN VALUE

A signed integer less than, equal to or greater than zero.

NOTE

Other C implementations may use a different collating sequence; the return value is negative, zero or positive, i.e. do not test explicitly for -1 or 1.

SEE ALSO

strlen, strcmp, strcpy, strcat

STRNCPY

SYNOPSIS

```
#include <string.h>
int      strncpy(char * s1, char * s2, size_t n);
```

DESCRIPTION

This function copies a null-terminated string **s2** to a character array pointed to by **s1**. At most **n** characters are copied. If string **s2** is longer than **n** then the destination string will not be null terminated. The destination array must be large enough to hold the entire string, including the null terminator.

EXAMPLE

```
#include        <string.h>
#include        <stdio.h>

main()
{
        char    buffer[256];
        char *  s1, * s2;

        strncpy(buffer, “Start of line”, 6);
        s1 = buffer;
        s2 = “ ... end of line”;
        strcat(s1, s2);
        printf(“Length = %d\n”, strlen(buffer));
        printf(“string = \”%s\"\n", buffer);
}
```

RETURN VALUE

The destination buffer pointer **s1** is returned.

SEE ALSO

strcpy, strcat, strlen, strcmp

STRRCHR

SYNOPSIS

```
#include <string.h>
char *   strrchr(char * s, int c)
```

DESCRIPTION

Strrchr() is similar to *strchr()* but searches from the *end* of the string rather than the beginning, i.e. it locates the *last* occurrence of the character **c** in the null-terminated string **s**. If successful it returns a pointer to that occurrence, otherwise it returns NULL.

RETURN VALUE

A pointer to the character, or NULL if none is found.

SEE ALSO

strchr, strlen, strcmp, strcpy, strcat

SYSTEM

SYNOPSIS

```
#include <sys.h>
int      system(char * s)
```

DESCRIPTION

When executed under MS-DOS *system()* will pass the argument string to the command processor, located via the environment string COMSPEC, for execution. The exit status of the command processor will be returned from the call to *system()*. Unfortunately this does not have any relation to the exit value returned by the command executed by the command processor. Use one of the *spawn()* functions if you need to test exit status.

The example sets the baud rate on COM1.

EXAMPLE

```
#include        <stdlib.h>
#include        <stdio.h>
#include        <time.h>

main(argc, argv)
char ** argv;
{
        /* set the baud rate on com1 */

        system("MODE COM1:96,N,8,1,P");
}
```

SEE ALSO

spawnl, spawnv

TAN

SYNOPSIS

```
#include <math.h>
double   tan(double f);
```

DESCRIPTION

This is the tangent function.

EXAMPLE

```
#include        <math.h>
#include        <stdio.h>

#define C       3.141592/180.0
main()
{
        double  i;

        for(i = 0 ; i <= 180.0 ; i += 10)
                printf("tan(%3.0f) = %f\n",
                        i, tan(i*C));
}
```

SEE ALSO

sin, cos, asin, acos, atan

TIME

SYNOPSIS

```
#include  <time.h>
time_t    time(time_t * t)
```

DESCRIPTION

This function returns the current time in seconds since 00:00:00 on Jan 1, 1970. If the argument **t** is non-null, the same value is stored into the object pointed to by **t**. The accuracy of this function is naturally dependent on the operating system having the correct time. This function does not work under CP/M 2.2 but does work under MS-DOS and CP/M+.

EXAMPLE

```
#include        <stdio.h>
#include        <time.h>

main()
{
        time_t  clock;

        time(&clock);
        printf("%s", ctime(&clock));
}
```

SEE ALSO

ctime, gmtime, localtime, asctime

TOLOWER, TOUPPER, TOASCII

SYNOPSIS

```
#include <ctype.h> char toupper(int c);
char tolower(int c);
char toascii(int c);
char     c;
```

DESCRIPTION

Toupper() converts its lower case alphabetic argument to upper case, *tolower()* performs the reverse conversion, and *toascii()* returns a result that is guaranteed in the range 0-0177. *Toupper()* and *tolower* return their arguments if it is not an alphabetic character.

SEE ALSO

islower, isupper, isascii et. al.

UNGETC

SYNOPSIS

```
#include <stdio.h>
int      ungetc(int c, FILE * stream)
```

DESCRIPTION

Ungetc() will attempt to push back the character **c** onto the named **stream**, such that a subsequent *getc()* operation will return the character. At most one level of pushback will be allowed, and if the **stream** is not buffered, even this may not be possible. EOF is returned if the *ungetc()* could not be performed.

SEE ALSO

getc

UNLINK

SYNOPSIS

```
#include <unixio.h>
int      unlink(char * name)
```

DESCRIPTION

Unlink() will remove (delete) the named file, that is erase the file from its directory. See *open()* for a description of the file name construction. Zero will be returned if successful, -1 if the file did not exist or it could not be removed. The ANSI function *remove()* is preferred to *unlink()*.

SEE ALSO

open, close, rename, remove

VA_START, VA_ARG, VA_END

SYNOPSIS

```
#include <stdarg.h>
void     va_start(va_list ap, parmN);
type     va_arg(ap, type);
void     va_end(va_list ap);
```

DESCRIPTION

These macros are provided to give access in a portable way to parameters to a function represented in a prototype by the ellipsis symbol (...), where type and number of arguments supplied to the function are not known at compile time. The rightmost parameter to the function (shown as *parmN*) plays an important role in these macros, as it is the starting point for access to further parameters. In a function taking variable numbers of arguments, a variable of type **va_list** should be declared, then the macro **va_start** invoked with that variable and the name of *parmN*. This will initialize the variable to allow subsequent calls of the macro **va_arg** to access successive parameters. Each call to **va_arg** requires two arguments; the variable previously defined and a type name which is the type that the next parameter is expected to be. Note that any arguments thus accessed will have been widened by the default conventions to *int*, *unsigned int* or *double*. For example if a character argument has been passed, it should be accessed by va_arg(ap, int) since the char will have been widened to *int*. An example is given below of a function taking one integer parameter, followed by a number of other parameters. In this example the function expects the subsequent parameters to be pointers to char, but note that the compiler is not aware of this, and it is the programmers responsibility to ensure that correct arguments are supplied.

EXAMPLE

```
#include        <stdio.h>
#include        <stdarg.h>

pf(int a, ...)
{
        va_list ap;

        va_start(ap, a);
        while(a—)
                puts(va_arg(ap, char *));
        va_end(ap);
}

main()
{
```

```
        pf(3, “Line 1", ”line 2", “line 3");
}
```

WRITE

SYNOPSIS

```
#include <unixio.h>
int      write(int fd, void * buf, size_t cnt)
```

DESCRIPTION

Write() will write from the buffer at **buf** up to **cnt** bytes to the file associated with the file descriptor **fd**. The number of bytes actually written will be returned. EOF or a value less than **cnt** will be returned on error. In any case, any return value not equal to **cnt** should be treated as an error (cf. *read()*).

EXAMPLE

```
#include        <unixio.h>

main()
{
        write(1, "A test string\r\n", 15);
}
```

SEE ALSO

open, close, read

_DOS_GETFTIME

SYNOPSIS

```
#include <dos.h>
int      _dos_getftime(int fd, unsigned short * date, unsigned short *
time);
```

DESCRIPTION

This function takes as its arguments a DOS file handle, and two pointers to unsigned shorts, into which will be stored the DOS date and time values associated with the file identified by the file handle. Refer to a DOS programming manual for information about the format of these values. The standard function *stat()* is preferred for determining the modification date of a file.

EXAMPLE

```
#include        <stdio.h>
#include        <dos.h>
#include        <stdlib.h>

main(argc, argv)
char ** argv;
{
        FILE *  fp;
        unsigned short  i, j;

        if(argc == 1)
                exit(1);
        if(!(fp = fopen(argv[1], “r”))) {
                perror(argv[1]);
                exit(1);
        }
        _dos_getftime(fileno(fp), &i, &j);
        printf(“date = %u, time =%u\n”, i, j);
}
```

SEE ALSO

stat

_EXIT

SYNOPSIS

```
#include <stdlib.h> void        _exit(int status)
```

DESCRIPTION

This function will cause an immediate exit from the program, without the normal flushing of stdio buffers that is performed by *exit()*. The argument is used as the program exit value for DOS.

EXAMPLE

```
main()
{
                        _exit(-1);
}
```

RETURN VALUE

None; the function will never return.

SEE ALSO

exit

_GETARGS

SYNOPSIS

```
#include <sys.h>
char ** _getargs(char * buf, char * name)
extern int _argc_;
```

DESCRIPTION

This routine performs I/O redirection (CP/M only) and wild card expansion. Under MS-DOS I/O redirection is performed by the operating system. It is called from startup code to operate on the command line if the -R option is used to the C command, but may also be called by user-written code. If the **buf** argument is null, it will read lines of text from standard input. If the standard input is a terminal (usually the console) the **name** argument will be written to the standard error stream as a prompt. If the **buf** argument is not null, it will be used as the source of the string to be processed. The returned value is a pointer to an array of strings, exactly as would be pointed to by the argv argument to the main() function. The number of strings in the array may be obtained from the global **_argc_.**

There will be one string in the array for each word in the buffer processed. Quotes, either single (‘) or double (“) may be used to include white space in ”words". If any wild card characters (? or *) appear in a non-quoted word, it will be expanded into a string of words, one for each file matching the word. The usual CP/M and DOS conventions are followed for this expansion. On CP/M any occurence of the redirection characters > and < outside quotes will be handled in the following manner:

> **name** will cause standard output to be redirected to the file **name**.

< **name** will cause standard input to be redirected from the file **name**.

> **name** will cause standard output to append to file **name**.

White space is optional between the > or < character and the file name, however it is an error for a redirection character not to be followed by a file name. It is also an error if a file cannot be opened for input or created for output. An append redirection () will create the file if it does not exist. If the source of text to be processed is standard input, several lines may be supplied by ending each line (except the last) with a backslash (\). This serves as a continuation character. Note that the newline following the backslash is ignored, and not treated as white space.

EXAMPLE

```
#include <sys.h>

main(argc, argv)
char ** argv;
{
```

```
    extern char ** _getargs();
    extern int     _argc_;

    if(argc == 1) {   /* no arguments */
    argv = _getargs(0, “myname”);
    argc = _argc_;
    }
    .
    .
    .
}
```

RETURN VALUE

A pointer to an array of strings.

NOTE

This routine is not usable in a ROM based system.

_XMS_ALLOC, _XMS_ALLOC_BYTES

SYNOPSIS

```
#include    <xms.h>
_XMS_HANDLE       _xms_alloc(_XMS_SIZE_T size)
_XMS_HANDLE       _xms_alloc_bytes(unsigned long size)
```

DESCRIPTION

The *_xms_alloc()* and *_xms_alloc_bytes()* functions are used to allocated blocks of XMS. The return value is an XMS handle, or 0 if no memory could be allocated. The _XMS_HANDLE returned is used by other XMS routines to access the allocated block. The argument to *_xms_alloc()* is a block size, in 1Kb increments. *_xms_alloc_bytes()* takes a block size in bytes, and will allocate an XMS block large enough to hold the specified number of bytes.

EXAMPLE

```
#include       <stdio.h>
#include       <xms.h>

main()
{
        _XMS_HANDLE blk1, blk2;

        blk1 = _xms_alloc(1);           /* get 1Kb */
        blk2 = _xms_alloc_bytes(2048);  /* get 2Kb */
        printf("blk1 = %4.4X, blk2 = %4.4X\n",
                blk1, blk2);
        if (blk1 != 0)
                _xms_dispose(blk1);
        if (blk2 != 0)
                _xms_dispose(blk2);
}
```

RETURN VALUE

An _XMS_HANDLE, non zero on success, 0 on failure or if XMS is not available.

DATA TYPES

typedef unsigned int _XMS_HANDLE; typedef unsigned int _XMS_SIZE_T;

SEE ALSO

_xms_dispose, _xms_malloc, _xms_resize, _xms_resize_bytes

_XMS_DISPOSE

SYNOPSIS

```
#include    <xms.h>
int     _xms_dispose(_XMS_HANDLE handle)
```

DESCRIPTION

_xms_dispose() is used to free XMS blocks which have been allocated by *_xms_alloc()*, *_xms_alloc_bytes()*, *_xms_resize()* or *_xms_resize_bytes()*. Only XMS handles which have been obtained using these routines should be passed to *_xms_dispose()*.

EXAMPLE

```
#include    <stdio.h>
#include    <xms.h>

main()
{
        _XMS_HANDLE   handle;

        handle = _xms_alloc(10);  /* get 10Kb */
        printf("handle = %4.4X\n", handle);
        if (handle)
                _xms_dispose(handle);
}
```

RETURN VALUE

Integer, 1 on success, 0 on failure.

DATA TYPES

typedef unsigned int _XMS_HANDLE;

NOTE

Never attempt to *_xms_dispose()* handles from pointers obtained using *_xms_calloc()*, *_xms_malloc()*, *_xms_realloc()* or *_xms_sbrk*.

SEE ALSO

_xms_alloc, _xms_alloc_bytes, _xms_resize, _xms_resize_bytes

_XMS_DRIVER

SYNOPSIS

```
#include    <xms.h>
_XMS_DRIVER      _xms_driver(void)
```

DESCRIPTION

_xms_driver() returns a pointer to the XMS driver entry point. The return value is a 32 bit segment:offset format pointer, suitable for use with the *_driver()* and *_driverx()* functions. If XMS is not installed, a NULL pointer will be returned.

EXAMPLE

```
#include       <stdio.h>
#include       <xms.h>

union {
        _XMS_DRIVER   ptr;
        struct {
                unsigned short  ofs;
                unsigned short  seg;
        } w;
} xdriver;

main()
{
        xdriver.ptr = _xms_driver();
        printf("Entry point = %4.4X:%4.4X\n",
                xdriver.w.seg, xdriver.w.ofs);
}
```

RETURN VALUE

A 32 bit segment:offset pointer which is the entry point for the XMS driver, or NULL if XMS is not installed.

DATA TYPES

typedef far void * _XMS_DRIVER;

SEE ALSO

_driver, _driverx, _xms_installed

_XMS_FLUSH

SYNOPSIS

```
#include    <xms.h>
void    _xms_flush(_XMS_HANDLE handle)
```

DESCRIPTION

The *_xms_flush()* routine writes all “dirty” cache blocks belonging to the specified XMS handle back to XMS. This routine may be used to ensure that all writes via the “high level” XMS support routines have actually been stored to XMS. All XMS reads and writes performed using the following routines are cached:
_xms_calloc()
_xms_get()
_xms_memmove()
_xms_memset()
_xms_put()
_xms_read()
_xms_write()
_xms_zalloc()

EXAMPLE

```
#include       <stdio.h>
#include       <xms.h>

#define SIZE   8192

main()
{
        _XMS_HANDLE    handle;

        handle = _xms_alloc_bytes(SIZE);
        if (handle) {
                _xms_memset(handle, 0, 0xFF, SIZE);
                printf(“Flush handle %4,4X\n”, handle);
                _xms_flush(handle);
                printf(“Done!\n”);
                _xms_dispose(handle);
        } else
```

```
                printf("Unable to use XMS!\n");
}
```

DATA TYPES

typedef unsigned int _XMS_HANDLE;

SEE ALSO

_xms_calloc, _xms_get, _xms_memmove, _xms_memset, _xms_put, _xms_read, _xms_write, _xms_zalloc

_XMS_INFO

SYNOPSIS

```
#include    <xms.h>
int     _xms_info(_XMS_HANDLE handle, struct _XMS_INFO * info)
```

DESCRIPTION

The *_xms_info()* call is used to determine the lock count for an XMS block, and the number of XMS handles which are available.

EXAMPLE

```
#include    <stdio.h>
#include    <xms.h>

main()
{
        _XMS_HANDLE       handle;
        unsigned long     addr;
        struct _XMS_INFO  info;

        handle = _xms_alloc(1);  /* grab 1K block */
        if (handle) {
                addr = _xms_lock(handle);
                _xms_info(handle, &info);
                printf("Handle  = %4.4X\n", handle);
                printf("Address = %8.8lX\n", addr);
                printf("Lock count = %d\n", info.lock_count);
                printf("Free handles = %d\n", info.free_handles);
                _xms_unlock(handle);
                _xms_dispose(handle);
        } else
                printf("Couldn't allocate XMS\n");
}
```

RETURN VALUE

Integer, 1 on success, 0 on failure.

DATA TYPES

```
typedef unsigned int _XMS_HANDLE; struct _XMS_INFO {
  unsigned char lock_count;
  unsigned char free_handles;
};
```

SEE ALSO

_xms_lock, _xms_unlock

_XMS_INSTALLED

SYNOPSIS

```
#include    <xms.h>
int     _xms_installed(void)
```

DESCRIPTION

_xms_installed() will return 1 if XMS is present, otherwise it will return 0. This function is generally used to determine whether an XMS or conventional memory allocation scheme should be used.

EXAMPLE

```
#include       <stdio.h>
#include       <xms.h>

main()
{
        if (_xms_installed())
                printf("Using XMS\n");
        else
                printf("XMS not present\n");
}
```

RETURN VALUE

An integer, 1 if XMS is present, otherwise 0.

SEE ALSO

_xms_driver

_XMS_LOCK

SYNOPSIS

```
#include    <xms.h>
unsigned long    _xms_lock(_XMS_HANDLE handle)
```

DESCRIPTION

_xms_lock() locks an XMS block, thereby preventing it from being moved, and returns its physical address. The physical address returned is only valid while the block is locked. A locked block should be unlocked as soon as possible, as it may prevent other blocks from being resized and cause memory fragmentation. A lock count is maintained for extended memory blocks, the same number of *_xms_lock()* calls as *_xms_unlock()* calls need to be made in order to unlock a block. The *_xms_info()* function may be used to determine the lock count for an XMS block.

EXAMPLE

```
#include    <stdio.h>
#include    <xms.h>

main()
{
        _XMS_HANDLE    handle;
        unsigned long  addr;

        handle = _xms_alloc(1);  /* grab 1K block */
        if (handle) {
                addr = _xms_lock(handle);
                printf("Block address = %8.8lX\n", addr);
                _xms_unlock(handle);
                _xms_dispose(handle);
        } else
                printf("Couldn't allocate XMS\n");
}
```

RETURN VALUE

Unsigned long, the physical address of the block on success, 0 on failure.

DATA TYPES

typedef unsigned int _XMS_HANDLE;

SEE ALSO

_xms_info, _xms_unlock

_XMS_MEMAVAIL

SYNOPSIS

```
#include    <xms.h>
int     _xms_memavail(struct _XMS_FREE * mem)
```

DESCRIPTION

The *_xms_memavail()* function returns information about the amount of free XMS in the struct _XMS_FREE provided by the user. All values in struct _XMS_FREE represent quantities of memory in Kilobytes. For example, a "maxblock" value of 128 means that the largest XMS block which may be allocated is 128Kb.

EXAMPLE

```
#include       <stdio.h>
#include       <xms.h>

main()
{
        struct _XMS_FREE  mem;

        if (_xms_memavail(&mem)) {
                printf("XMS available = %uKb", mem.total);
                printf("Largest block = %uKb", mem.maxblock);
                printf("XMS used = %uKb", mem.used);
        } else
                printf("XMS not available\n");
}
```

RETURN VALUE

Integer, 1 on success, 0 on failure or if XMS is not present.

DATA TYPES

typedef unsigned int _XMS_SIZE_T; struct _XMS_FREE {
 _XMS_SIZE_T maxblock;
 _XMS_SIZE_T total;
 _XMS_SIZE_T used;
};

SEE ALSO

_xms_info, _xms_version

_XMS_MEMSET

SYNOPSIS

```
#include    <xms.h>
int     _xms_memset(_XMS_HANDLE dst, unsigned long ofs, int ch,
unsigned long size);
```

DESCRIPTION

_xms_memset() is used to set a block of XMS to a specified value. "Dst" is the destination handle, "ofs" is the offset within the XMS block, "ch" is the fill character, and "size" is the number of bytes to clear. All writes are performed via the XMS cache using the *_xms_write()* routine.

RETURN VALUE

Integer, 1 on success, 0 on failure.

DATA TYPES

typedef unsigned int _XMS_HANDLE;

SEE ALSO

_xms_memmove, _xms_write, _xms_zalloc

_XMS_MOVE

SYNOPSIS

```
#include    <xms.h>
int     _xms_move(struct _XMS_MOVE * move)
```

DESCRIPTION

_xms_move() is the C interface to the XMS driver "Move Extended Memory Block" call. Although this function is intended to move blocks of data between conventional DOS memory and XMS, it can also move data within conventional DOS memory and within XMS. XMS addresses are encoded as an XMS handle plus a 32 bit offset. Conventional memory addresses are encoded as a zero handle plus a 32 bit pointer.

The parameters for the move are taken from the _XMS_MOVE structure. The length of the transfer must be even. Performance is improved if the blocks are word-aligned, or double word-aligned on 80386 and 80486 machines. Transfers of overlapping blocks should not be attempted using this routine. The *_xms_memmove()* function provides a safe way of transferring overlapping blocks.

EXAMPLE

```
#include       <stdio.h>
#include       <string.h>
#include       <stdlib.h>
#include       <xms.h>

main()
{
        _XMS_HANDLE      handle;
        struct _XMS_MOVE move;
        char             buf[256];

        handle = _xms_alloc_bytes(sizeof(buf));
        if (!handle) {
                printf("Couldn't get XMS\n");
                exit(1);
        }
        strcpy(buf, "** TEST STRING - 0123456789 **");
        move.count = sizeof(buf);
        move.src_handle = 0;  /* conventional memory */
        move.src.ptr = buf;
        move.dst_handle = handle;  /* XMS */
        move.dst.offset = 0;
        if (!_xms_move(&move)) {
```

```
                printf("Store to XMS failed!\n");
                _xms_dispose(handle);
                exit(1);
        }
        strcpy(buf, "THIS IS A DIFFERENT STRING!!!!");
        move.count = sizeof(buf);
        move.src_handle = handle;  /* XMS */
        move.src.offset = 0;
        move.dst_handle = 0;  /* conventional memory */
        move.dst.ptr = buf;
        if (!_xms_move(&move)) {
                printf("Retrieve from XMS failed!\n");
                _xms_dispose(handle);
                exit(1);
        }
        printf("Retrieved '%s' from XMS\n", buf);
        _xms_dispose(handle);
        exit(0);
}
```

RETURN VALUE

Integer, 1 on success, 0 on failure or if XMS is not present.

DATA TYPES

```
union ptr_ofs {  far void *    ptr;
  unsigned long offset;
};

struct _XMS_MOVE {
  unsigned long  count;
  _XMS_HANDLE    src_handle;
  union ptr_ofs  src;
  _XMS_HANDLE    dst_handle;
  union ptr_ofs  dst;
};
```

SEE ALSO

_xms_memmove, _xms_read, _xms_write

_XMS_READ

SYNOPSIS

```
#include    <xms.h>
int     _xms_read(_XMS_HANDLE src, unsigned int size,
                  unsigned long ofs, far void * dst)
```

DESCRIPTION

_xms_read()
is used to read from XMS via the cache. "Size" bytes are transferred to the destination address from the specified offset within the source XMS handle. Odd length transfers are permitted, as _xms_read() uses the XMS cache to hide the limitations of the XMS driver.

EXAMPLE

```
#include       <stdio.h>
#include       <xms.h>

static char    test[] = "This is a test string!";

main()
{
        _XMS_HANDLE   handle;
        char          buf[256];

        handle = _xms_alloc_bytes(sizeof(buf));
        if (handle) {
                _xms_write(handle, sizeof(test), 0, test);
                 _xms_read(handle, sizeof(test), 0, buf);
                 printf("buf = '%s'\n", buf);
                 _xms_dispose(handle);

        } else
                printf("Unable to allocate XMS\n");
}
```

RETURN VALUE

Integer, 1 on success, 0 on failure.

DATA TYPES

typdef unsigned int _XMS_HANDLE;

SEE ALSO

_xms_write

_XMS_RESIZE, _XMS_RESIZE_BYTES

SYNOPSIS

```
#include    <xms.h>
int     _xms_resize(_XMS_HANDLE handle, _XMS_SIZE_T new_size)
int     _xms_resize_bytes(_XMS_HANDLE handle, unsigned long
new_size)

unsigned char    _xms_driver_realloc;
```

DESCRIPTION

The *_xms_resize()* and *_xms_resize_bytes()* functions are used to change the size of XMS blocks. If the new block is the same size or larger, all data is copied. If the new size is smaller, all data at the upper end of the block is lost. These functions both return an _XMS_HANDLE which is the new handle for the block. With some XMS drivers this will be the same as the original handle. If the *_xms_driver_realloc* flag is set, *_xms_resize()* and *_xms_resize_bytes()* will fail if the XMS driver "realloc" call cannot be used and the same handle cannot be returned.

EXAMPLE

```
#include       <stdio.h>
#include       <xms.h>

main()
{
        _XMS_HANDLE    handle, new;

        handle = _xms_alloc(32);   /* grab 32Kb */
        if (handle) {
                new = _xms_resize(handle, 16);
                printf("old handle = %4.4X\n", handle);
                printf("new handle = %4.4X\n", new);
                if (new)
                        _xms_dispose(new);
                else
                        _xms_dispose(handle);
        } else
                printf("Unable to allocate XMS\n");
}
```

RETURN VALUE

An _XMS_HANDLE on success, 0 on failure.

DATA TYPES

typedef unsigned int _XMS_HANDLE; typedef unsigned int _XMS_SIZE_T;

NOTE

If these functions fail, the original block will not be affected and the old handle will still be valid.

_XMS_UNLOCK

SYNOPSIS

```
#include    <xms.h>
int     _xms_unlock(_XMS_HANDLE handle)
```

DESCRIPTION

_xms_unlock() is used to unlock XMS blocks which have previously been locked by one or more *_xms_lock()* calls. Lock counts are maintained for XMS blocks, so *_xms_unlock()* and *_xms_lock()* calls need to be balanced. The *_xms_info()* function may be used to determine the lock count for an XMS block.

EXAMPLE

```
#include    <stdio.h>
#include    <xms.h>

main()
{
        _XMS_HANDLE    handle;
        unsigned long  addr;

        handle = _xms_alloc(1);  /* grab 1K block */
        if (handle) {
                addr = _xms_lock(handle);
                printf("Block address = %8.8lX\n", addr);
                _xms_unlock(handle);
                _xms_dispose(handle);
        } else
                printf("Couldn't allocate XMS\n");
}
```

RETURN VALUE

Integer, 1 on success, 0 on failure.

DATA TYPES

typedef unsigned int _XMS_HANDLE;

SEE ALSO

_xms_info, _xms_lock

_XMS_VERSION

SYNOPSIS

```
#include    <xms.h>
int     _xms_version(struct _XMS_VERSION * vers)
```

DESCRIPTION

This function returns information about the XMS driver in the _XMS_VERSION structure provided by the user. The return value is 1 on success, 0 if XMS is not installed or driver information could not be obtained. The “hma_exists” flag in struct _XMS_VERSION is used to determine whether the MS-DOS High Memory Area at segment FFFF exists.

EXAMPLE

```
#include       <stdio.h>
#include       <xms.h>

main()
{
        struct _XMS_VERSION  vi;

        if (_xms_version(&vi)) {
                printf(“XMS V%X.%2.2X\n”, vi.ver_major, vi.ver_minor);
                printf(“Driver revision %X.%2.2X\n”, vi.rev_major,
vi.rev_minor);
                if (vi.hma_exists)
                        printf(“HMA exists\n”);
                else
                        printf(“No HMA\n”);
        } else
                printf(“XMS not available\n”);
}
```

RETURN VALUE

1 on success, 0 on failure or if XMS is not installed.

DATA TYPES

```
struct _XMS_VERSION {  unsigned char ver_major;
  unsigned char ver_minor;
  unsigned char rev_major;
  unsigned char rev_minor;
```

```
  unsigned char hma_exists;
};
```

SEE ALSO

_xms_info, _xms_memavail

_XMS_WRITE

SYNOPSIS

```
#include    <xms.h>
int     _xms_write(_XMS_HANDLE dst, unsigned int size,
                        unsigned long ofs, far void * src)
```

DESCRIPTION

_xms_write()
is used to write to XMS via the cache. "Size" bytes are transferred from the source address to the specified offset within then destination XMS block. Odd length transfers are permitted, as _xms_write() uses the XMS cache to hide the limitations of the XMS driver.

EXAMPLE

```
#include       <stdio.h>
#include       <xms.h>

static char    test[] = "This is a test string!";

main()
{
        _XMS_HANDLE   handle;
        char          buf[256];

        handle = _xms_alloc_bytes(sizeof(buf));
        if (handle) {
                _xms_write(handle, sizeof(test), 0, test);
                 _xms_read(handle, sizeof(test), 0, buf);
                 printf("buf = '%s'\n", buf);
                 _xms_dispose(handle);

        } else
                printf("Unable to allocate XMS\n");
}
```

RETURN VALUE

Integer, 1 on success, 0 on failure.

DATA TYPES

typedef unsigned int _XMS_HANDLE;

SEE ALSO

_xms_flush, _xms_read

_XMS_ZALLOC

SYNOPSIS

```
#include    <xms.h>
_XMS_HANDLE   _xms_zalloc(unsigned int n_elem, unsigned int size)
```

DESCRIPTION

This function will allocate and clear a block of XMS large enough to hold an array of “n_elem” elements, each of “size” bytes. The block is cleared via a call to *_xms_memset()*.

EXAMPLE

```
#include       <stdio.h>
#include       <xms.h>

#define COUNT  32

main()
{
        _XMS_HANDLE    handle;

        handle = _xms_zalloc(COUNT, sizeof(int));
        if (handle) {
                printf(“handle = %4.4X\n”, handle);
                _xms_dispose(handle);
        } else
                printf(“Couldn’t allocate XMS\n”);
}
```

RETURN VALUE

Integer, 1 on success, 0 on failure.

DATA TYPES

typedef unsigned int _XMS_HANDLE;

Appendix B - Library Functions

B

Index

!

32
    bit, 64
80186, 56
80286, 56
80386, 56
80486, 56
8086, 55, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 100
80C188EB, 87
_DOS_GETFTIME, 309
_EXIT, 310
_GETARGS, 311
_XMS_ALLOC, _XMS_ALLOC_BYTES, 313
_XMS_DISPOSE, 315
_XMS_DRIVER, 316
_XMS_FLUSH, 317
_XMS_INFO, 319
_XMS_INSTALLED, 321
_XMS_LOCK, 322
_XMS_MEMAVAIL, 323
_XMS_MEMSET, 325
_XMS_MOVE, 326, 327
_XMS_READ, 328, 329
_XMS_RESIZE, _XMS_RESIZE_BYTES, 330, 331
_XMS_UNLOCK, 332
_XMS_VERSION, 333
_XMS_WRITE, 335
_XMS_ZALLOC, 337

A

ACOS, 153
address
    arithmetic, 78
    link, 49
    load, 49, 50
argc, 78
argument
    passing, 45
arguments
    to, 44
argv, 77, 78
arrays
    displaying, 84
as86
    options, 55, 56, 57
    pseudo, 67
    pseudo-ops, 61, 62, 63, 64, 65, 66, 67, 68, 69
ASCTIME, 154, 155
ASIN, 156
assembler, 55, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76
    compatibility, 55
    interface, 46
    listing, 56
    output, 56
ASSERT, 157
ATAN, 158
ATEXIT, 159
ATOF, 160
ATOI, 161
ATOL, 162
AUTOEXEC.BAT, 3

B

baud rate , 96
bp, 44
branch, 57
BSEARCH, 163

C

CALLOC, 165
CEIL, 166
CGETS, 167
CHDIR, 168
CHDRV, 169
CHMOD, 170
CLOSE, 171
CLRERR, CLREOF, 172
COM port, 96
Command
    line, 77, 78
compiler driver
    command line, 3
CONFIG.SYS, 3
constants, 59
COS, 173
COSH, SINH, TANH, 174
CPUTS, 175
CREAT, 176
cross
    reference, 57
CTIME, 177

D

data
    representation, 48
DI, EI, 178
DIV, 179
Download , 96
DRIVER, 180, 181
DUP, 182
DUP2, 183

E

environment variables
    PATH, 3
    TEMP, 1
EPROM , 87
EPROM programmer, 95
error
    messages, 72, 73, 75
EXIT, 184
EXP, 185
expressions, 59
Extern
    functions, 81

F

FABS, 186
far
    call, 57
FARMALLOC, FARFREE, FARREAL-
LOC, 187
FARMEMCPY, FARMEMSET, 189
FCLOSE, 190
FDOPEN, 191
FEOF, FERROR, 192
FFIRST, FNEXT, 193
FFLUSH, 194
FGETC, 195
FGETS, 196
FILENO, 197
fixups, 53
FLOOR, 198
FOPEN, 199
FPRINTF, 201, 202
FPUTC, 203
FPUTS, 204
frame
    pointer, 84
FREAD, 205

FREE, 206
FREOPEN, 207
FREXP, 208
FSCANF, 209
FSEEK, 210, 211
FTELL, 212
function
    prototypes, 45
FWRITE, 213

G

GETC, 214
GETCH, GETCHE, UNGETCH, 215
GETCHAR, 216
GETCWD, 217
GETDRV, 218
GETENV, 219
GETFREEMEM, 220
GETIVA, 221
GETS, 222
getting started, 5
GETW, 223
GMTIME, 224, 225

H

heap, 45

I

include
    files, 57
INSTALL program, 1
installation, 1, 2, 3, 87, 95
    custom, 2, 91
installation key, 2, 95
INT86, INT86X, 226, 227
INTDOS, INTDOSX, 228, 229
Intel
    assembler, 55
    object, 56
inter-segment
    call, 57
interrupt vectors, 88, 94
ISALNUM, ISALPHA, ISDIGIT, ISLOWER et. al., 230, 231
ISATTY, 232
ISNEC98, 233

J

jump
    optimization, 55

K

KBHIT, 234

L

LDEXP, 235
LDIV, 236
Library
    functions, 81
LINK, 3
link address, 100
linker, 99, 100, 102, 104, 106, 108
linking, 48, 49, 51, 53
listing
    width, 56
load address, 100
local
    data, 44
    symbols, 56
LOCALTIME, 237
LOG, LOG10, 239
LONGJMP, 240, 241
LSEEK, 242

M

main(), 43
MALLOC, 243
MEMCHR, 244
MEMCMP, 245
MEMCPY, 246
MEMMOVE, 247
memory
    allocation, 45
Memory Model, 89
memory organization , 88
MEMSET, 248
Microsoft
    assembler, 55
    object, 56
MKDIR, 249
MS-DOS, 53
    linker, 56

N

non-volatile RAM, 93
numbers
    in linker options, 101

O

OPEN, 250, 251
Optimisation , 94
options
    linker, 101, 103, 105

P

PACC, 6
PACC.EXE, 3
PACKING.LST, 1
PATH environment variable, 3
peek(), 46
PERROR, 252
pointers, 48
POW, 253
powerup routine, 92
PRINTF, VPRINTF, 254, 255
Processor Type, 91
prototypes
    function, 45
psect, 49, 59, 61
    flags, 63, 64
    linking, 99
PUTC, 257
PUTCH, 258
PUTCHAR, 259
PUTS, 260
PUTW, 261

Q

QSORT, 262, 263

R

RAM
    non-volatile, 93
RAND, 264
READ, 265
REALLOC, 266
reboot
    after installation, 3
reloc=, 50
relocatable
    code, 59
relocation, 99
REMOVE, 267
RENAME, 268
reset address, 88
REWIND, 269
RMDIR, 270

ROM
code, 48
ROM Development, 87
run-time startups, 87
runtime
organization, 43, 44, 46, 48, 50, 52, 54
startup, 43

S

SBRK, 271
SCANF, VSCANF, 272, 273
segment, 49
SEGREAD, 274
serial number, 2, 95
serial port , 88
serial.c, 96
SET_VECTOR, 278, 279
SETJMP, 275
SETVBUF, SETBUF, 276, 277
SIGNAL, 280, 281
signatures, 47
SIN, 282
size
error, 57
Source File List, 94
SPAWNL, SPAWNV, SPAWNVE, 283
SPRINTF, 285
SQRT, 286
SRAND, 287
SSCANF, 288
stack, 45
frame, 43
STAT, 289
STRCAT, 291
STRCHR, 292
STRCMP, 293
STRCPY, 294
STRLEN, 295
STRNCAT, 296
STRNCMP, 297
STRNCPY, 298
STRRCHR, 299
symbols, 57
global, 100
SYSTEM, 300

T

TAN, 301
temporary
labels, 58
TIME, 302
TOLOWER, TOUPPER, TOASCII, 303

U

undefined
symbols, 55
UNGETC, 304
UNLINK, 305

V

VA_START, VA_ARG, VA_END, 306, 307

W

WRITE, 308

1 Introduction

2 Quick Start Guide

3 Using PPD

4 Runtime Organization

5 8086 Assembler Reference Manual

6 Lucifer- A Source Level Debugger

7 Rom Development with Pacific C

8 Linker Reference Manual

9 Librarian Reference Manual

A Error Messages

B Library Functions

Pacific C for DOS, 2nd printing, April 1995
Copyright © 1995 HI-TECH Software