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: 8 _

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:

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

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

- q Other (please specify)

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

Please indicate programming languages you use

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

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

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

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

Thank you!

HI-TECH SOFTWARE

Tape here

HI-TECH SOFTWARE

Please attach correct postage

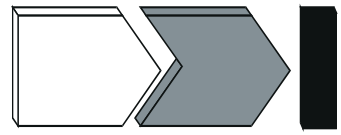
HI-TECH Software
PO Box 103
ALDERLEY QLD 4051
AUSTRALIA

There is also a mailing list you may join which carries discussion of Pacific C. For information or to join this list, send electronic ma
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

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 calledPACKING.LST on disk 1. To install the compiler, you must use theINSTALL program on disk 1. This is located in the root directory of disk 1. Place disk 1 in either floppy drive, then typeA:INSTALL or B:INSTALL as appropriate. TheINSTALL 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 theSET command) in case you have an environment variable calledTEMP 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

TheINSTALL 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 whichINSTALL 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 selectTERMINATE , the installation will be incomplete. A sliding bar will indicate the approximate degree of completion of the installation.

1.2.1.1 Installation Steps

WhenINSTALL prompts for action, you may either pressENTER 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.

1

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 C:\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, p 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 README.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.

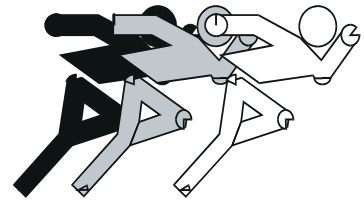
1.2.1.4 Accessing the Compiler

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.

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 Pacific C Programmer's Development system integrated environment. It assumes that you already have an installed Pacific C compiler, if you haven't installed your compiler 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.

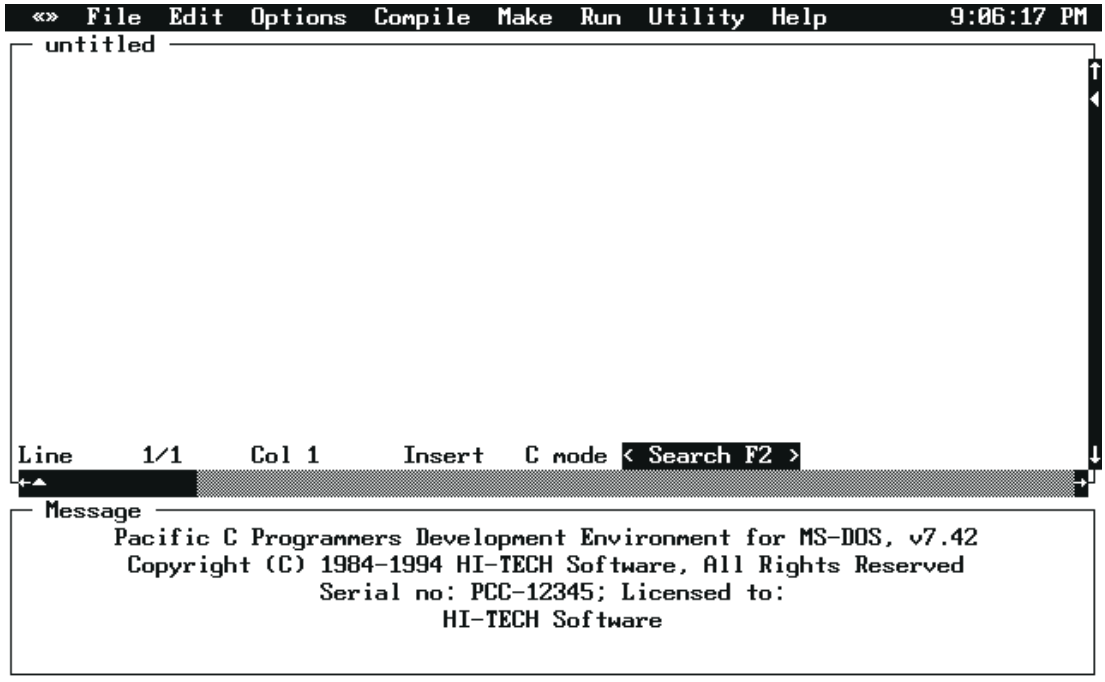


Figure 3-1; PPD Startup screen

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 gainedhot-keysequences 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 a /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 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 the Alt key. 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.

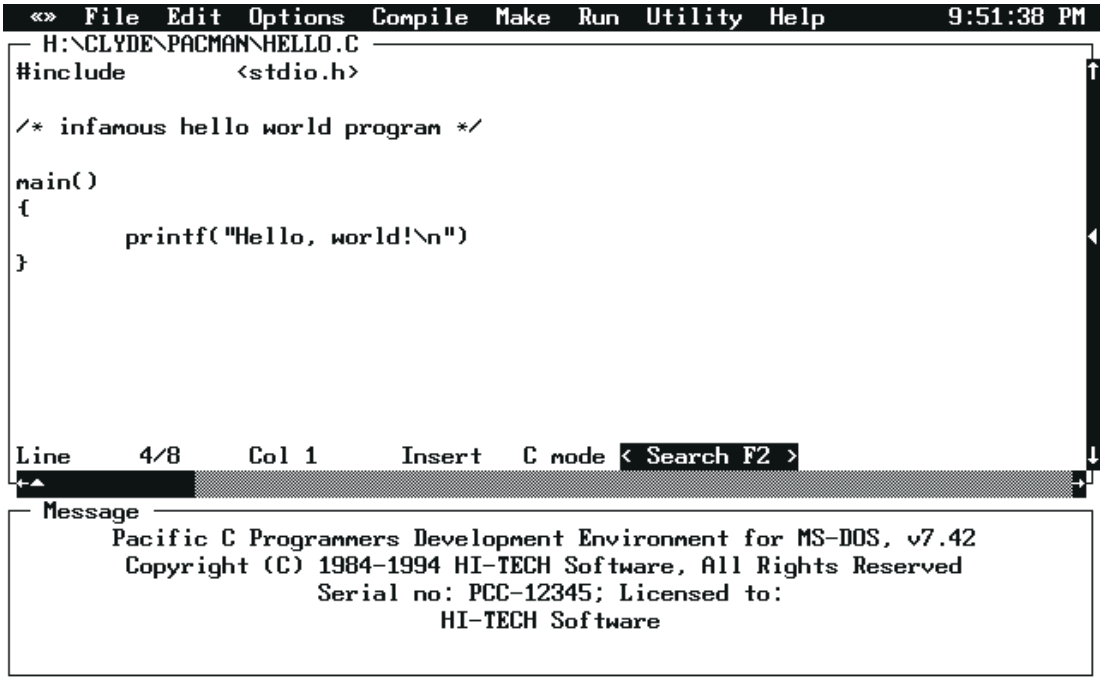


Figure 3-2; Hello proram in PPD

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.

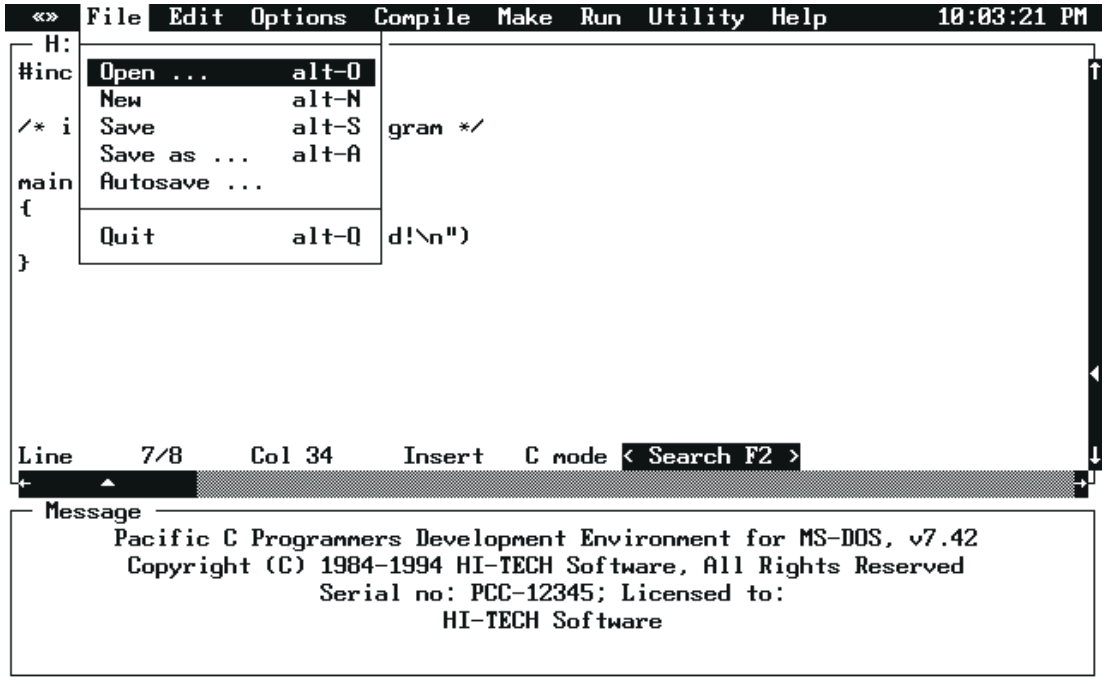


Figure 3-3; PPD File menu

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.

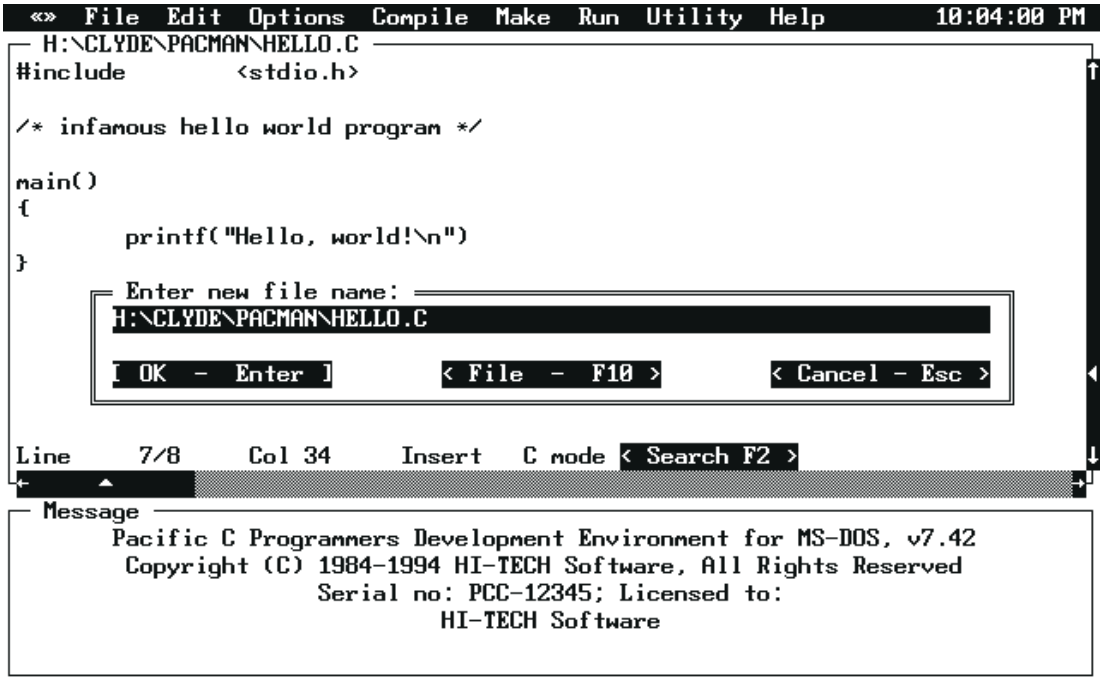


Figure 3-4; PPD File save dialogue

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).

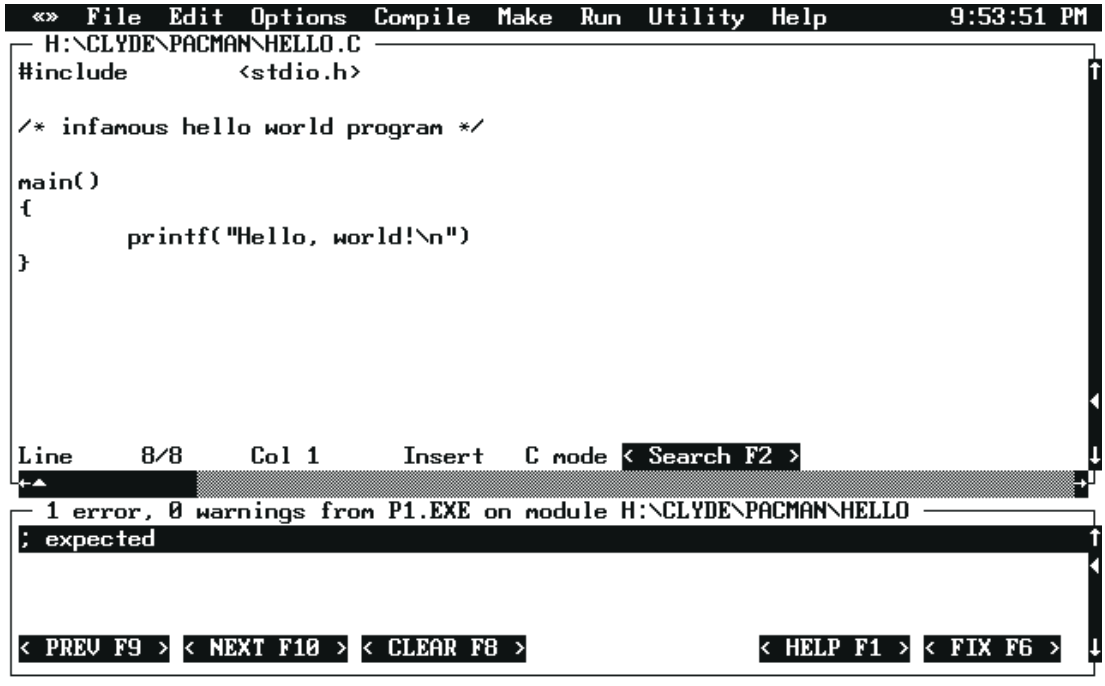


Figure 3-5; PPD Error window

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 Ins key toggles the mode.

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 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 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 all 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.

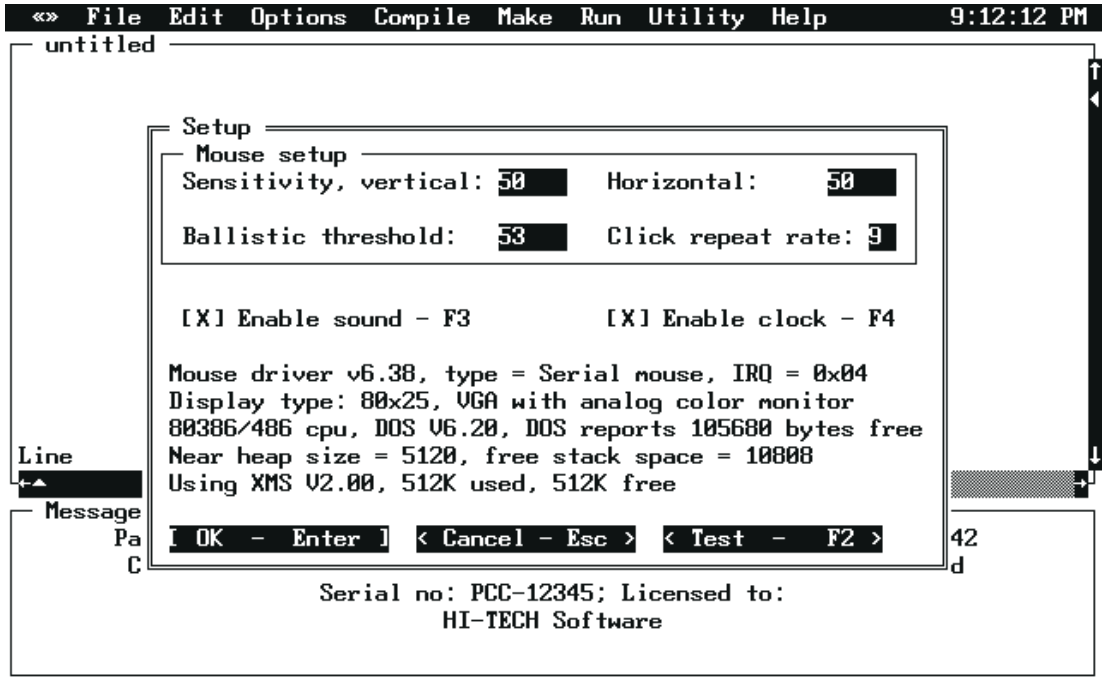


Figure 3-6; Setup dialogue

3.5.2 File Menu

The Edit menu contains editor file handling commands and the 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

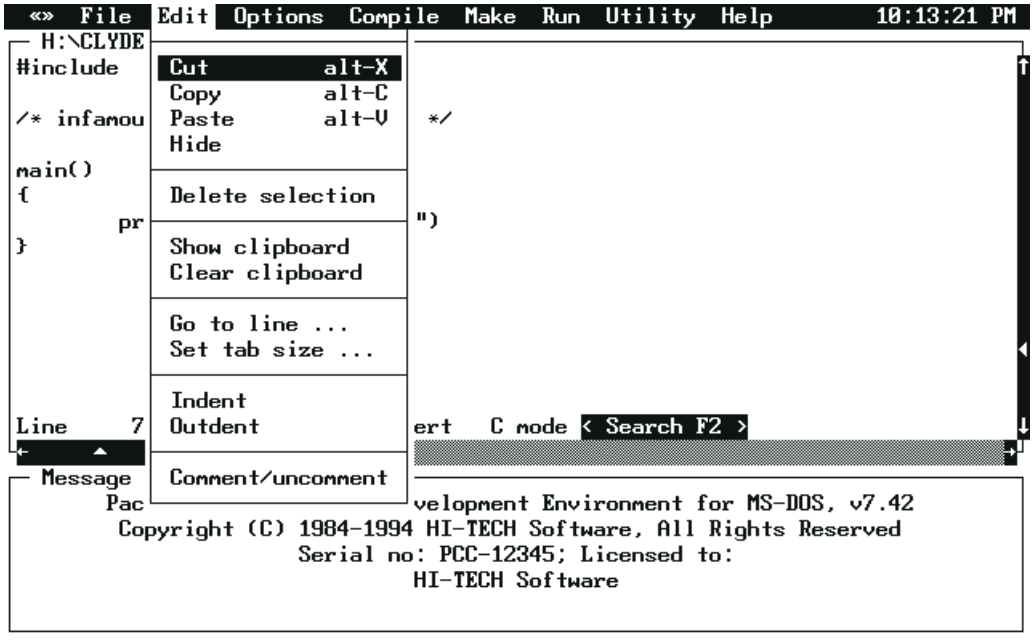


Figure 3-7; PPD edit menu

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 opti
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.

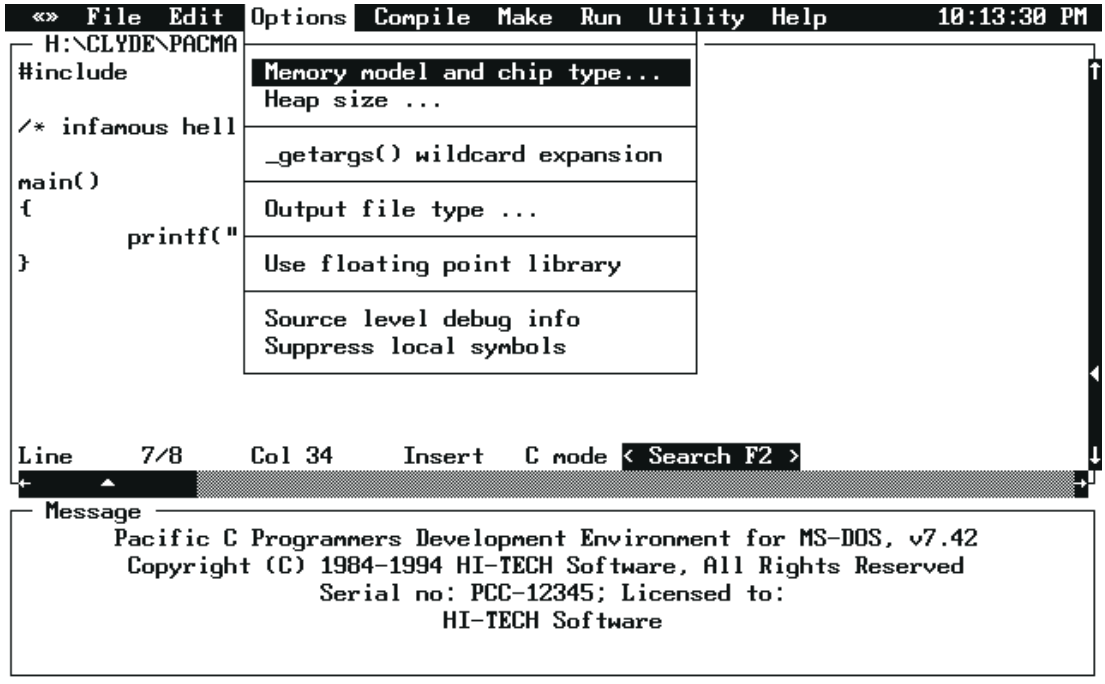


Figure 3-8; PPD options menu

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.

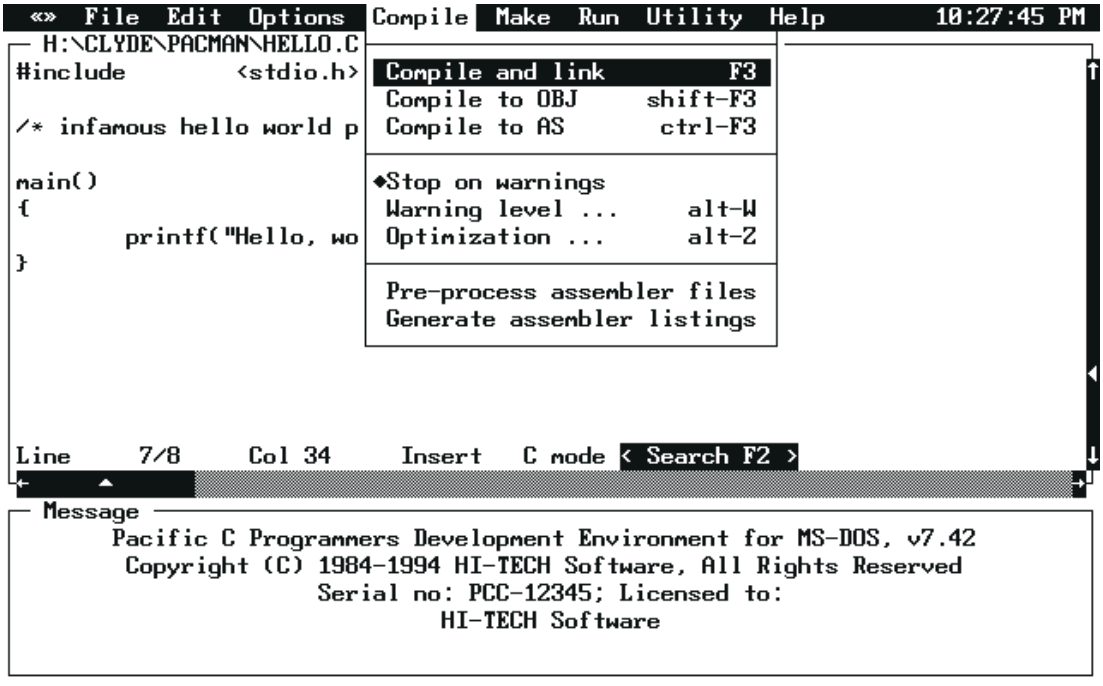


Figure 3-9; PPD Compile Menu

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 determ 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.

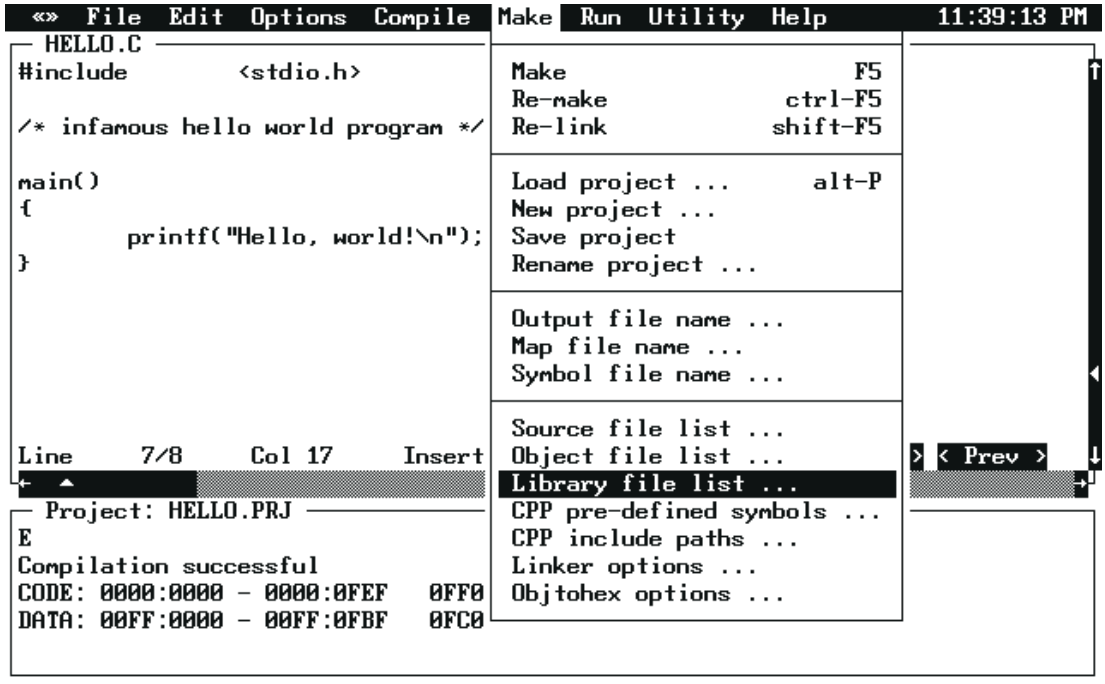


Figure 3-10; PPD make menu

- 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
When a header file enclosed in angle brackets, for example <stdio.h> is included, the compiler will se
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.

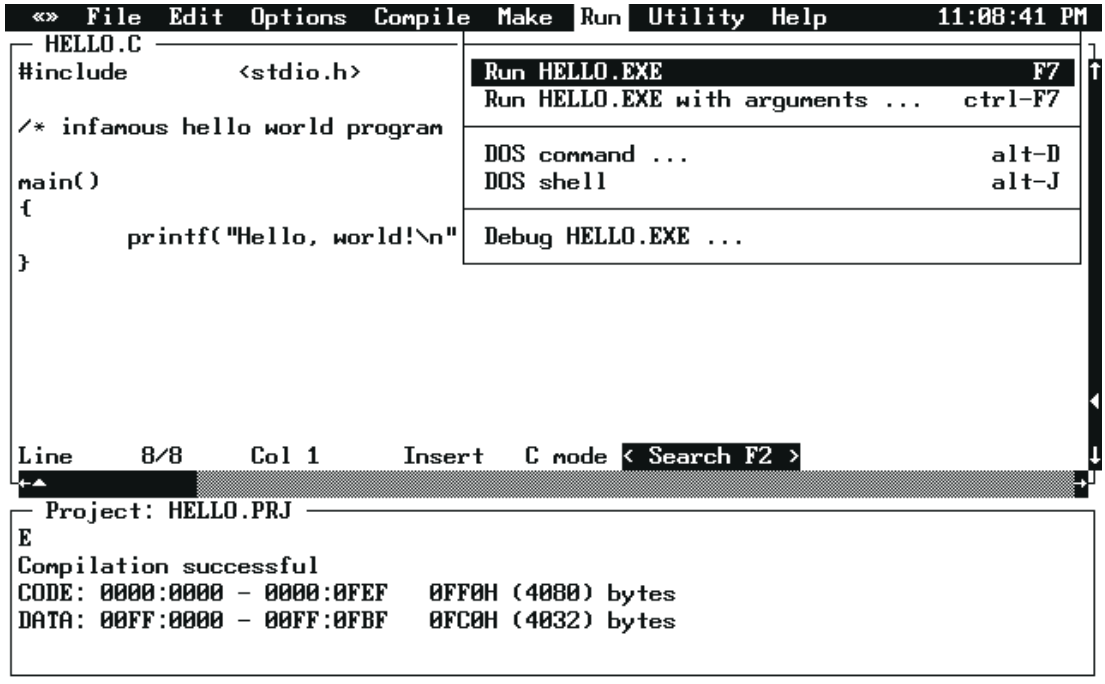


Figure 3-11; PPD run menu

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.

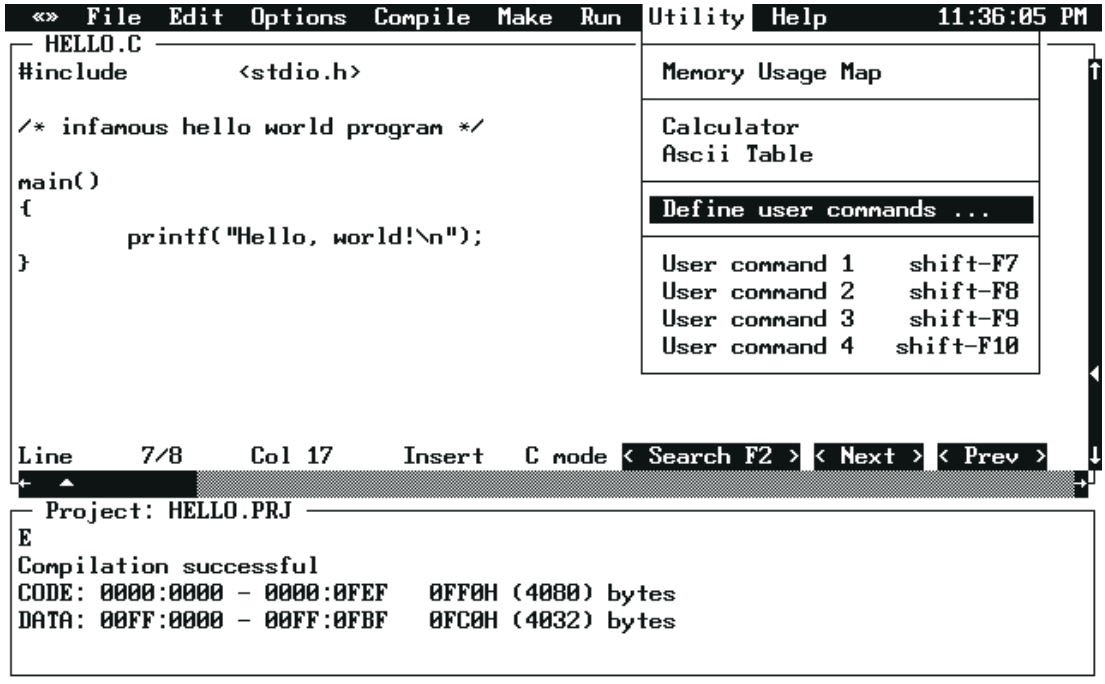


Figure 3-12; PPD utility menu

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.

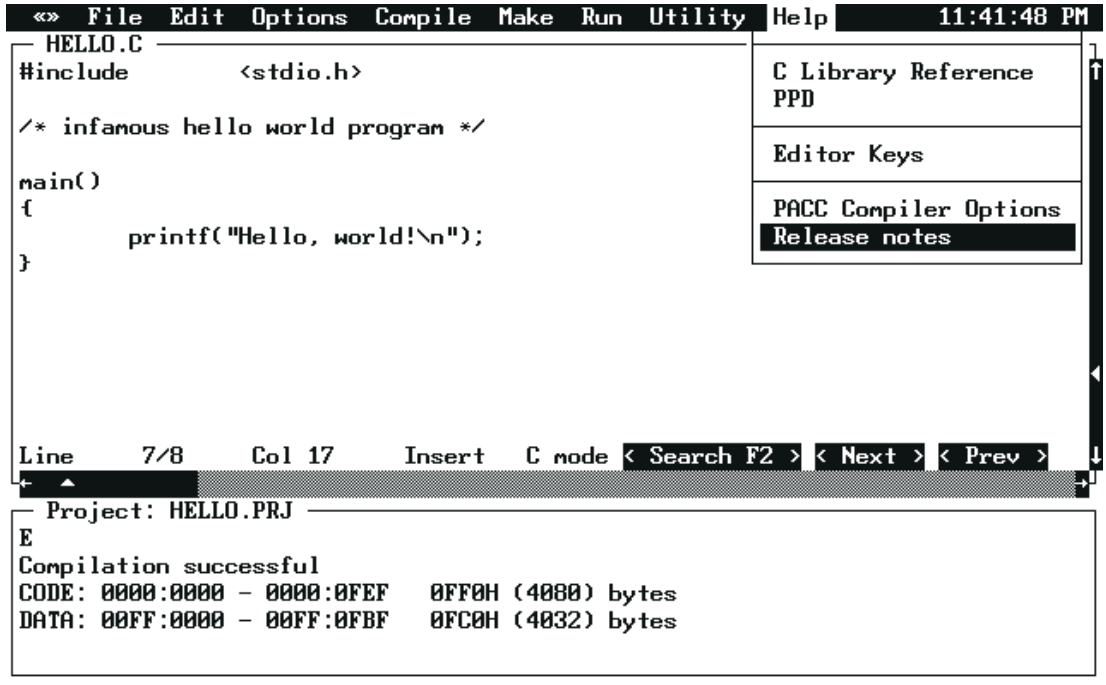


Figure 3-13; PPD help menu

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 fputc , 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 small model means not more than 64K bytes. Note also that in small model, the stack size is included in the data size. In large 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.

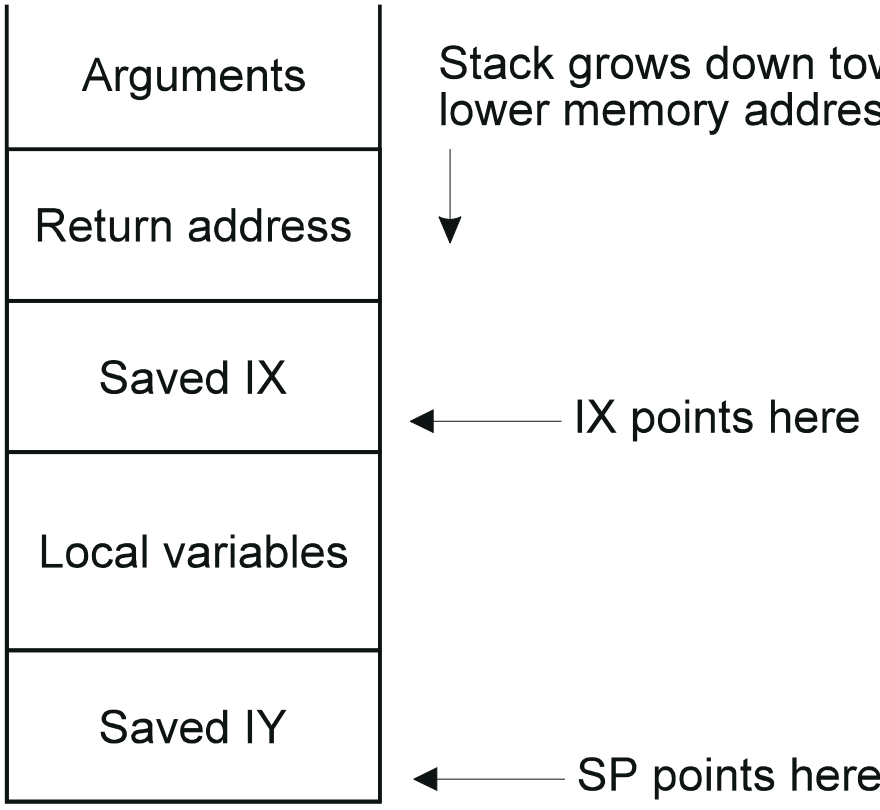


Figure 4-1; Stack frame

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 le or less, be passed DX and AX respectively. Arguments past the first two or bigger than int size wi 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 by 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 __bss. 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 the 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

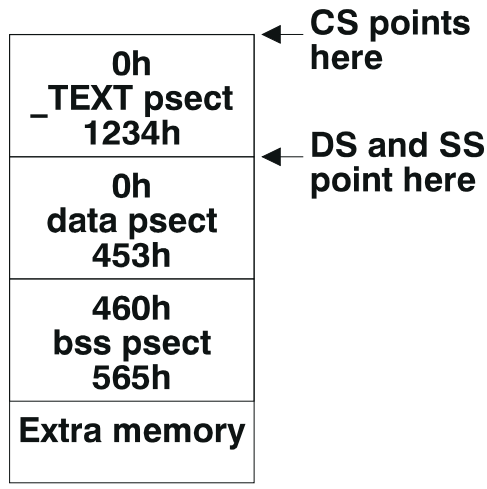


Figure 4-4; Small model memory layout

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

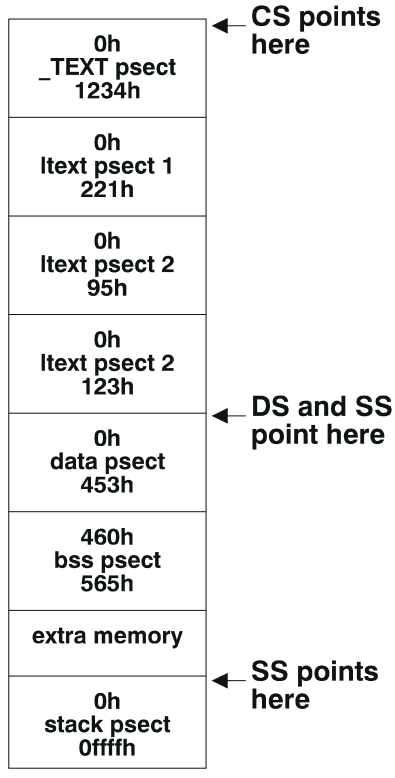


Figure 4-5; Large model memory layout

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 text psects, and allows all the text 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 text 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 -M 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

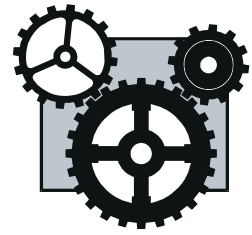

```
io_port  |=  0x40;

*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 where 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 -Ofile

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 -Llist

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

5.2.5 -Wwidth

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 an l, 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 nam 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 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 branch 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 register 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.

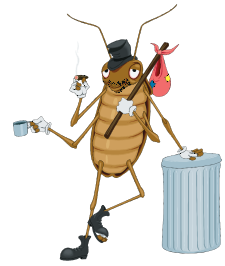
80386 instruction/addressing mode
An instruction or addressing mode peculiar to the 80386 encountered. Use a .386 or .486 directive to allow these.
80486 instruction/addressing mode
An instruction or addressing mode peculiar to the 80486 encountered. Use a .486 directive to allow these.
absolute expression 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 "Development with Pacific C" for more information on remote debugging.

6.2 Usage

To use Lucifer you should compile your program with the -G option. This will produce a symbol file with line number and filename symbols included. If you use the -Hs 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. func+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 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 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, md consists of zero or more modifiers which effect the way the type is displayed, id 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 "test.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 the 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

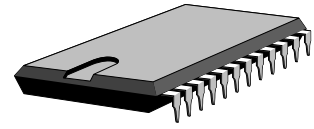
The ! command is used to execute an operating system shell command line without exiting from Lucifer. Any text typed after the ! is passed through to the shell without modification.

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 pop embedded variant of the 8086 family, however the techniques discussed are equally applicable to 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 (ɛ-eks) 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 ne have both knowledge and understanding of how to program the I/O ports to achieve the function 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 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 R 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 e[illegible] match the actual sizes for the program to work; for example a 64K byte ROM would still work with t[illegible] setup, but only the top half would be accessible, and when programming it the code would have [illegible] 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 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 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!

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 berelocatablesince 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

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 segm addresses, e.g. 007F:1234 represents a segment number of 7F and an offset of 1234.

8.7.3 -Aclass= 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 -Cpsect=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 -Dsymfile

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 -Gspec

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 -Hsymfile

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 -Mmapfile

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 -Ooutfile

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 -Pspec

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:

-P *psect* =*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 -Spsect=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 -Usymbol

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 -Vavmap

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 -Wnum

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.