Z80

C CROSS-COMPILER FOR THE **Z80** AND **Z180** MICROCONTROLLERS

HI-TECH SOFTWARE

Z80

C CROSS-COMPILER FOR THE **Z80** AND **Z180** MICROCONTROLLERS

HI-TECH SOFTWARE

HI-TECH Software
A division of Gretetoy Pty. Ltd. ACN 002 724 549
PO Box 103 Alderley QLD 4051 Australia
Phone +61 7 3355 8333 Fax +61 7 3355 8334
E-mail: hitech@htsoft.com
Web: http://www.htsoft.com
FTP: ftp.htsoft.com

Second Printing (c), March 2000

Technical Support

For technical support on the HI-TECH C compiler, you should contact HI-TECH Software by one of the means listed in the table below. To obtain technical support you must have registered your compiler, by sending in the registration card found inside the front cover of this manual. This will entitle you to 3 months free technical support, from the time you make your first technical support request. You will also be entitled to one free update, which will be sent to you automatically if you have returned your registration card.

After the initial free support period, you may take out an annual support agreement to cover this compiler. Contact HI-TECH Software or your supplier for pricing information. The annual support agreement will provide you with priority access to technical support, and all updates, sent to you automatically.

If you do not wish to pay for annual support, then you may send technical support requests, but these will be responded to only as time permits. This may result in considerable delay. Minor updates can be obtained free of charge by downloading patch files from our WWW and ftp servers. These will correct problems with released compiler versions. They will not upgrade from one major version to another. Upgrades may be purchased individually.

World Wide Web	http://www.htsoft.com
Electronic Mail	support@htsoft.com
Fax	+61 7 3355 8333
Telephone	+61 7 3355 8334
Postal	PO Box 103 ALDERLEY QLD 4051 Australia

In other countries, you may also contact the following distributors for support and sales enquiries - **always contact your supplier first for technical support**. Policies of individual resellers may vary with regard to free support.

Country	Dealer	Phone	Fax	E-mail
Australia	HI-TECH Software	+61 7 3355 8333	+61 7 3355 8334	hitech@htsoft.com
Brazil	Anacom Software	+55 11 453 5588	+55 11 441 5177	esouto@anacom.com.br
Canada	Ximetrix Systems	+1 905 681 9600	+1 905 681 3141	facts@ximetrix.ca
Denmark	Digitek Instruments	+45 4342 4742	+45 4342 4743	digitek@pip.dknet.dk
France	Emulations	+33 1 69 412 801	+33 1 60 192 950	ac@emulations.fr
Germany	Reichmann Microcomputer	+49 7141 71 042	+49 7141 75 312	ThomasReichmann@reichmann-mc.de
Ireland	Ashling Microsystems	+353 61 33 4466	+353 61 33 4477	ashling@iol.ie
Italy	Grifo SNC	+39 51 892 052	+39 51 893 661	sales@grifo.it
Japan	Unidux Inc.	+81 422 32 4500	+81 422 31 0331	yamato@unidux.co.jp
Netherlands	Tritec Benelux BV	+31 184 41 41 31	+31 184 42 36 11	development-tools@tritec.nl
Norway	Component-74 Eidsvold A/S (C74)	+47 63 95 60 10	+47 63 95 10 19	c74@riksnett.no
New Zealand	Brent Brown	+64 7 849 0069	+64 7 849 0069	brent.brown@clear.net.nz
Poland	Amart Logic	+48 22 872 46 44	+48 22 612 69 14	cichy@amart.com.pl
South Africa	Avnet Kopp (Pty) Ltd	+27 11 444 2333	+27 11 444 1706	Stuart@avnetkopp.co.za
Spain	SPRINT TRONICA SYSTEM SL	+34 1 319 46 97	+34 1 308 47 70	franlp@iies.es
Sweden	Nohau Elektronik AB	+46 40 592 200	+46 40 592 229	mj@nohau.se
Switzerland	COMSOL AG	+41 31 998 44 11	+41 31 998 44 18	info@comsol.ch
Taiwan	CHINATECH CORPORATION	+886 2 916 0977	+886 2 912 6641	chntech@ms2.hinet.net
UK	Nohau UK Ltd	+44 196 273 3140	+44 196 273 5408	cliffm@nohau.co.uk
	Computer Solutions Ltd	+44 1932 829460	+44 1932 829460	sales@computer-solutions.co.uk
USA	HI-TECH Software LLC	+1 800 735 5715	+1 407 722 2902	hitech@htsoft.com
	CMX Company	+1 508 872 7675	+1 508 620 6828	cmx@cmx.com

Introduction

1.1 Typographic conventions

Throughout this manual, we will adopt the convention that any text you need to type will be printed in bold type. Computer prompts and responses will be printed in `constant spaced type` which will be in **`bold`** where you are required to type it. Particularly useful points and new terms will be emphasised using *italicised type*. With a window based program like HPD, some concepts are difficult to convey in text. These will be introduced using short tutorials and sample screen displays.

1.2 The HI-TECH C Z80 cross compiler

This manual covers the HI-TECH C Z80 cross compiler for the Z80 family of micro-controllers. In this manual you will find information on installing, using and customising the compiler.

The compiler runs under MS-DOS, Unix and Xenix. For use under MS-DOS, HI-TECH C requires an 8088, 8086, 80186, 80286, 80386, 80486 or Pentium processor with at least 512K of free conventional memory, and a hard disk. MS-DOS 3.1 or later is required. We recommend MS-DOS 3.3 or later, or DR-DOS 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.

1.3 Installation

The installation process depends on the operating system which you are using.

1.3.1 MS-DOS

The HI-TECH C Z80 cross compiler is supplied on two or more 3.5" or 5.25" diskettes. The contents of the disks are listed in a file called PACKING.LST on disk 1. To install the compiler, you must use the INSTALL program on disk 1. This is located in the root directory of the disk.

Place disk 1 in either floppy drive, then type **`a:install`** or **`b:install`** as appropriate. The INSTALL program will then present a series of messages and ask for various information as it proceeds. You may need to check your environment variables, (use the SET command), in case you have an environment variable called TEMP set. If this variable is set, its value should be a directory path. The full path must exist and designate a directory in which temporary files can be created. If the TEMP variable specifies a non-existent directory, the INSTALL program may fail.

1.3.2 INSTALL program

The INSTALL program uses several on-screen windows. There is a message window, in which messages and prompts are displayed. There is also a one-line status window in which INSTALL shows what it is currently doing. Other windows will pop up from time to time.

A dialog or alert window will pop up when INSTALL needs you to act, or when any kind of error occurs. These offer the opportunity to continue, retry (if appropriate) or terminate. If you select TERMINATE, the installation will be incomplete. A sliding bar indicates the approximate degree of completion of the installation.

1.3.2.1 Installation steps

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

Initially INSTALL will simply advise it is about to install a HI-TECH Software package. Select CONTINUE or press ENTER. You will then be asked to choose between a full, no questions asked installation, or a custom installation. A custom installation allows you to 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.

1.3.2.2 Custom installation

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

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

1.3.2.3 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.

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

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

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

After this INSTALL will display some messages, then advise you to press ENTER to read the release notes for the compiler. This will load the file READ.ME in the screen window and allow you to read it. The file is also automatically copied onto your hard disk. 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 tell you to do this if it is necessary. The installation is now complete.

1.3.3 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 may have the same name as one of the compiler programs. To overcome this you may need to re-organise your PATH environment variable.

The compiler drivers are ZC.EXE (command line version) and HPDZ.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.

1.4 Unix Installation

Unix versions of the Z80 C Cross compiler are usually supplied on TAR format diskettes or tapes. To install the compiler, you will first need to create a directory. By default the directory the compiler driver will expect is */usr/hitech* but this can be overridden by setting the HTC_Z80 environment variable just as for MS-DOS.

To extract the compiler, first create the direcory into which you will install the compiler:

```
mkdir /usr/hitech
```

Replace the directory name with whatever you have chosen. Then change into the directory:

```
cd /usr/hitech
```

Then extract the files from the diskette or tape. Replace the device name with whatever physical device you will use to read the tape or disk:

```
tar xf /dev/install
```

The compiler will now be installed. You should add to your PATH environment variable the *bin* directory, for example */usr/hitech/bin*. If the directory is anything other than */usr/hitech* then you will need to set the environment variable in your .cshrc or .profile file. For example, in .cshrc (for the c-shell) add the line:

```
setenv HTC_Z80 /home/hitech
```

A Bourne shell user will require the following lines in .profile:

```
HTC_Z80=/home/hitech
export HTC_Z80
```

Again replace the directory name with whatever you have chosen.

1.4.1 Accessing the compiler under Unix

The Unix compiler has only a command line driver - HPDZ is not provided. The command line options are the same as for DOS - just use the ZC command as described in subsequent chapters.

1.5 Getting started

For new users of the HI-TECH C compiler, the following section provides a step-by-step guide to getting your first program running in a target system. You'll need a working Z80 or Z80 derivative system, with RAM and ROM, and some means of either downloading code or programming an EPROM. And, of course, you'll need to have installed the compiler as described in the previous chapter.

You will find a complete guide to using HPDZ and ZC in the chapters "Using HPDZ" and "Features and Runtime Environment".

One thing should be made clear; with embedded programming there really is no such thing as a "quick start". There are several variables, e.g. the hardware, memory, I/O devices and the software, all of which must be exactly right or the program will simply not work. There are no error messages when your embedded program crashes - it is a black box. Be prepared to check everything carefully, and if possible start with known working hardware. Debugging hardware and software at the same time squares the degree of difficulty.

1.6 A Sample Program

```
/*
 *      Test program for Z180.
 *      Flashes LEDs attached to I/O port
 *      at 0x80
 */

/*
 *      Define the LED port
 */
static port unsigned char LED @ 0x80;
main()
{
        register int    i, j;
        i = 0;          /* 0 turns leds on */
        for(;;) {       /* loop endlessly */
                LED = i;
                for(j = 0 ; ++j ;
                        continue;       /* delay */
                i = ~i;
        }
}
```

The small sample program shown is written for the Z80180 processor, with 8 LEDs attached to the output port at 0x80. It loops forever flashing the LEDs. If your hardware is different, as it almost certainly is, you should write a similar program tailored for your particular hardware. Flashing LEDs is however a good place to start, as it provides a visual indication of program function. If you don't have LEDs attached, then you could monitor an output port line with a CRO or logic probe. Even monitoring address lines with a CRO or logic analyser can be used. The idea is to be able to determine that the program is running correctly using a minimum of resources, so as to remove as many variables as possible from the problem

Once the first program is running, it is easier to progress from that point than to try and run a complex program from the beginning. To get this program running, you will need to compile it, either using HPDZ, the integrated development environment, or ZC, the command line compiler driver.

1.6.1 Memory Map

Before compiling your sample program, you will need to know what memory map you are using. If the test hardware you will execute this code in is a standard Z80 or equivalent, i.e. it has a 64K memory space with no bank switching, then you should simply have a block of ROM at zero, and a block of RAM at some other address in the first 64K. You will need to know what address the RAM starts at, and how big it is.

If you are using a Z180 or 64180, it is likely that your memory will be located at other addresses in the 512K or 1M memory space of these processors. The ROM will probably still start at zero, but it is possible the RAM starts at an address bigger than 64K. You will need to know this address, as well as choosing an address within the first 64K to map the RAM to, e.g. you might have RAM at 10000 hex, but want to map it to 8000, extending to FFFF. This would leave 0 to 7FFF for ROM.

1.7 Using HPDZ

To enter this program, simply follow these steps:

❐ Start HPDZ by typing HPDZ, then press **Enter**. If you have installed HPDZ properly, it will be in your search path. You should have on screen a *menu bar,* a large *edit window*, and a smaller m*essage 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 **SAMPLE.C** and press **Return**. The file will be saved to disk.

❐ Press **ALT-P** to open the **Options** menu. Use either the mouse or arrow keys to select the item **Memory model and chip type**. This will open a dialog box enabling you to select either Z80 or Z180 code, and small or banked model. Select the small model, and either Z80 or Z180 depending on what processor you are using. Press **Return** to exit the dialog box.

❐ Press **ALT-P** again and select the **ROM output file** item. This will open a dialog box allowing you to select an output format for your executable file. Choose a file format compatible with your EPROM programmer. Intel HEX or binary are the most commonly used formats. Press **Return** to exit the dialog box.

❐ Press **F3** to compile the program. If you haven't saved the edit file, you will be prompted to do so now. Save it as SAMPLE.C. A dialog box will open asking for memory addresses. Enter in the ROM address field the address at which you want the code to start (this will almost always be zero) and enter in the RAM address field the start of your RAM. If you are using a Z180 you will also have to enter a RAM physical address. This should be the linear address in the Z180's address space at which the RAM is addressed. In this case the first RAM address can be set at any value below 64K, but should be chosen to avoid low memory where ROM will be located.

Press **Enter** to exit the dialog box. HPDZ 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 SAMPLE.HEX (or SAMPLE.BIN for binary) will be left in the current directory.

❐ Exit HPD by pressing **ALT-Q**.

1.8 Using ZC

To use ZC to compile your sample program, you will first need to create a file containing the program. You can use whatever text editor you are familiar with, as long as it can create a plain ASCII file. The DOS EDIT command is satisfactory. Call the file SAMPLE.C. To run ZC, type:

```
ZC SAMPLE.C
```

If you are compiling for a Z180, then you should add a `-180` option, e.g.

```
ZC -180 SAMPLE.C
```

This both generates code for the Z180 instruction set, and prompts you for the physical RAM address as well as the other addresses needed.

If you have correctly entered the sample program, no error messages should result. If you do get error messages, edit the program to correct the errors, and recompile with ZC as before. ZC will then prompt for some memory addresses, as shown in the following sample:

```
C:\>zc -180 sample.c
HI-TECH C COMPILER (Z80/Z180/64180) V7.60
Copyright (C) 1984-1996 HI-TECH Software
Serial no: CZ80-12345; Licensed to:
Alfred

Invalid or no memory addresses specified:
```

```
Use a -Arom,ram,ramsize,ramphys,nvram option, e.g.
        -A8000,6000,2000,10000,4000
specifies ROM at 8000(hex) and 8K bytes of RAM at physical 10000(hex)
mapped into common area 1 at 6000(hex)
ROM (vectors) address: 0
RAM address: 8000
RAM size(default 800 hex):
Physical address of RAM: 10000
Non-volatile RAM address (if used):
Next time use the option -A0,8000,800,10000

Linking:
Objtohex: .........................

Memory Usage Map:
User:   0069H -  0102H   009AH (154) bytes
CODE:   0000H -  0068H   0069H (105) bytes
CODE:   0103H -  015AH   0058H (88) bytes
RAM:    8000H -  8023H   0024H (36) bytes
```

Points to note:

❒ The ROM address has been specified as zero. It is most unlikely that you will specify anything else here, unless you are going to be running the code in RAM on a board with a ROM monitor.

❒ The RAM address has been given as 8000. This is the hex address that RAM is mapped at by hardware address decoding, or that you want it to be mapped at by the MMU on a Z180 system.

❒ The RAM size has been allowed to stay at the default 800 (hex). This is adequate for this sample program, but in general you should specify here the actual size of the RAM that is available to the compiled program.

❒ The -180 option was used, so the driver has prompted for a physical RAM address. This is required for the Z180 to allow the code to set up the MMU at run time. In this example the RAM physical address has been specified as 10000. You should enter here the actual RAM address. If you are using a Z180 where the RAM and ROM are both mapped into the first 64K bytes, then it is usually best to give the actual address of the RAM for the RAM address and the physical RAM address values.

❒ No non-volatile RAM (NVRAM) has been specified. If you are using NVRAM, enter the physical address of the NVRAM here. Variables defined using the *persistent* keyword are stored in NVRAM.

- ❒ ZC has printed a short summary of memory usage. Check that the addresses in this correspond to the memory addresses you specified.

1.8.1 Output File Format Selection

The compiler supports various output file formats. The two most commonly used for input into and EPROM programmer are Intel HEX and binary. The default is Intel HEX, but can be changed via the **ROM Output File ...** menu selection in the **Options** menu of HPDZ, or with one of the following options to ZC:

-MOTOROLA	Produce Motorola S1/S9 HEX file
-BIN	Produce a binary output file
-UBROF	Produce an IAR Ubrof file
-AAHEX	Produce HEX records with symbols for American Automation emulators
-TEKHEX	Produce Tektronix HEX file output

1.9 Running your program

Once you have compiled the program, you will have a file called SAMPLE.HEX in the current directory. How you get this into your hardware will vary depending on just what you have to work with, but generally speaking you will need either an EPROM programmer or an in-circuit-emulator to allow you to get the program into the memory of your target system The exact procedures for doing so are beyond the scope of this manual.

Tutorials

The following are tutorials to aid in the understanding and usage of HI-TECH's C cross compilers. These tutorials should be read in conjunction with the appropriate sections in the manual as they are aimed at giving a general overview of certain aspects of the compiler. Some of the tutorials here are generic to all HI-TECH C compilers and may include information not specific for the compiler you are using.

2.1 Overview of the compilation process

This tutorial gives an overview of the compilation process that takes place with HI-TECH C compilers in terms of how the input source files are processed. The origin of files that are produced by the compiler is discussed as well as their content and function.

2.1.1 Compilation

When a program is compiled, it is done so by many separate applications whose operations are controlled by either the *command-line driver* (CLD) or *HPD driver*[1] (HPD). In either case, HPD or the CLD take the options specified by the programmer (menu options in the case of HPD, or command-line arguments for the CLD) to determine which of the internal applications need to be executed and what options should be sent to each. When the term *compiler* is used, this is intended to donate the entire collection of applications and driver that are involved in the process. In the same way, *compilation* refers to the complete transformation from input to output by the compiler. Each application and its function is discussed further on in this document.

The compiler drivers use several files to store options and information used in the compilation process and these file types are shown in Table 2 - 1 on page 26. The HPD driver stores the compiler options into a project file which has a ".prj" extension. HPD itself stores its own configurational settings in an INI file, e.g. **`HPD51.ini`** in the **`bin`** directory. This file stores information such as colour values and mouse settings. Users who wish to use the CLD can store the command line arguments in a DOS batch file.

Some compilers come with chip info files which describe the memory arrangements of different chip types. If necessary this file can be edited to create new chip types which can then be selected with the appropriate command-line option of from the **select processor...** menu. This file will also have a ".ini" extension and is usually in the **`lib`** directory.

1. The command line driver and HPD driver have processor-specific names, such as picc, c51, or HPDXA, HPDPIC etc.

The compilation process is discussed in the following sections both in terms of what takes place at each

Table 2 - 1 Configuration files

extension	name	contents
.prj	project file	compiler options stored by HPD driver
.ini	HPD initialisation file	HPD environment settings
.bat	batch file	command line driver options stored as DOS batch file
.ini	chip info file	information regarding chip families

stage and the files that are involved. Reference should be made to Figure 2 - 1 on page 27 which shows the block diagram of the internal stages of the HI-TECH compiler, and the tables of file types throughout this tutorial which list the filename extension[2] used by different file formats and the information which the file contains. Note that some older HI-TECH compilers do not include all the applications discussed below.

The internal applications generate output files and pass these to the next application as indicated in the figure. The arrows from one application (drawn as ellipses) to another is done via temporary files that have non-descriptive names such as **`$$003361.001`**. These files are temporarily stored in a directory pointed to by the DOS environment variable **`TEMP`**. Such a variable is created by a **`set`** DOS command. These files are automatically deleted by the driver after compilation has been completed.

2.1.2 The compiler input

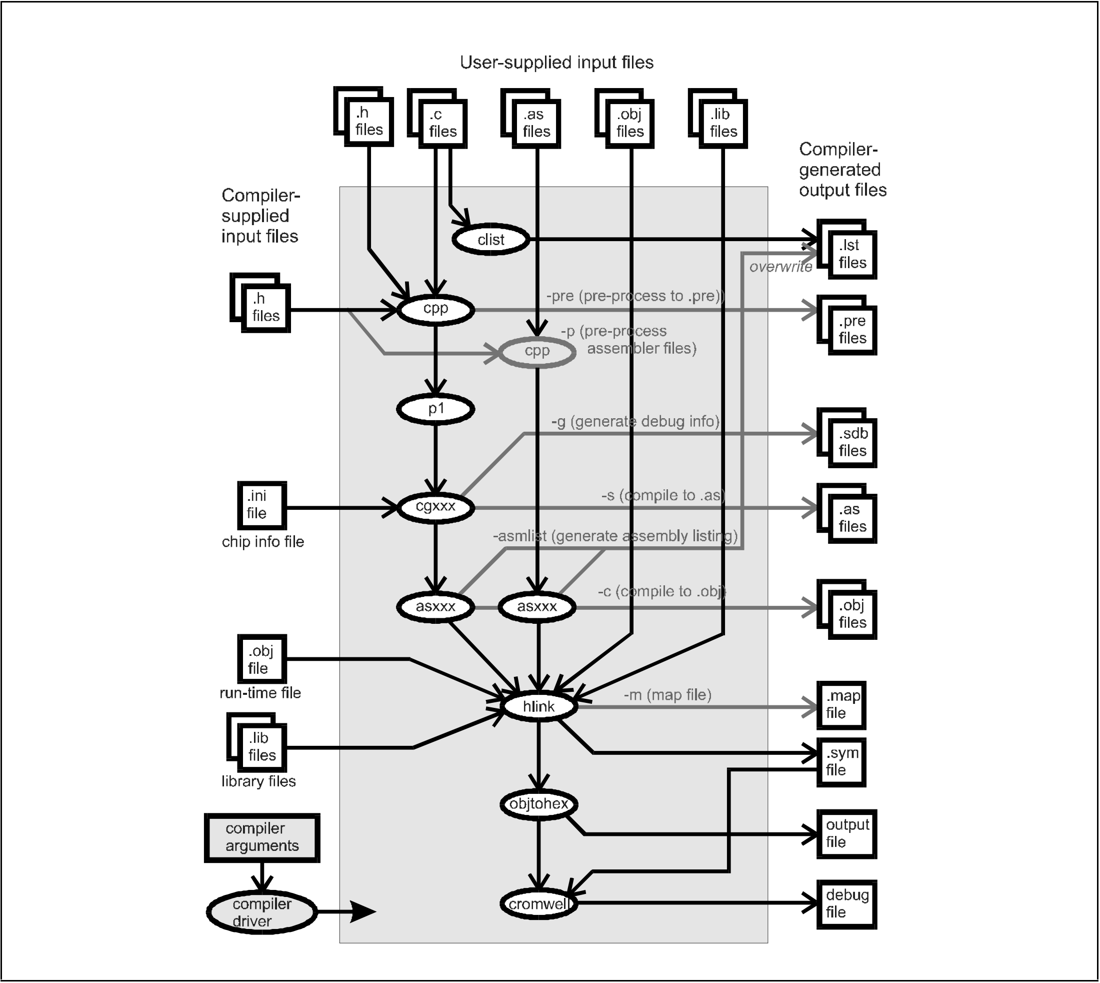
The user supplies several things to the compiler to make a program: the input files and the compiler options, whether using the CLD or HPD. The compiler accepts many different input file types. These are discussed below.

It is possible, and indeed in a large number of projects, that the only files supplied by the user are *C source files* and possibly accompanying header files. It is assumed that anyone using our compiler is familiar with the syntax of the C language. If not, there is a seemingly endless selection of texts which cover this topic. C source files used by the HI-TECH compiler must use the extension ".c" as this extension is used by the driver to determine the file's type. C source files can be listed in any order on the command line if using the CLD, or entered into the **source file list...** dialogue box if using HPD.

A *header file* is usually a file which contains information related to the program, but which will not directly produce executable code when compiled. Typically they include declarations (as opposed to definitions) for functions and data types. These files are included into C source code by a pre-processor

2. The extensions listed in these tables are in lower case. DOS compilers do not distinguish between upper- and lower-case file names and extensions, but in the interest of writing portable programs you should use lower-case extensions in file names and in references to these files in your code as UNIX compilers do handle case correctly.

Figure 2 - 1 Compilation overview



directive and are often called *include files*. Since header files are referenced by a command that includes the file’s name and extension (and possibly a path), there are no restrictions as to what this name can be although convention dictates a ".h" extension.

Although executable C code may be included into a source file, a file using the extension ".h" is assumed to have non-executable content. Any C source files that are to be included into other source files should still retain a ".c" extension. In any case, the practise of including one source file into another is best avoided as it makes structuring the code difficult, and it defeats many of the advantages of having a compiler capable of handling multiple-source files in the first place. Header files can also be included into assembler files. Again, it is recommended that the files should only contain assembler declarations.

Table 2 - 2 Input file types

extension	name	content
.c	C source file	C source conforming to the ANSI standard possibly with extensions allowed by HI-TECH C
.h	header file	C/assembler declarations
.as	assembler file	assembler source conforming to the HI-TECH assembler format
.obj	(relocatable) object file	pre-compiled C or assembler source as HI-TECH relocatable object file
.lib	library file	pre-compiled C or assembler source in HI-TECH library format

HI-TECH compilers comes with many header files which are stored in a separate directory of the distribution. Typically user-written header files are placed in the directory that contains the sources for the program. Alternatively they can be placed into a directory which can be searched by using a **`-I`** (**CPP include paths...**) option.

An *assembler file* contains assembler *mnemonics* which are specific to the processor for which the program is being compiled. Assembler files may be derived from C source files that have been previously compiled to assembler, or may be hand-written and highly-prized works of art that the programmer has developed. In either case, these files must conform to the format expected of the HI-TECH assembler that is part of the compiler. This processor-dependence makes assembly files quite un-portable and they should be avoided if C source can be made to perform the task at hand. Assembler files must have a ".as" extension as this is used by the compiler driver to determine the file's type. Assembler files can be listed in any order on the command line if using the CLD, or entered into the **source file list...** dialogue box if using HPD, along with the C source files.

The compiler drivers can also be passed pre-compiled HI-TECH object files as input. These files are discussed below in Section 2.1.2.1 on page 29. These files must have a ".obj" extension. Object files can be listed in any order on the command line if using the CLD, or entered into the **object file list...** dialogue box if using HPD. You should not enter the names of object files here that have been compiled from source files already in the project, only include object files that have been pre-compiled and have no corresponding source in the project, such as the run-time file should be listed.

Commonly used program routines can be compiled into a file called a *library file*. These files are more convenient to handle and can be accessed quickly by the compiler. The compiler can accept library files directly like other source files. A ".lib" extension indicates the type of the file and so library files must be named in this way. Library files can be listed in any order on the command line if using the CLD, or entered into the **library file list...** dialogue box if using HPD.

The HI-TECH library functions come pre-compiled in a library format and are stored in a special directory in your distribution.

2.1.2.1 Steps before linking

Of all the different types of files that can be accepted by the compiler, it is the C source files that require the most processing. The steps involved in compiling the C source files are examined first.

For each C source file, a *C listing file* is produced by an application called **`clist`**. The listing files contain the C source lines proceeded by a line number before any processing has occurred. The C listing for a small test program called **`main.c`** is shown in Table 2 - 3 on page 29.

Table 2 - 3 clist output

C source	C listing
`#define VAL 2`	`1: #define VAL 2`
	`2:`
`int a, b = 1;`	`3: int a, b = 1;`
	`4:`
`void`	`5: void`
`main(void)`	`6: main(void)`
`{`	`7: {`
`    /* set starting value */`	`8:     /* set starting value */`
`    a = b + VAL;`	`9:     a = b + 2;`
`}`	`10: }`

The input C source files are also passed to the *preprocessor*, **`cpp`**. This application has the job of preparing the C source for subsequent interpretation. The tasks performed by **`cpp`** include removing comments and multiple spaces (such as tabs used in indentation) from the source, and executing any pre-processor directives in the source. Directives may, for example, replace macros with their replacement text or remove source if certain conditions are not true. The pre-processor also copies header files, whether user- or compiler-supplied, into the source. Table 2 - 4 on page 30 shows pre-processor output for the test program.

The output of the pre-processor is C source, but it may contain code which has been included by the pre-processor from other files and only contain code if the pre-processor evaluates specific conditions to be true. The pre-processor output is often referred to as a *module* or *translational unit.* The term "module" is sometimes used to describe the actual source file from which the "true" module is created. This is not strictly correct, but the meaning is clear enough.

The code generation that follows operate on the **`cpp`** output module, not the C source and so special steps must be taken to be able to reconcile errors and their position in the original C source files. The **`# 1 main.c`** line in the pre-processor output is included by the pre-processor to indicate the file name and line number in the C source file that corresponds to this position. Notice in this example that the comment and macro definition have been removed, but blank lines take their place so that line numbering information is kept intact.

Like all compiler applications, the pre-processor is controlled by the compiler driver (either the CLD or

Table 2 - 4 Pre-processor output

C source	Pre-processed output

`#define VAL 2`
 	
`int a, b = 1;`
 	
`void`
`main(void)`
`{`
`    /* set starting value */`
`    a = b + VAL;`
`}`
	
	`# 1 "main.c"`
	
	`int a, b = 1;`
	
	`void`
	`main(void)`
	`{`
	
	`a = b + 2;`
	`}`

HPD). The type of information that the driver supplies the pre-processor includes directories to search for header files that are included into the source file, and the size of basic C objects (such as **`int`**, **`double`**, **`char *`**, etc.) using the **`-S, -SP`** options so that the pre-processor can evaluate pre-processor directives which contain a **`sizeof(`*`type`*`)`** expression. The output of the pre-processor is not normally seen unless the user uses the **`-pre`** option in which case the compiler output can then be re-directed to file.

The output of **`cpp`** is passed to **`p1`**, the *parser*. The parser starts the first of the hard work involved with

Table 2 - 5 Intermediate and Support files

extension	name	contents
.pre	pre-processed file	C source or assembler after the pre-processing stage
.lst	C listing file	C source with line numbers
.lst	assembler listing	C source with corresponding assembler instructions
.map	map file	symbol and psect relocation information generated by the linker
.err	error file	compiler warnings and errors resulting from compilation
.rlf	relocation listing file	information necessary to update list file with absolute addresses
.sdb	symbolic debug file	object names and types for module
.sym	symbol file	absolute address of program symbols

turning the description of a program written in the C language into the actual executable itself consisting of assembler instructions. The parser scans the C source code to ensure that it is valid and then replaces C expressions with a modified form of these. (The description of code generation that follows need not be followed to understand how to use the HI-TECH compiler, but has been included for curious readers.)

For example the expression a **`= b + 2`** is re-arranged to a prefix notation like **`= a + b 2`**. This notation can easily be interpreted as a tree with **`=`** at the apex, **`a`** and **`+`** being branches below this, and **`b`** and **`2`** being sub-branches of the addition. The output of the parser is shown in Table 2 - 6 on page 31 for our small C program. The assignment statement in the C source has been highlighted as well as the output the parser generates for this statement. Notice that already, the global symbols in the parser output have had an underscore character pre-pended to their name. From now on, reference will be made to them using these symbols. The other symbols in this highlighted line relate to the constant. The ANSI standard states that the constant **`2`** in the source should be interpreted as a **`signed int`**. The parser ensures this is the case by casting the constant value. The **`->`** symbol represents the cast and the **`‘i`** represents the type. Line numbering, variable declarations and the start and end of a function definition can be seen in this output.

It is the parser that is responsible for finding a large portion of the errors in the source code. These errors

Table 2 - 6 Parser output

C source	Parsed output
`#define VAL 2`	`Version 3.2 HI-TECH Softwa...`
	`"3 main.c`
`int a, b = 1;`	`[v _a ‘i 1 e ]`
	`[v _b ‘i 1 e ]`
`void`	`[i _b`
`main(void)`	`-> 1 ‘i`
`{`	`]`
`    /* set starting value */`	`"7`
`    a = b + VAL;`	`[v _main ‘(v 1 e ]`
`}`	`{`
	`    [e :U _main ]`
	`    [f ]`
	`    "9`
	`[; ;main.c: 9: b = a + 2;`
	`    [e = _a + _b -> 2 ‘i ]`
	`    "10`
	`[; ;main.c: 10: }`
	`    [e :UE 1 ]`
	`}`

will relate to the syntax of the source code. The parser also reports warnings if the code is unusual.

The parser passes its output directly to the next stage in the compilation process. There are no driver options to force the parser to generate parsed-source output files as these files contain no useful information for the programmer.

Now the tricky part of the compilation: code generation. The *code generator* converts the parser output into assembler mnemonics. This is the first step of the compilation process which is processor-specific. Whereas all HI-TECH pre-processors and parsers have the same name and are in fact the same application, the code generators will have a specific, processor-based name, for example **`cgpic`**, or **`cg51`**.

The code generator uses a set of rules, or *productions*, to produce the assembler output. To understand

Table 2 - 7 Code generator output

C source	assembler (XA) code

```
#define VAL 2

int a, b = 1;

void
main(void)
{
    /* set starting value */
    a = b + VAL;
}
```
	```
	    psect    text
	_main:
	;main.c: 9: a = b + 2;
	    global _b
	    mov     r0,#_b
	    movc.w  r1,[ro+]
	    adds.w  r1,#02h
	    mov.w   _a,r1
	```

how a production works, consider the following analogy of a production used to generate the code for the addition expression in our test program. "If you can get one operand into a register" and "one operand is a int constant" then here is the code that will perform a 2-byte addition of them. Here, each quoted string would represent a sub-production which would have to be matched. The first string would try to get the contents of **`_a`** into a register by matching further sub-productions. If it cannot, this production cannot be used and another will be tried. If all the sub-productions can be met, then the code that they produce can be put together in the order specified by the production tree. Not all productions actually produce code, but are necessary for the matching process.

If no matching production/subproductions can be found, the code generator will produce a "Can't generate code for this expression" error. This means that the original C source code was legal and that the code generator did try to produce assembler code for it, but that in this context, there are no productions which can match the expression.

Typically there may be around 800 productions to implement a full code generator. There were about a dozen matching productions to generate code for the statement highlighted in Table 2 - 7 on page 32 for the XA code generator. It checked about 70 productions which were possible matches before finding a solution. The exact code generation process is too complex to describe in this document and is not required to be able to use the compiler efficiently.

The user can stop the compilation process after code generation by issuing a **`-S`** (**compile to .as**) option to the driver. In this case, the code generator will leave behind assembler files with a ".as" extension.

Table 2 - 7 on page 32 shows output generated by the XA code generator. Only the assembler code for the opening brace of **`_main`** and the highlighted source line is shown. This output will be different for other compilers and compiler options.

The code generator may also produce debugging information in the form of an ".sdb" file. This operation is enabled by using the **`-g`** (**source level debug info**) option. One debug file is produced for each module that is being compiled. These ASCII files contain information regarding the symbols defined in each module and can be used by debugging programs. Table 2 - 5 on page 30 shows the debug files that can be produced by the compiler at different stages of the compilation. Several of the output formats also contain debugging information in addition to the code and data.

The code generator optionally performs one other task: optimization. HI-TECH compilers come with several different optimizer stages. The code generator is responsible for *global optimization* which can be enabled using a **`-Zg`** (**global optimization**) option. This optimization is performed on the parsed source. Amongst other things, this optimization stage allocates variables to registers whenever possible and looks for constants that are used consecutively in source code to avoid reloading these values unnecessarily.

Assembly files are the first files in the compilation process that make reference to *psects*, or program sections. The code generator will generate the psect directives in which code and data will be positioned.

The output of the code generator is then passed to the *assembler* which converts the ASCII representation of the processor instructions - the ASCII mnemonics - to binary *machine code*. The assembler is specific for each compiler as has a processor-dependent name such as **`aspic`** or **`asxa`**. Assembler code also contains assembler directives which will be executed by the assembler. Some of these directives are to define ROM-based constants, others define psects and others declare global symbols.

The assembler is optionally preceded by an optimization of the generated assembler. This is the peephole optimization. With some HI-TECH compilers the peephole optimizer is contained in the assembler itself, e.g. the PIC assembler, however others have a separate optimization application which is run before the assembler is executed, e.g. **`opt51`**. Peephole optimization is carried out separately over the assembler code derived from each single function.

In addition to the peephole optimizer, the assembler itself may include a separate assembler optimizer step which attempts to replace long branches with short branches where possible. The -O option enables both assembler optimizers, even if they are performed by separate applications, however HPD includes menu items for both optimizer stages (**Peephole optimization** and **Assembler optimization**). If the peephole optimizer is part of the assembler, the assembler optimization item in HPD has no effect.

The output of the assembler is an object file. An *object file* is a formatted binary file which contains machine code, data and other information relating to the module from which it has been generated. Object files come in two basic types: *relocatable* and *absolute* object files. Although both contain

Table 2 - 8 Assembler output

C source	Relocatable object file
#define VAL 2	11 TEXT 22
	text 0 13
int a, b;	**99 08 00 00 88 10 A9 12 8E 00 00** D6 80
	12 RELOC 63
void	2 RPSECT data 2
main(void)	9 COMPLEX 0
{	Key: direct
/* set start...	0x7>=(high bss)
a = b + VAL;	9 COMPLEX 1
}	((high bss)&0x7)+0x8
	10 COMPLEX 1
	low bss

machine code in binary form, relocatable object files have not had their addresses resolved to be absolute values. The binary machine code is stored as a block for each psect. Any addresses in this area are temporarily stored as 00h. Separate relocation information in the object file indicates where these unresolved addresses lie in the psect and what they represent. Object files also contain information regarding any psects that are defined within so that the linker may position these correctly.

Object files produced by the assembler follow a format which is standard for all HI-TECH compilers, but obviously their contents are machine specific. Table 2 - 8 on page 34 shows several sections of the HI-TECH format relocatable object file that has been converted to ASCII for presentation using the DUMP executable which comes with the compiler. The highlighted source line is represented by the highlighted machine code in the object file. This code is positioned in a psect called **text**. The underlined bytes in the object file are addresses that as yet are unknown and have been replaced with zeros. The lines after the **text** psect in the object file show the information used to resolve the addresses needed by the linker. The two bytes starting at offset 2 and the two single bytes at offset 9 and 10 are represented here and as can be seen, their address will be contained at an address derived from the position of the **data** and **bss** psects, respectively.

If a **-asmlist** (**generate assemble listing**) option was specified, the assembler will generate an assembler listing file which contains both the original C source lines and the assembler code that was generated for each line. The assembler listing output is shown in Table 2 - 9 on page 35. Unresolved addresses are listed as being zero with unresolved-address markers "**'**" and "*****" used to indicate that the values are not absolute. Note that code is placed starting from address zero in the new **text** psect. The entire psect will be relocated by the linker.

Some HI-TECH assemblers also generate a *relocatable listing file* (extension: ".rlf").[3] This contains address information which can be read by the linker and used to update the assembler listing file, if such

3. The generation of this file is not shown in Figure 2 - 1 on page 27 in the interests of clarity.

Table 2 - 9 Assembler listing

C source	Assembler listing
`#define VAL 2`	`10   0000'                 psect text`
	`11   0000'             _main:`
`int a, b;`	`12                         ;main.c: 9: a = b + 2;`
	`13   0000' 99 08 0000'         mov.w   r0,#_b`
`void`	`14   0004' 88 10               movc.w  r1,[r0+]`
`main(void)`	`15   0006' A9 12               adds.w  r1,#2`
`{`	`16   0008' 8E 00* 00*          mov.w   _a,r1`
`    /* set start...`	`17                         ;main.c: 10: }`
`    **a = b + VAL;**`	`18   000B' D6 80                ret`
`}`	

a file was created. After linking, the assembler listing file will have addresses and unresolved address markers removed and replaced with absolute addresses.

The above series of steps: pre-processing, parsing, code generation and assembly, are carried out for each C source file passed to the driver in turn. Errors in the code are reported as they are detected. If a file cannot be compiler due to an error, the driver halts compilation of that module after the application that generated the error completes and continues with the next file which were passed to it, starting again with the **`clist`** application.

For any assembler files passed to the driver, these do not require as much processing as C source files, but they must be assembled. The compiler driver will pass any ".as" files straight to the assembler. If the user specifies the **`-p`** (**pre-process assembler files**) the assembler files are first run through the C pre-processor allowing the using of all pre-processor directives within assembly code. The output of the pre-processor is then passed to the assembler.

Object and library files passed to the compiler are already compiled and are not processed at all by the first stages of the compiler. They are not used until the link stage which is explained below.

If you are using HPD, *dependency information* can be saved regarding each source and header file by clicking the **save dependency information** switch. When enabled, the HPD driver determines only which files in the project need be re-compiled from the modification dates of the input source files. If the source file has not been changed, the existing object file is used.

2.1.2.2 The link stage

The format of object files are again processor-independent so the linker and other applications discussed below are common across the whole range of HI-TECH compilers. The linker's name is **`hlink`**.[4]

The tasks of the linker are many. The linker is responsible for combining all the object and library files into a single file. The files operated on by the linker include all the object files compiled from the input

4. Early HI-TECH linkers were called **`link`**.

C source files and assembler files, plus any object files or library files passed to the compiler driver, plus any run-time object files and library files that the driver supplies. The linker also performs *grouping* and *relocation* of the psects contained in all of the files passed to it, using a relatively complex set of linker options. The linker also resolves symbol names to be absolute addresses after relocation has made it possible to determine where objects are to be stored in ROM or RAM. The linker then adjusts references to these symbols - a process known as address *fixup*. If the symbol address turns out to be too large to fit into the space in the instruction generated by the code generator, a "fixup overflow" error occurs.

The linker can also generate a map file which has detailed information regarding the position of the psects and the addresses assigned to symbols. The linker may also produce a symbol file. These files have a ".sym" extension and are generated when the **`-g`** (**source level debug info**) option is used. This symbol file is ASCII-based and contains information for the entire program. Addresses are absolute as this file is generated after the link stage.

The output of linkers for compilers which compile for an operating-system based computer are true executable object files that can be loaded and run. The compilation process, however, is not quite finished for a cross compiler. Although the object file produced by hlink contains all the information necessary to run the program, the program has to be somehow transferred from the host computer to the embedded hardware.

There are a number of standard formats that have been created for such a task. Emulators and chip programmers often can accept a number of these formats. The Motorola HEX (S record) or Intel HEX formats are common formats. These are ASCII formats allowing easy viewing by any text editor. They include *checksum* information which can be used by the program which downloads the file to ensure that it was transmitted without error. These formats include address information which allows those areas which do not contain data to be omitted from the file. This can make these files significantly smaller than, for example, a binary file.

The **`objtohex`** application is responsible for producing the output file requested by the user. It takes the absolute object file produced by the linker and produces an output under the direction of the compiler driver. The **`objtohex`** application can produce a variety of different formats to satisfy most development systems. The output types available with most HI-TECH compilers are shown in Table 2 - 10 on page 37.

In some circumstances, more than one output file is required. In this case an application called **`cromwell`**, the reformatter, is executed to produce further output files. For example it is commonly used with the PIC compiler to read in the HEX file and the SYM file and produce a COD file.

Table 2 - 10 Output formats

extension	name	content
.hex	Motorola hex	code in ASCII, Motorola S19 record format
.hex	Intel hex	code in ASCII, Intel format
.hex	Tektronix hex	code in ASCII Tek format
.hex	American Automation hex	code and symbol information in binary, American Automation format
.bin	binary file	code in binary format
.cod	Bytecraft COD file	code and symbol information in binary Bytecraft format
.cof	COFF file	code and symbol information in binary common object file format
.ubr	UBROF file	code and symbol information in universal binary relocatable object format
.omf	OMF-51 file	code and symbol information in Intel Object Module Format for 8051
.omf	enhanced OMF-51 file	code and symbol information in Keil Object Module Format for 8051

2.2 Psects and the linker

This tutorial explains how the compiler breaks up the code and data objects in a C program into different parts and then how the linker is instructed to position these into the ROM and RAM on the target.

2.2.1 Psects

As the code generator progresses it generates an assembler file for each C source file that is compiled. The contents of these assembly files include different sections: some containing assembler instructions that represent the C source; others contain assembler directives that reserve space for variables in RAM; others containing ROM-based constants that have been defined in the C source; and others which hold data for special objects such as non-volatile variables, interrupt vectors and configuration words used by the processor. Since there can be more than one input source file there will be similar sections of assembler spread over multiple assembler files which need to be grouped together after all the code generation is complete.

These different sections of assembler need to be grouped in special ways: It makes sense to have all the initialised data values together in contiguous blocks so they can be copied to RAM in one block move rather than having them scattered in-between sections of code; the same applies to uninitialised global objects which have to be allocated a space which is then cleared before the program starts; some code or objects have to be positioned in certain areas of memory to conform to requirements in the processor's addressing capability; and at times the user needs to be able to position code or data at specific absolute addresses to meet special software requirements. The code generator must therefore include information which indicates how the different assembler sections should be handled and positioned by the linker later in the compilation process.

The method used by the HI-TECH compiler to group and position different parts of a program is to place all assembler instructions and directives into individual, relocatable sections. These sections of a program are known as *psects* - short for program sections. The linker is then passed a series of options which indicate the memory that is available on the target system and how all the psects in the program should be positioned in this memory space.

2.2.1.1 The psect directive

The psect assembler directives (generated by the code generator or manually included in other assembly files) define a new psect. The general form of this directive is shown below.

```
psect name,option,option...
```

It consists of the token **`psect`** followed by the name by which this psect shall be referred. The name can be any valid assembler identifier and does not have to be unique. That is, you may have several psects with the same name, even in the same file. As will be discussed presently, psects with the same name are usually grouped together by the linker.

The directive options are described in the assembler section of the manual, but several of these will be discussed in this tutorial. The options are instructions to the linker which describe how the psect should be grouped and relocated in the final absolute object file.

Psects which all have the same name imply that their content is similar and that they should be grouped and linked together in the same way. This allows you to place objects together in memory even if they are defined in different files.

After a psect has been defined, the options may be omitted in subsequent psect directives in the same module that use the same name. The following example shows two psects being defined and filled with code and data.

```
psect text,global
begin:
    mov   r0,#10
    mov   r2,r4
    add   r2,#8
psect data
input:
    ds  8
```

```
psect text
next:
    mov  r4,r2
    rrc  r4
```

In this example, the psect **`text`** is defined including an option to say that this is a **`global`** psect. Three assembler instructions are placed into this psect. Another psect is created: **`data`**. This psect reserves 8 bytes of storage space for data in RAM. The last psect directive will continue adding to the first psect. The options were omitted from the **`psect`** directive in this example as there has already been a psect directive in this file that defines the options for a psect of this name. The above example will generate two psects. Other assembler files in the program may also create psects which have the same name as those here. These will be grouped with the above by the linker in accordance with the **`psect`** directive flags.

2.2.1.2 Psect types

Psects come in three broad types: those that will reside permanently in ROM[5]; those that will be allocated space in RAM after the program starts; and those that will reside in ROM, but which will be copied into another reserved space in RAM after the program starts. A combination of code - known as the *run-time* (or *startup*) code - and psect and linker options allow all this to happen.

Typically, psects placed into ROM contain instructions and constant data that cannot be modified. Those psects allocated space in RAM only are for global data objects that do not have to assume any non-zero value when the program starts, i.e. they are uninitialised. Those psects that have both a ROM image and space reserved in RAM are for modifiable, global data objects which are initialised, that is they contain some specific value when the program begins, but that value can be changed by the program during its execution.

The following C source shows two objects being defined. The object **`input`** will be placed into a data psect; the value 22 will reside in ROM and be copied to the RAM space allocated for **`input`** by the run-time code. The object **`output`** will not contribute directly to the ROM image. A an area of memory will be reserved for it in RAM and this area will be cleared by the run-time code (**`output`** will be assigned the value 0).

```
int input = 22;  // an initialised object
int output;      // an uninitialised object
```

Snippets from the assembler listing file show how the 8051XA compiler handles these two objects. Other compilers may produce differently structured code. The psect directive flags are discussed

5. The term "ROM" will be used to refer to any non-volatile memory.

presently, but note that for the initialised object, **`input`**, the code generator used a **`dw`** (define word) directive which placed the two bytes of the **`int`** value (16 and 00) into the output which is destined for the ROM. Two bytes of storage were reserved using the **`ds`** assembler directive for the uninitialised object, **`output`**, and no values appear in the output.

```
 1   0000'                psect data,class=CODE,space=0,align=0
 2                        global _input
 3                        align.w
 4   0000'             _input:
 5   0000' 16 00          dw 22

13   0000'                psect bss,class=DATA,space=1,align=0
14                        global _output
15                        align.w
16   0000'             _output:
17   0000'                ds 2
```

Auto variables and function parameters are local to the function in which they are defined and so are handled different by the compiler. The may be allocated space dynamically (for example on the stack) in which case they are not stored in psects by the compiler.

Two addresses are used to refer to the location of a psect: the *link address* and the *load address*. The link address is the address at which the psect (and any objects or labels within the psect) can be accessed whilst the program is executing. The load address is the address at which the psect will reside in the output file that creates the ROM image, or, alternatively, the address of where the psect can be accessed in ROM.

For the psect types that reside in ROM their link and load address are be the same as they reside in ROM and are never copied to a new location. Psects that are allocated space in RAM only will have a link address, but a load address is not applicable. The compiler often makes the load address of these psects the same as the link address. Since no ROM image of these psects is formed, the load address is meaningless and can be ignored. Any access to objects defined in these psects is performed using the link address. The psects that reside in ROM, but are copied to RAM have link and load addresses that are usually different. Any references to symbols or labels in these psects are always made using the link addresses.

2.3 Linking the psects

After the code generator and assembler[6] have finished their jobs, the object files passed to the linker can be considered to be a mixture of psects that have to be grouped and positioned in the available ROM and

RAM. The linker options indicate the memory that is available and the flags associated with a psect directive indicate how the psects are to be handled.

2.3.1 Grouping psects

There are two psect flags that affect the grouping, or merging, of the psects. These are the **LOCAL** and **GLOBAL** flags. **GLOBAL** is the default and tells the linker that the psects should be grouped together with other psects of the same name to form a single psect. **LOCAL** psects are not grouped in this way unless they are contained in the same module. Two local psects which have the same name, but which are defined in different modules are treated and positioned as separate psects.

2.3.2 Positioning psects

Several psect flags affect how the psects are positioned in memory. Psects which have the same name can be positioned in one of two ways: they can be overlaid one another, or they can be placed so that each takes up a separate area of memory.

Psects which are to be overlaid will use the **OVLRD** psect directive flag. At first it may seem unusual to have overlaid psects as they might destroy other psects' contents as they are positioned, however there are instances where this is desirable.

One case where overlaid psect are used is when the compiler has to use temporary variables. When the compiler has to pass several data objects to, say, a floating-point routine, the floats may need to be stored in temporary variables which are stored in RAM. It is undesirable to have the space reserved if it is not going to be used, so the routines that use the temporary objects are also responsible for defining the area and reserving the space in which these will reside. However several routines may called and hence several temporary areas created. To get around this problem, the psects which contain the directives to reserve space for the objects are defined as being overlaid so that if more than one is defined, they since simply overlap each other.

Another situation where overlaid psects are used is when defining the interrupt vectors. The run-time code usually defines the reset vector, but other vectors are left up to the programmer to initialize. Interrupt vectors are placed into a separate psect (often called **vectors**). Each vector is placed at an offset from the beginning of the vectors area appropriate for the target processor. The offset is achieved via an **ORG** assembler directive which moves the location counter relative to the beginning of the current psect. The macros contained in the header file <**intrpt.h>**, which allow the programmer to define additional interrupt vectors, also place the vectors they define into a psect with this same name, but with different offsets, depended on the interrupt vector being defined. All these psects are grouped and overlaid such that the vectors are correctly positioned from the same address - the start of the vectors psect. This merged psect is then positioned by the linker so that it begins at the start of the vectors area.

6. The assembler does not modify psect directives in any way other than encoding the details of each into the object file.

Most other compiler-generated psects are not overlaid and so they will each occupy their own unique address space. Typically these psects are placed one after the other in memory, however there are several psect flags that can alter the positioning of the psects. Some of these psect flags are discussed below.

The **RELOC** flag is used when psects must be aligned on a boundary in memory. This boundary is a multiple of the value specified with the flag. The **ABS** flag specifies that the psect is absolute and that it should start at address 0h. Remember, however, that if there are several psects which use this flag, then after grouping only one can actually start at address 0h unless the psects are also defined to be overlaid. Thus **ABS** means that one of the psects with this name will be located at address 0h, the others following in memory subject to any other psect flags used.

2.3.3 Linker options to position psects

The linker is told of the memory setup for a target program by the linker options that are generated by the compiler driver. The user informs the compiler driver about memory using either the **-A** option[7] with the command line driver (CLD), or via the **ROM & RAM addresses** dialogue box under HPD. Additional linker options indicate how the psects are to be positioned into the available memory.

The linker options are a little confusing at first, but the following example shows how the options could be built up as a program develops, and then discusses some of the specific schemes used by HI-TECH compilers. When compiling using either the CLD or HPD, a full set of default linker options are used, based on either the **-A** option values, or the **ROM & RAM addresses** dialogue values. In most cases the linker options do not need to be modified.

2.3.3.1 Placing psects at an address

Let us assume that the processor in a target system can address 64 kB of memory and that ROM, RAM and peripherals all share this same block of memory. The ROM is placed in the top 16 kB of memory (C000h - FFFFh); RAM is placed at addresses from 0h to FFFh.

Let us also assume that three object files passed to the linker: one a run-time object file; the others compiled from the programmer’s C source code. Each object file contains a compiler-generated **text** psect (a psect called **text**): the psect in one file is 100h bytes long; that from other file is 200h bytes long; that from the run-time object file is 50h long. These psects are to be placed in ROM and all have the simple definition generated by the code generator:

```
        psect text,class=CODE
```

7. The **-A** option on the PIC compiler serves a different purpose. Most PIC devices only have internal memory and so a memory option is not required by the compiler. High-end PICs may have external memory, this is indicated to the compiler by using a **-ROM** option to the CLD or by the **RAM & ROM addresses...** dialogue box under HPDPIC. The **-A** option is used to shift the entire ROM image, when using highend devices.

The **CLASS** flag is typically used with these types of psects and is considered later in this tutorial. By default, these psects are **GLOBAL**, hence after scanning all the object files passed to it, the linker will group all the **text** psects together so that they are contiguous[8] and form one larger **text** psect. The following **-p** linker option could be used to position the **text** psect at the bottom of ROM.

```
-ptext=0C000h
```

There is only one address specified with this linker option since the psects containing code are not copied from ROM to RAM at any stage and the link and load addresses are the same.

The linker will relocate the grouped **text** psect so that it starts at address C000h. The linker will then define two global symbols with names: **__Ltext** and **__Htext** and equate these with the values: C000h and C350h which are the start and end addresses, respectively, of the **text** psect group.

Now let us assume that the run-time file and one of the programmer's files contains interrupt vectors. These vectors must be positioned at the correct location for this processor. Our fictitious processor expects its vectors to be present between locations FFC0h and FFFFh. The reset vector takes up two bytes at address FFFEh an FFFFh, and the remaining locations are for peripheral interrupt vectors. The run-time code usually defines the reset vector using code like the following.

```
globalstart
psect vectors,ovlrd
org   3Eh
dw    start
```

This assembler code creates a new psect which is called **vectors**. This psect uses the overlaid flag (**OVLRD**) which tells the linker that any other psects with this name should be overlaid with this one, not concatenated with it. Since the psect defaults to being global, even **vectors** psects in other files will be grouped with this one. The **org** directive tells the assembler to advance **3Eh** locations into this psect. It does *not* tell the assembler to place the object at address **3Eh**. The final destination of the vector is determined by the relocation of the psect performed by the linker later in the compilation process. The assembler directive **dw** is used to actually place a word at this location. The word is the address of the (global) symbol **start**. (**start** or **powerup** are the labels commonly associated with the beginning of the run-time code.)

In one of the user's source files, the macro **ROM_VECTOR** has been used to supply one of the peripheral interrupts at offset **10h** into the vector area. The macro expands to the following in-line assembler code.

8. Some processors may require word alignment gaps between code or data. These gaps can be handled by the compiler, but are not considered here.

```
        global_timer_isr
        psect vectors,ovlrd
        org   10h
        dw    _timer_isr
```

After the first stages of the compilation have been completed, the linker will group together all the **`vectors`** psects it finds in all the object files, but they will all start from the same address, i.e. they are all placed one over the other. The final **`vectors`** psect group will contain a word at offset 10h and another at offset 3Eh. The space from 0h to offset 0Fh and in-between the two vectors is left untouched by the linker. The linker options required to position this psect would be:

```
        -pvectors=0FFC0h
```

The address given with this option is the base address of the vectors area. The **`org`** directives used to move within the **`vectors`** psects hence were with respect to this base address.

Both the user's files contain constants that are to be positioned into ROM. The code generator generates the following **`psect`** directive which defines the psect in which it store the values.

```
        psect const
```

The linker will group all these **`const`** psects together and they can be simply placed like the **`text`** psects. The only problem is: where? At the moment the **`text`** psects end at address C34Fh so we could position the const psects immediately after this at address C350h, but if we modify the program, we will have to continually adjust the linker options. Fortunately we can issue a linker option like the following.

```
        -ptext=0C000h,const
```

We have not specified an address for the psect **`const`**, so it defaults to being the address immediately after the end of the preceding psect listed in the option, i.e. the address immediately after the end of the **`text`** psect. Again, the **`const`** psect resides in ROM only, so this one address specifies both the link and load addresses.

Now the RAM psects. The user's object files contain uninitialised data objects. The code generator generates **`bss`** psects in which are used to hold the values stored by the uninitialised C objects. The area of memory assigned to the **`bss`** psect will have to be cleared before **`main()`** is executed.

At link time, all **`bss`** psects are grouped and concatenated. The psect group is to be positioned at the beginning of RAM. This is easily done via the following option.

```
        -pbss=0h
```

The address 0h is the psect's link address. The load address is meaningless, but will default to the link address. The run-time code will clear the area of memory taken up by the **`bss`** psect. This code will use the symbols __**`Lbss`** and __**`Hbss`** to determine the starting address and the length of the area that has to be cleared.

Both the user's source files contain initialised objects like the following.

```
int init = 34;
```

The value **`34`** has to be loaded into the object **`init`** before the **`main()`** starts execution. Another of the tasks of the run-time code is to initialise these sorts of objects. This implies that the initial values must be stored in ROM for use by the run-time code. But the object is a variable that can be written to, so it must be present in RAM once the program is running. The run-time code must then copy the initialised values from ROM into RAM just before **`main()`** begins. The linker will place all the initial values into ROM in exactly the same order as they will appear in RAM so that the run-time code can simply copy the values from ROM to RAM as a single block. The linker has to be told where in ROM these values should reside as it generates the ROM image, but is must also know where in RAM these objects will be copied to so that it can leave enough room for them and resolve the run-time addresses for symbols in this area.

The complete linker options for our program, including the positioning of the **`data`** psects, might look like:

```
-ptext=0C000h,const
-pvectors=0FFC0h
-pbss=0h,data/const
```

That is, the **`data`** psect should be positioned after the end of the **`bss`** psect in RAM. The address after the slash indicates that this psect will be copied from ROM and that its position in ROM should be immediately after the end of the **`const`** psect. As with other psects, the linker will define symbols __**`Ldata`** and __**`Hdata`** for this psect, which are the start and end link addresses, respectively, that will be used by the run-time code to copy the **`data`** psect group. However with any psects that have different link and load addresses, another symbol is also defined, in this case: __**`Bdata`**. This is the load address in ROM of the **`data`** psect.

2.3.3.2 Exceptional cases

The PIC compiler handles the data psects in a slightly different manner. It actually defines two separate psects: one for the ROM image of the data psects; the other for the copy in RAM. This is because the length of the ROM image is different to the length of the psect in RAM. (The ROM is wider than the RAM and values stored in ROM may be encoded as **`retlw`** instructions.) The linker options in this case will contain two separate entries for both psects instead of the one psect with different link and load addresses specified. The names of the data psects in RAM are similar to **`rdata_0`**; those in ROM are like **`idata_0`**. The digit refers to a RAM bank number.

The link and load addresses reported for psects that contain objects of type **`bit`** have slightly different meaning to ordinary link and load addresses. In the map file, the link address listed is the link address of the psect specified as a bit address. The load address is the link address specified as a byte address. Bit objects cannot be initialised, so separate link and load addresses are not required. The linker knows to handle these psects differently because of the **`BIT`** psect flag. Bit psects will be reported in the map file as having a *scale* value of 8. This relates to the number of objects that can be positioned in an addressable unit.

2.3.3.3 Psect classes

Now let us assume that more ROM is added to our system since the programmers have been busy and filled the 16 kB currently available. Several peripheral devices were placed in the area from B000h to BFFFh so the additional ROM is added below this from 7000h to AFFFh. Now there are two separate areas of ROM and we can no longer give a single address for the **`text`** psects.

What we can now do to take advantage of this extra memory is define several ranges of addresses that can be used by ROM-based psects. This can be done by creating a *psect class*. There are several ways that psects can be linked when using classes. Classes are commonly used by HI-TECH C compilers to position the code or **`text`** psects. Different strategies are employed by different compilers to better suit the processor architecture for which the compilation is taking place. Some of these schemes are discussed below. If you intend to modify the default linker options or generate your own psects, check the linker options and psect directives generated by the code generator for the specific compiler you are using.

A class can be defined using another linker option. For example to use the additional memory added to our system we could define a class using the linker option:

```
-ACODE=7000h-AFFFh,C000h-FFFFh
```

The option is a **`-A`** immediately followed by the class name and then a comma-separated list of address ranges. Remember this is an option to the linker, not the CLD. The above example defines two address ranges for a class called **`CODE`**.

Here is how drivers for the 8051, 8051XA and Z80 compilers define the options passed to the linker to handle the code psects. In large model the 8051 psect definitions for psects that contain code are as follows.

```
psect text,class=CODE
```

The **`CLASS`** psect flag specifies that the psect **`text`** is a member of the class called **`CODE`**.

If a single ROM space has been specified by either not using the **`-ROM`** option with the CLD or by selecting **single ROM** in the **ROM & RAM addresses** dialogue box under HPD, no class is defined and the psects are linked using a **`-p`** option as we have been doing above. Having the psects within a class, but not having that class defined is acceptable, provided that there is a **-p** option to explicitly position the psects after they have been grouped. If there is no class defined and no **-p** option a default memory address is used which will almost certainly be inappropriate.

If multiple ROM spaces have been specified by using either the **-ROM*ranges*** option with the CLD, or specifying the address ranges in the **ROM & RAM addresses** (after selecting the **multiple ROMs** button) dialogue box under HPD, a class is defined by the driver using the **-A** linker option similar to that shown above and the **-p** option is omitted from the options passed to the linker.

The PIC compiler does things a little differently as it has to contend with multiple ROM pages that are quite small. The PIC code generator defines the psects which hold code like the following.

```
psect text0,local,class=CODE,delta=2
```

The **DELTA** value relates to the ROM width and need not be considered here. The psects are placed in the **CODE** class, but note that the they are made local using the **LOCAL** psect flag. The psects that are generated from C functions each have unique names which proceed: **text0**, **text1**, **text2** etc. Local psects are not grouped across modules, i.e. if there are two modules, each containing a local psect of the same name, they are treated are separate psects. Local psects cannot be positioned using a **-p** linker option as there can be more than one psect with that name. Local psects must be made members of a class, and the class defined using a **-A** linker option. The PIC works in this way to assist with the placement of the code in its ROM pages. This is discussed further in Section 2.3.4 on page 49.

A few general rules apply when using classes: If, for example, you wanted to place a psect that is not already in a class into the memory that a class occupies, you can replace an address or psect name in a linker **-p** option with a class name. For instance, in the generic example discussed above, the **const** psect was placed after the **text** psect in memory. If you would now like this psect to be positioned in the memory assigned to the **CODE** class the following linker options could be used.

```
-pconst=CODE
-pvectors=0FFC0h
-pbss=0h,data/CODE
-ACODE=7000h-AFFFh,C000h-FFFFh
```

Note also that the **data** psect's load location has been swapped from after the end of the **const** psect to within the memory assigned to the **CODE** class to illustrate that the load address can be specified using the class name.

Another class definition that is sometimes seen in PIC linker options specifies three addresses for each memory range. Such an option might look something like:

```
-AENTRY=0h-FFh-1FFh
```

The first range specifies the address range in which the psect must start. The psects are allowed to continue past the second address as long as they do not extend past the last address. For the example above, all psects that are in the **ENTRY** class must start at addresses between 0 and FFh. The psects must end before address 1FFh. No psect may be positioned so that its starting address lies between 100h and 1FFh. The linker may, for example, position two psects in this range: the first spanning addresses 0 to 4Fh and the second starting at 50h and finishing at 138h. Such linker options are useful on some PIC processors (typically baseline PICs) for code psects that have to be accessible to instructions that modify the program counter. These instructions can only access the first half of each ROM page.

2.3.3.4 User-defined psects

Let us assume now that the programmer wants to include a special initialised C object that has to be placed at a specific address in memory, i.e. it cannot just be placed into, and linked with, the **data** psect. In a separate source file the programmer places the following code.

```
#pragma psect data=lut
int lookuptable[] = {0, 2, 4, 7, 10, 13, 17, 21, 25};
```

The pragma basically says, from here onwards in this module, anything that would normally go into the **data** psect should be positioned into a new psect called **lut**. Since the array is initialised, it would normally be placed into **data** and so it will be re-directed to the new psect. The psect **lut** will inherit any psect options (defined by the psect directive flags) which applied to **data**.

The array is to be positioned in RAM at address 500h. The **-p** option above could be modified to include this psect as well.

```
-pbss=0h,data/const,lut=500h/
```

(The load address of the **data** psect has been returned to its previous setting.) The addresses for this psect are given as "500h/". The address "500h" specifies the psect’s link address. The load address can be anywhere, but it is desirable to concatenate it to existing psects in ROM. If the link address is not followed by a load address at all, then the link and load addresses would be set to be the same, in this case 500h. The "/", which is not followed by an address, tells the linker that the load address should be immediately after the end of the previous psect’s load address in the linker options. In this case that is the **data** psect’s load address, which in turn was placed after the **const** psect. So, in ROM will be placed the **const**, **data** and **lut** psect, in that order.

Since this is an initialised data psect, it is positioned in ROM and must be copied to the memory reserved for it in RAM. Although different link and load addresses have been specified with the linker option, the programmer will have to supply the code that actually performs the copy from ROM to RAM. (The data psects normally created by the code generator have code already supplied in the run-time file to copy the psects.) The following is C code which could perform the copy.

```
extern unsigned char *_Llut, *_Hlut, *_Blut;
unsigned char *i;

void copy_my_psect(void)
{
      for(i=_Llut; i<_Hlut; i++, _Blut++)
            *i = *_Blut;
}
```

Note that to access the symbols __**Llut** etc. from within C code, the first underscore character is dropped. These symbols hold the addresses of psects, so they are declared (not defined) as pointer objects in the C code using the **extern** qualifier. Remember that the object **lookuptable** will not be initialised until this C function has been called and executed. Reading from the array before it is initialized will return incorrect values.

If you wish to have initialised objects copied to RAM before **main()** is executed, you can write assembler code, or copy and modify the appropriate routine in the run-time code that is supplied with the compiler. You can create you own run-time object file by pre-compiling the modified run-time file and using this object file instead of the standard file that is automatically linked with user's programs. From assembler, both the underscore characters are required when accessing the psect address symbols.

If you define your own psect based on a **bss** psect, then, in the same way, you will have to supply code to clear this area of memory if you are to assume that the objects defined within the psect will be cleared when they are first used.

2.3.4 Issues when linking

The linker uses a relatively complicated algorithm to relocate the psects contained in the object and library files passed to it, but the linker needs more information than that discussed above to know exactly how to relocate each psect? This information is contained in other the linker options passed to the linker by the driver and in the psect flags which are used with each **psect** directive. The following explain some of the issues the linker must take into account.

2.3.4.1 Paged memory

Let's assume that a processor has two ROM areas in which to place code and constant data. The linker will never split a psect over any memory boundary. A memory boundary is assumed to exist wherever there is a discontinuity in the address passed to the linker in the linker options. For example, if a class is specified using the addresses as follows:

```
-ADATA=0h-FFh,100h-1FFh
```

It is assumed that some boundary exists between address FFh and 100h, even though these addresses are contiguous. This is why you will see contiguous address ranges specified like this, instead of having one range covering the entire memory space. To make it easy to specify similar contiguous address ranges, a repeat count can be used, like:

```
-ADATA=0h-FFhx2
```

can be used; in this example, two ranges are specified: 0 to FFh and then 100h to 1FFh. Some processors have memory pages or banks. Again, a psect will not straddle a bank or page boundary.

Given that psects cannot be split over boundaries, having large psects can be a problem to relocate. If there are two, 1 kB areas of memory and the linker has to position a single 1.8 kB psect in this space, it will not be able to perform this relocation, even though the size of the psect is smaller than the total amount of memory available. The larger the psects, the more difficult it is for the linker to position them. If the above psect was split into three 0.6 kB psects, the linker could position two of them - one in each memory area - but the third would still not fit in the remaining space in either area. When writing code for processors like the PIC, which place the code generated from each C function into a separate, local psect, functions should not become too long.

If the linker cannot position a psect, it generates a "Can’t find space for psect xxxx" error, where xxxx is the name of the psect. Remember that the linker relocates psects so it will not report memory errors with specific C functions or data objects. Search the assembler listing file to identify which C function is associated with the psect that is reported in the error message if local psects are generated by the code generator.

Global psects that are not overlaid are concatenated to form a single psect by the linker before relocation takes place. There are instances where this grouped psect appears to be split again to place it in memory. Such instances occur when the psect class within which it is a member covers several address ranges and the grouped psect is too large to fit any of the ranges. The linker may use intermediate groupings of the psects, called *clutches* to facilitate relocation within class address ranges. Clutches are in no way controllable by the programmer and a complete understanding of there nature is not required to able to understand or use the linker options. It is suffice to say that global psects can still use the address ranges within a class. Note that although a grouped psect can be comprised of several clutches, an individual psect defined in a module can never be split under any circumstances.

2.3.4.2 Separate memory areas

Another issue faced by the linker is this: On some processors, there are distinct memory areas for program and data, i.e. Harvard architecture chips like the 8051XA. For example, ROM may extend from 0h - FFFFh and separate RAM may extend from 0h - 7FFh. If the linker is asked to position a psect at address 100h via a **`-p`** option, how does the linker know whether this is an address in program memory or in the data space? The linker makes use of the **`SPACE`** psect flag to determine this. Different areas are assigned a different space value. For example ROM may be assigned a **SPACE** value of 0 and RAM a **SPACE** flag of 1. The space flags for each psect are shown in the map file.

The space flag is not used when the linker can distinguish the destination area of an object from its address. Some processors use memory banks which, from the processors's point of view, cover the same range of addresses, but which are within the same distinct memory area. In these cases, the compiler will assign unique addresses to objects in banked areas. For example, some PIC processors can access four banks of RAM, each bank covering addresses 0 to 7Fh. The compiler will assign objects in the first bank (bank 0) addresses 0 to 7Fh; objects in the second bank: 80h to FFh; objects in the third bank: 100h to 17Fh etc. This extra bank information is removed from the address before it is used in an assembler instruction. All PIC RAM banks use a **SPACE** flag of 1, but the ROM area on the PIC is entirely separate and uses a different **SPACE** flag (0). The space flag is not relevant to psects which reside in both memory areas, such as the data psects which are copied from ROM to RAM.

After relocation is complete, the linker will group psects together to form a *segment*. Segments, along with clutches, are rarely mentioned with the HI-TECH compiler simply because they are an abstract object used only by the linker during its operation. Segment details will appear in the map file. A segment is a collection of psects that are contiguous and which are destined for a specific area in memory. The name of a segment is derived from the name of the first psect that appears in the segment and should not be confused with the psect which has that name.

2.3.4.3 Objects at absolute addresses

After the psects have been relocated, the addresses of data objects can be resolved and inserted into the assembler instructions which make reference to an object's address. There is one situation where the linker does not determine and resolve the address of a C object. This is when the object has been defined as absolute in the C code. The following example shows the object **DDRA** being positioned at address 200h.

```
unsigned char DDRA @ 0x200;
```

When the code generator makes reference to the object **DDRA**, instead of using a symbol in the generated assembler code which will later be replaced with the object's address after psect relocation, it will immediately use the value 200h. The important thing to realise is that the instructions in the assembler that access this object will not have any symbols that need to be resolved, and so the linker will simply skip over them as they are already complete. If the linker has also been told, via its linker options, that there is memory available at address 200h for RAM objects, it may very well position a psect such that an object that resides in this psect also uses address 200h. As there is no symbol associated with the absolute object, the linker will not see that two objects are sharing the same memory. If objects are overlapping, the program will most likely fail unpredictably.

When positioning objects at absolute address, it vital to ensure that the linker will not position objects over those defined as absolute. Absolute objects are intended for C objects that are mapped over the top of hardware registers to allow the registers to be easily access from the C source. The programmer must ensure that the linker options do not specify that there is any general-purpose RAM in the memory space taken up by the hardware. Ordinary variables to be positioned at absolute addresses should be done so using a separate psect (by simply defining your own using a **psect** directive in assembler code, or by using the **`#pragma psect`** directive in C code) and linker option to position the objects. If you must use an absolute address for an object in general-purpose RAM, make sure that the linker options are modified so that the linker will not position other psects in this area.

2.3.5 Modifying the linker options

In most applications, the default linker options do not need to be modified. It is recommended that if you think the options should be modified, but you do not understand how the linker options work, that you seek technical assistance in regard to the problem at hand.

If you do need to modify the linker options, there are several ways to do this. If you are simply adding another option to those present by default, the option can be specified to the CLD using a **`-L`** option. To position the **`lut`** psect that was used in the earlier example, the following option could be used.

```
-L-plut=500/const
```

The **`-L`** simply passes whatever follows to the linker. If you want to add another option to the default linker options and you are using HPD and a project, then it is a simple case of opening the **linker options...** dialogue box and adding the option to the end of those already there. The options should be entered exactly as they should be presented to the linker, i.e. you do not need the **`-L`** at the front.

If you want to modify existing linker options other than simply changing the memory address that are specified with the **`-A`** CLD option, then you cannot use the CLD to do this directly. What you will need to do is to perform the compilation and link separately. Let's say that the file **`main.c`** and **`extra.c`** are to be compiled for the 8051 with modified linker options. First we can compile up to, but not include, the link stage by using a command line something like this.

```
c51 -o -Zg -asmlist -C main.c extra.c
```

The **`-C`** options stops the compilation before the link stage. Include any other options which are normally required. This will create two object files: **`main.obj`** and **`extra.obj`**, which then have to be linked together.

Run the CLD again in verbose mode by giving a **`-V`** option on the command line, passing it the names of the object files created above, and redirect the output to a file. For example:

```
c51 -v -A8000,0,100,0,0 main.obj extra.obj > main.lnk
```

Note that if you do not give the **`-A`** CLD option, the compiler will prompt you for the memory addresses, but with the output redirected, you will not see the prompts.

The file generated (**`main.lnk`**) will contain the command line that CLD generated to run the linker with the memory values specified using the **`-A`** option. Edit this file and remove any messages printed by the compiler. Remove the command line for any applications run after the link stage, for example **`objtohex`** or **`cromwell`**, although you should take note of what these command lines are as you will need to run these applications manually after the link stage. The linker command line is typically very long and if a DOS batch file is used to perform the link stage, it is limited to lines 128 characters long. Instead the linker can be passed a command file which contains the linker options only. Break up the linker command line in the file you have created by inserting backslash characters "\" followed by a return. Also remove the name and path of the linker executable from the beginning of the command line so that only the options remain. The above command line generated a **`main.lnk`** file that was then edited as suggested above to give the following.

```
-z -pvectors=08000h,text,code,data,const,strings \
-prbit=0/20h,rbss,rdata/strings,irdata,idata/rbss \
-pbss=0100h/idata -pnvram=bss,heap -ol.obj \
-m/tmp/06206eaa /usr/hitech/lib/rt51--ns.obj main.obj \
extra.obj /usr/hitech/lib/51--nsc.lib
```

Now, with care, modify the linker options in this file as required by your application.

Now perform the link stage by running the linker directly and redirecting its arguments from the command file you have created.

```
hlink < main.lnk
```

This will create an output file called **`l.obj`**. If other applications were run after the link stage, you will need to run them to generate the correct output file format, for example a HEX file.

Modifying the options to HPD is much simpler. Again, simply open the **linker options...** dialogue box and make the required changes, using the buttons at the bottom of the box to help with the editing. Save and re-make your project.

The map file will contain the command line actually passed to the linker and this can be checked to confirm that the linker ran with the new options.

Using HPDZ

3

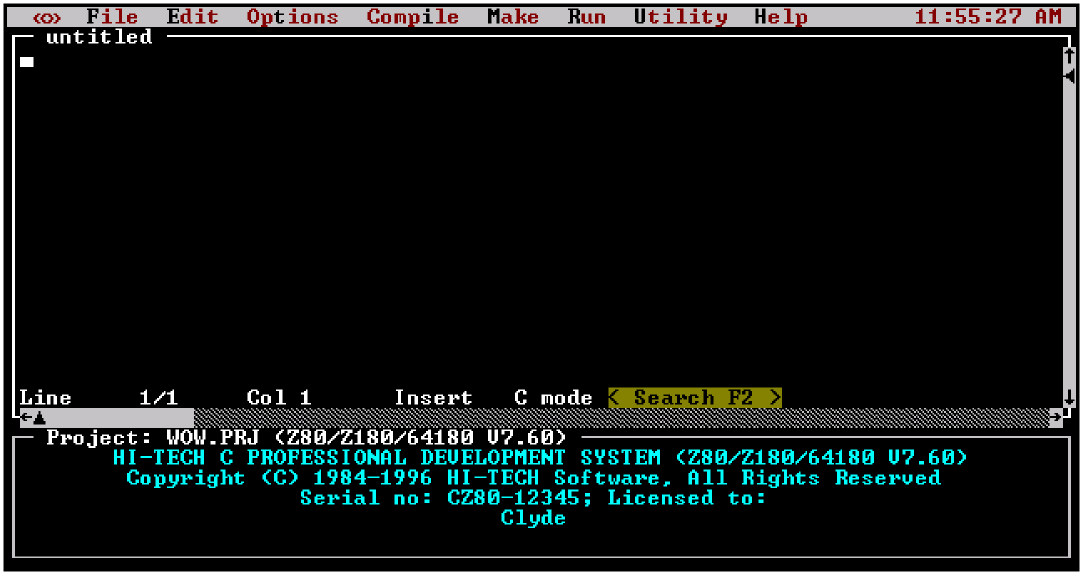
3.1 Introduction

This chapter covers HPD, the **HI**-TECH C **P**rogrammer's **D**evelopment system integrated environment. It assumes that you have already installed your HI-TECH C compiler. If you haven't installed your compiler go to chapter 1, Introduction, and follow the installation instructions there. HPDZ is the version of HPD applicable to the Z80 compiler.

3.1.1 Starting HPDZ

To start HPDZ, simply type HPDZ at the DOS prompt. After a brief period of disk activity you will be presented with a screen similar to the one shown in Figure 3 - 1 on page 55.

Figure 3 - 1 HPDZ Startup Screen



<<>> File Edit Options Compile Make Run Utility Help 11:55:27 AM
untitled
Search F2

```
Line      1/1      Col 1       Insert    C mode
Project: WOW.PRJ (Z80/Z180/64180 V7.60)
     HI-TECH C PROFESSIONAL DEVELOPMENT SYSTEM (Z80/Z180/64180 V7.60)
      Copyright (C) 1984-1996 HI-TECH Software, All Rights Reserved
               Serial no: CZ80-12345; Licensed to:
                              Clyde
```

The initial HPDZ screen is broken into three windows. The top window contains the menu bar, the middle window the HPDZ text editor and the bottom window is the message window. Other windows may appear when certain menu items are selected. The editor window is what you will use most of the time.

HPDZ uses the HI-TECH Windows user interface to provide a text screen based, user interface. This has multiple overlapping windows and pull down menus. The user interface features which are common to all HI-TECH Windows applications are described later in this chapter.

Alternatively, HPDZ can use a single command line argument. This 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`, HPDZ 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, HPDZ 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 HPDZ is invoked with the command `HPDZ 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.

3.2 The HI-TECH Windows user interface

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

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 HPDZ is executed. Higher modes can also be used with a `/screen:xx` option as described below. Problems may be experienced with some poorly written VGA utilities. These may 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 HPDZ set the screen display mode on EGA or VGA displays to show more than 25 lines. The option **/SCREEN:nn** where **nn** is one of 25, 28, 43 or 50 will cause HPDZ to set the display to that number of lines, or as close as possible. EGA display supports 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 HPDZ exits. The selected number of lines will be saved in the `HPDZ.INI` file and used for subsequent invocations of HPDZ unless overridden by another **/SCREEN** option.

HPDZ 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. Some older mouse drivers are missing a number of the driver status calls. If you are using such a mouse driver, HPDZ will still work with the mouse, but the *Mouse Setup* dialog in the **<<>>** menu will not work.

3.2.2 Colours

Colours are used in two ways in HPDZ. First, there are colours associated with the screen display. These can be changed to suit your own preference. The second use of colour is to optionally code text entered into the text window. This assists you to see the different elements of a program as it is entered and compiled. These colours can also be changed to suit your requirements. Colours comprise two elements, the actual colour and its attributes such as bright or inverse. Table 3 - 1 on page 57 shows the colours and their values, whilst Table 3 - 2 on page 58 shows the attributes and their meaning.

Table 3 - 1 Colour values

Value	Colour
0	black
1	blue
2	green
3	cyan
4	red
5	magenta
6	brown
7	white
8	grey
9	bright blue
10	bright green
11	bright cyan
12	bright red
13	bright magenta
14	yellow
15	bright white

Any colours are valid for the foreground but only colours 0 to 7 are valid for the background. Table 3 - 3 on page 58 shows the definition settings for the colours used by the editor when colour coding is selected.

The standard colour schemes for both the display colours and the text editor colour coding can be seen in the colour settings section of the HPDZ.ini file. The first value in a colour definition is the foreground

Table 3 - 2 Colour attributes

Attribute	description
normal:	normal text colour
bright:	bright/highlighted text colour
inverse:	inverse text colour
frame:	window frame colour
title:	window title colour
button:	colour for any buttons in a window

Table 3 - 3 Colour coding settings

Setting	Description
C_wspace:	White space - foreground colour affects cursor
C_number:	Octal, decimal and hexadecimal numbers
C_alpha:	Alphanumeric variable, macro and function names
C_punct:	Punctuation characters etc.
C_keyword:	C keywords and variable types: eg int, static, etc.
C_brace:	Open and close braces: { }
C_s_quote:	Text in single quotes
C_d_quote:	Text in double quotes
C_comment:	Traditional C style comments: /* ... */
Cpp_comment	C++ style comments: // ...
C_preprocessor:	C pre-processor directives: #blah
Include_file:	Include file names
Error:	Errors - anything incorrect detected by the editor
Asm_code:	Inline assembler code (#asm...#endasm)
Asm-comment:	Assembler comments: ; ...

colour and the second is the background colour. To set the colours to other than the default sets you should remove the # before each line, then select the new colour value.

The `.ini` file also contains an example of an alternative standard colour scheme. The same process can be used to set the colour scheme for the menu bars and menus.

3.2.3 Pull-down menus

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

The menu system can be used with the keyboard, mouse, or a combination of mouse and keyboard actions. The keyboard and mouse actions that are supported are listed in Table 3 - 4 on page 59.

Table 3 - 4 Menu system key and mouse actions

Action	Key	Mouse
Open menu	Alt-space	Press left button in menu bar or press middle button anywhere in screen
Escape from menu	Alt-space or Escape	Press left button outside menu system displays
Select item	Enter	Release left or centre button on highlighted item or 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

3.2.3.1 Keyboard menu selection

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

3.2.3.2 Mouse menu selection

Selecting a menu item using the mouse appears to be somewhat awkward to new users. However, it soon becomes second nature. To open the menu system, move the pointer to the title of the menu which you require and press the left button. You can 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. You may also operate the menu system with the middle button on three button mice. Press the middle button to bring the menu bar to the front. This makes 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. Then you can select 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, you can select by dragging the mouse to the desired item and releasing the button.

3.2.3.3 Menu hot keys

When browsing through the menu system you will notice that some menu items have *hot key* sequences displayed. For example, the HPDZ menu item *Save* has the key sequence **alt-S** shown as part of the display. 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 HPDZ **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 monochrome displays) and are highlighted with inverse video when the **alt** key is depressed. A full list of HPDZ key equivalents is shown in Table 3 - 5 on page 61.

3.2.4 Selecting windows

HI-TECH Windows allows you to overlap or tile windows. 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.

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 HPDZ 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 at the front, so the program text can be altered.

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

Table 3 - 5 HPDZ 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-J	DOS Shell
Alt-F	Open File menu
Alt-E	Open Edit menu
Alt-I	Open Compile menu
Alt-M	Open Make menu
Alt-R	Open Run menu
Alt-T	Open Options menu
Alt-U	Open Utility menu
Alt-H	Open Help menu
Alt-P	Open Project file
Alt-W	Warning level dialog
Alt-Z	Optimization menu
Alt-D	Command.com
F3	Compile and link single file
Shift-F3	Compile to object file
Ctrl-F3	Compile to assembler code
Ctrl-F4	Retrieve last file
F5	Make target program
Shift-F5	Re-link target program
Ctrl-F5	Re-make all objects and 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
Alt-X	Cut to clipboard
Alt-C	Copy to clipboard
Alt-V	Paste from clipboard

3.2.5 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 would not make sense. The HPDZ calculator is an example of a window which can be moved but not resized.

Windows can be moved and resized using the keyboard or the mouse. Using the keyboard, move/resize mode can be entered by pressing **ctrl-alt-space**. The application will respond by replacing the menu bar with the move/resize menu strip. This allows the front most window to be moved and resized. when the resizing is complete you should press **Enter** to return to the operating function of the window. A full list of all the operating keys is shown in Table 3 - 6 on page 62.

Table 3 - 6 Resize mode keys

Key	Action
Left arrow	Move window to right
Right arrow	Move window to left
Up arrow	Move window upwards
Down arrow	Move window downwards
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

Move/resize mode can be exited with any normal application action, like a mouse click, pressing a hot key or menu system activation by pressing **alt-space**. There are other ways of moving and resizing windows:

❐ Windows can be moved and resized using the mouse. You can move any visible window 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. Then you can move the window in any direction, including diagonally. If a window is grabbed near the middle of the top or bottom edge the pointer will change to a vertical arrow. Then you can 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. Then 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. Instead it activates 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.You can resize any visible window by pressing the right mouse button on its bottom or left frame. Then drag the frame to a new boundary and release the button. If a window is grabbed near its lower right corner the pointer changes to a diamond and it is be possible to resize the window in any direction. If the frame is grabbed anywhere else on the bottom edge, it is only possible to resize vertically. If the window is grabbed anywhere else on the right edge it is only possible to resize horizontally. If the right button is pressed anywhere in the top or left edges nothing will happen.
- ❒ You can also *zoom* a window to its maximum size. The front most window can be zoomed by pressing **shift-(keypad)+**, if it is zoomed again it reverts to its former size. In either the zoomed or unzoomed state the window can be moved and resized. Zoom effectively toggles between two user defined sizes. You can also zoom a window by clicking the right mouse button in its content region.

3.2.6 Buttons

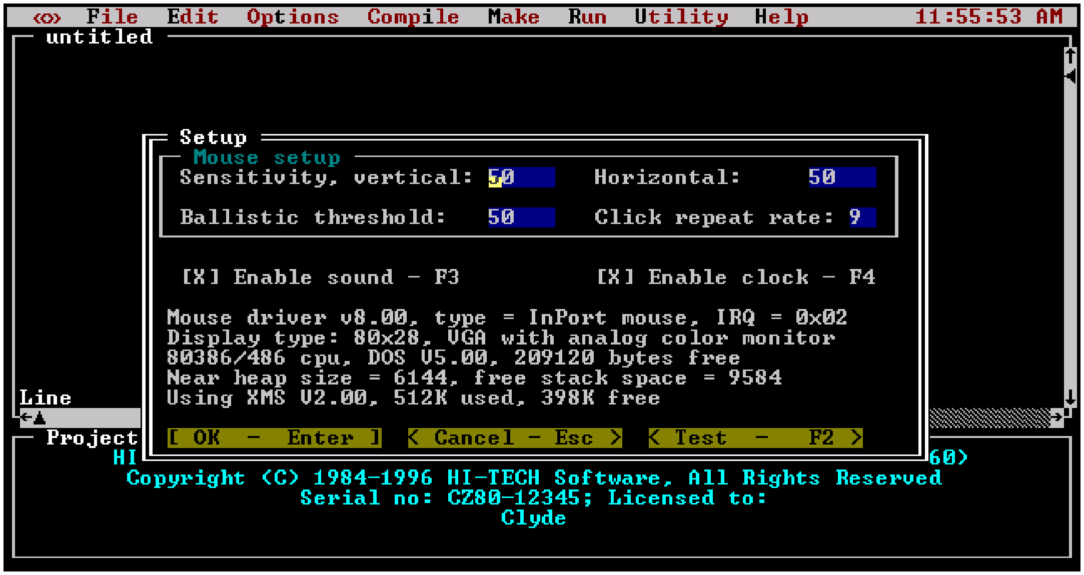
Some windows contain *buttons* which can be used to select particular actions immediately. Buttons are like menu items which are always visible and selectable. A button can be selected either by clicking the left mouse button on it or by using its key equivalent. The key equivalent to a button will either be displayed as part of the button, or as part of a help message somewhere else in the window. For example, the HPDZ error window (Figure 3 - 6 on page 68) contains a number of buttons, to select HELP you would either click the left mouse button on it or press **F1**.

3.2.7 The Setup menu

If you open the system menu, identified by the symbol <<>> on the menu bar, you will find two entries: the *About HPDZ* entry, which displays information about the version number of HPDZ; and the *Setup* entry. Selecting the Setup entry opens a dialog box as shown in Figure 3 - 2 on page 64. This box displays information about HPDZ's memory usage, 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 HPDZ's initialisation file, so they will be reloaded next time you run HPDZ.The sound and clock settings will be stored in the initialisation file if you select OK.

3

Figure 3 - 2 Setup Dialogue



<o> File Edit Options Compile Make Run Utility Help 11:55:53 AM
untitled
Line
Project
[OK - Enter]
< Cancel - Esc >
< Test - F2 >

```
Setup
 Mouse setup
 Sensitivity, vertical: 50        Horizontal:         50

 Ballistic threshold:   50        Click repeat rate:  9

 [X] Enable sound - F3            [X] Enable clock - F4

Mouse driver v8.00, type = InPort mouse, IRQ = 0x02
Display type: 80x28, VGA with analog color monitor
80386/486 cpu, DOS V5.00, 209120 bytes free
Near heap size = 6144, free stack space = 9584
Using XMS V2.00, 512K used, 398K free
```

```
HI                                                    60>
  Copyright (C) 1984-1996 HI-TECH Software, All Rights Reserved
             Serial no: CZ80-12345; Licensed to:
                           Clyde
```

3.3 Tutorial: Creating and compiling a C program

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

Before starting HPDZ, you 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> HPDZ
```

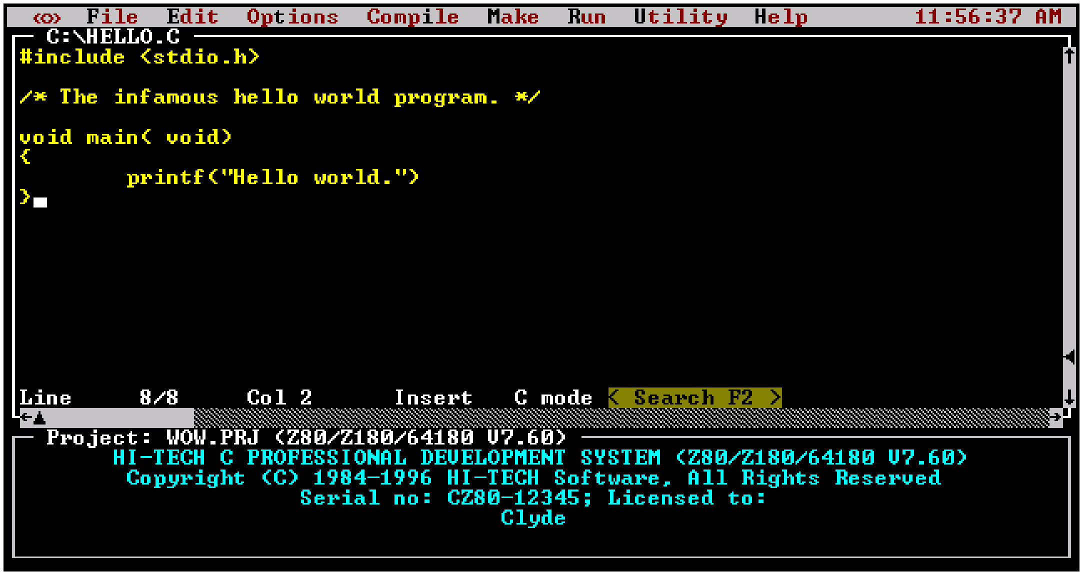
You will be presented with the HPDZ startup screen. At this stage, the editor is ready to accept whatever text you type. A flashing block cursor should be visible in the top left corner of the edit window. You are now ready to enter your first C program using HPDZ. This will naturally be the infamous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

Type the following text, pressing enter once at the end of each line. You can enter blank lines by pressing enter without typing any text.

```
#include  <stdio.h>

main()
{
        printf("Hello, world")
}
```

Note that a semi-colon has been deliberately omitted from the end of the printf() statement in order to demonstrate HPDZ's error handling facilities. Figure 3 - 3 on page 65, shows the screen as it should appear after entry of the "Hello world" program).

Figure 3 - 3 Hello program in HPDZ


‹O› File Edit Options Compile Make Run Utility Help 11:56:37 AM
C:\HELLO.C

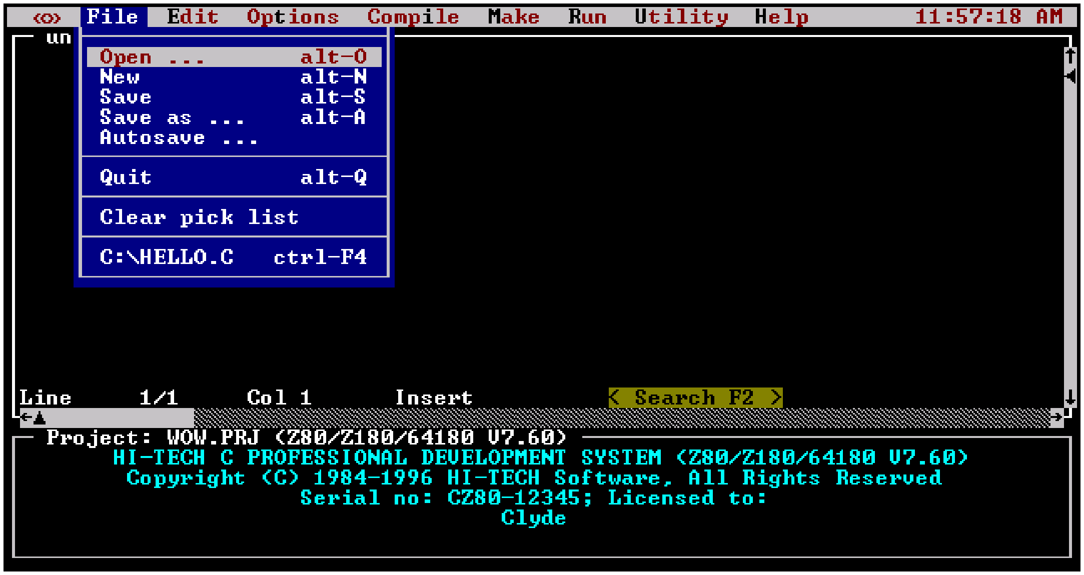
```
#include <stdio.h>

/* The infamous hello world program. */

void main( void)
{
        printf("Hello world.")
}
```

```
Line      8/8      Col 2      Insert    C mode  < Search F2 >
Project: WOW.PRJ (Z80/Z180/64180 V7.60)
     HI-TECH C PROFESSIONAL DEVELOPMENT SYSTEM (Z80/Z180/64180 V7.60)
       Copyright (C) 1984-1996 HI-TECH Software, All Rights Reserved
             Serial no: CZ80-12345; Licensed to:
                              Clyde
```

You now have a C program (complete with one error!) entered and almost ready for compilation. All you 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 - 4 on page 66)

If you do not have a mouse, follow these steps:

Figure 3 - 4 HPDZ File Menu



```
<◊>  File  Edit  Options  Compile  Make  Run  Utility  Help        11:57:18 AM
      un
       Open ...          alt-O
       New               alt-N
       Save              alt-S
       Save as ...       alt-A
       Autosave ...

       Quit              alt-Q

       Clear pick list

       C:\HELLO.C       ctrl-F4

Line      1/1      Col 1          Insert            < Search F2 >
 Project: WOW.PRJ (Z80/Z180/64180 V7.60)
     HI-TECH C PROFESSIONAL DEVELOPMENT SYSTEM (Z80/Z180/64180 V7.60)
      Copyright (C) 1984-1996 HI-TECH Software, All Rights Reserved
                 Serial no: CZ80-12345; Licensed to:
                                Clyde
```

❒ Open the menu system by pressing **alt-Space**

❒ Move to the **Edit** menu using the **right arrow** key

❒ Move down to the **Save** item using the **down arrow** key

❒ When the **Save** item is highlighted, select it by pressing the **Enter** key.

If you are using the mouse, follow these steps:

❒ Open the **File** menu by moving the pointer to the word **File** in the menu bar and pressing the left button

❒ Highlight the **Save** item by dragging the mouse downwards with the left button held down, until the **Save** item is highlighted

❒ When the **Save** item is highlighted, select it by releasing the left button.

When the **File** menu (Figure 3 - 4 on page 66) was open, you may have noticed that the **Save** item included the text **alt-S** 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.

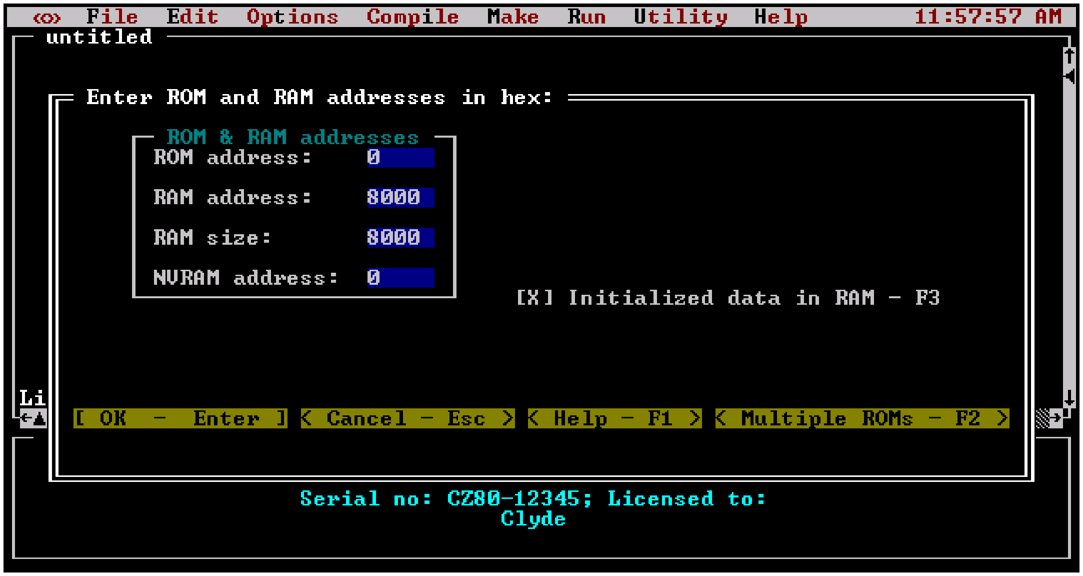
After **Save** has been selected, you should be presented with a *dialog* prompting you for the file name. If HPDZ needs more information, such as a file name, before it is able to act on a command, it will always prompt you with a standard dialog like the one below.

The dialog contains an *edit line* where you can enter the file name to be used, and a number of *buttons*. These may be used to perform various actions within the dialog. A button may be selected 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, save your C program to a file called “hello.c”. Type **hello.c** and then press **Enter**. There should be a brief period of disk activity as HPDZ saves the file.

You need to set the memory configuration. To do this select ***ROM & RAM addresses...*** from the **Options** menu. You should use 0 for the ROM address, 8000 for the RAM address and 8000 for the RAM size. If you are not using non-volatile RAM or all your RAM is non-volatile set the NVRAM to 0. Once you have set the memory configuration the dialog should look like Figure 3 - 5 on page 67.

Figure 3 - 5 ROM and RAM Address dialog



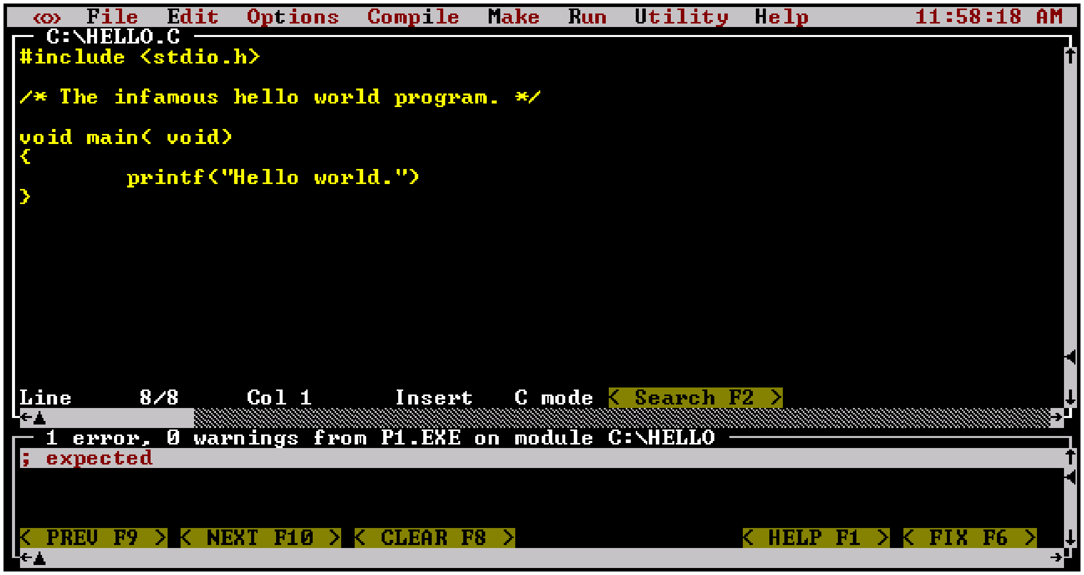
<<>> File Edit Options Compile Make Run Utility Help 11:57:57 AM
untitled
Li

```
Enter ROM and RAM addresses in hex:

  ROM & RAM addresses
  ROM address:      0
  RAM address:      8000
  RAM size:         8000
  NVRAM address:    0
                                [X] Initialized data in RAM - F3

[ OK  -  Enter ] < Cancel - Esc > < Help - F1 > < Multiple ROMs - F2 >

Serial no: CZ80-12345; Licensed to:
Clyde
```

You 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.

This time, the compiler will not run to completion. This is because we deliberately omitted a semicolon on the end of a line, in order to see how HPDZ 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. The message window will be replaced by a window containing a number of buttons and the message "`; expected`", as shown in Figure 3 - 6 on page 68.

Figure 3 - 6 Error window



<<>> File Edit Options Compile Make Run Utility Help 11:58:18 AM
C:\HELLO.C
Line 8/8 Col 1 Insert C mode
< Search F2 >
< PREV F9 >
< NEXT F10 >
< CLEAR F8 >
< HELP F1 >
< FIX F6 >

```
#include <stdio.h>

/* The infamous hello world program. */

void main( void)
{
        printf("Hello world.")
}
```

```
1 error, 0 warnings from P1.EXE on module C:\HELLO
; expected
```

The text in the frame of the error window shows the number of compiler errors generated, and which phase of the compiler generated them. Most errors will come from P1.EXE and CGxx.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 HPDZ 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.

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. This 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 HELP button with a mouse click, or press **F1**.

The error explanation for the missing semi-colon does not give much more information than we already have. 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 explanations by selecting the HIDE button (press **Escape** or use the mouse).

In this instance HPDZ has analysed 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 HPDZ is unable to analyse 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, HPDZ will move to the next error.

To manually correct the error, move the cursor to the end of the printf() statement and add the missing semi-colon. 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 semi-colon 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. HPDZ 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`.

This tutorial has presented a simple overview of single file edit/compile development. HPDZ 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 HPDZ menu system, editor and project facility.

3.4 The HPDZ editor

HPDZ 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 HPDZ editor without further instruction. HPDZ also supports the standard PC keys, and thus should be readily usable by anyone familiar with typical MS-DOS or Microsoft Windows editors.

The HPDZ 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*.

3.4.1 Frame

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.

3.4.2 Content region

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.

3.4.3 Status line

The bottom line of the edit window is the status line. It contains the following information about the file being edited:

❒ `Line` shows the current line number, counting from the start of the file, and the total number of lines in the file.

❒ `Col` 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 section - Keyboard commands on page 71, 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 section Keyboard commands on page 71, for a list of the valid **ctrl-Q** commands.

❒ `Insert` indicates whether text typed on the keyboard will be inserted at the cursor position. Using the *insert mode toggle* command (the **Ins** key on the keypad, or **ctrl-V**), the mode can be toggled between *Insert* and *Overwrite*. In overwrite mode, text entered on the keyboard will overwrite characters under the cursor, instead of inserting them before the cursor.

❒ `Indent` indicates that the editor is in auto indent mode. Auto indent mode is toggled using the **ctrl-Q I** key sequence. By default, auto indent mode is enabled. When auto indent mode is enabled, every time you add a new line the cursor is 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 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 `PREVIOUS` 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 `PREVIOUS` are **ctrl-F2** and **ctrl-P**.

3.4.4 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 - 8 on page 73 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 HPDZ menu bar may still be accessed either by pressing the **Alt** key or by using the middle button on a three button mouse.

3.4.5 Block commands

In addition to the movement and editing command listed in the “Editor Keys” table, the HPDZ 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 - 7 on page 72 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

Table 3 - 7 Block operation keys

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

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.

Table 3 - 8 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
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 of cursor	Backspace	Ctrl-H
Delete line		Ctrl-Y
Delete to end of line		Ctrl-Q Y
Search	F2	Ctrl-Q F
Search forward	Shift-F2	Crtl-L
Search backward	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	

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 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 opposite of 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 only accessible via the *Edit* menu.

Comment/Uncomment
Also available 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.

3.4.6 Clipboard editing

The HPDZ 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.

3.4.6.1 Selecting Text

Blocks of text may be selected using left mouse button and click or drag operations. The following mouse operations may be used:

❒ 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.

❒ 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.

❒ 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.

3.4.6.2 Clipboard commands

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

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

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

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

Hide **ctrl-K H**

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

Show clipboard

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

Clear clipboard

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

Delete selection

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

3.5 HPDZ menus

This section presents a item-by-item description of each of the HPDZ menus. The description of each menu includes a screen print showing the appearance of the menu within a typical HPDZ screen.

3.5.1 <<>> menu

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

About HPDZ ...

The About HPDZ dialog displays information on the version number of the compiler and the licence details.

Setup ...

This menu item selects the standard mouse firmware configuration menu, and is present in all HI-TECH Windows based applications. The “mouse setup” dialog 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.

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.

This menu item will not be selectable if there is no mouse driver installed. With some early mouse drivers, this dialog 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 dialog will also display information about what kind of video card and monitor you have, what DOS version is used and free DOS memory available. See Figure 3 - 2 on page 64

3.5.2 File menu

The **File** menu contains file handling commands, the HPDZ **Quit** command and the pick list:

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 dialog. If the previous edit file has been modified but not saved, you will be given an opportunity to save it or abort the Open 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 New command.

Save **alt-S**
This command saves 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.

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

Quit **alt-Q**
The **Quit** command is used to exit from HPDZ to the operating system. If the current edit file has been modified but not saved, you will be given an opportunity to save it or abort the Quit command.

Clear pick list
This clears the list of recently-opened files which appear below this option.

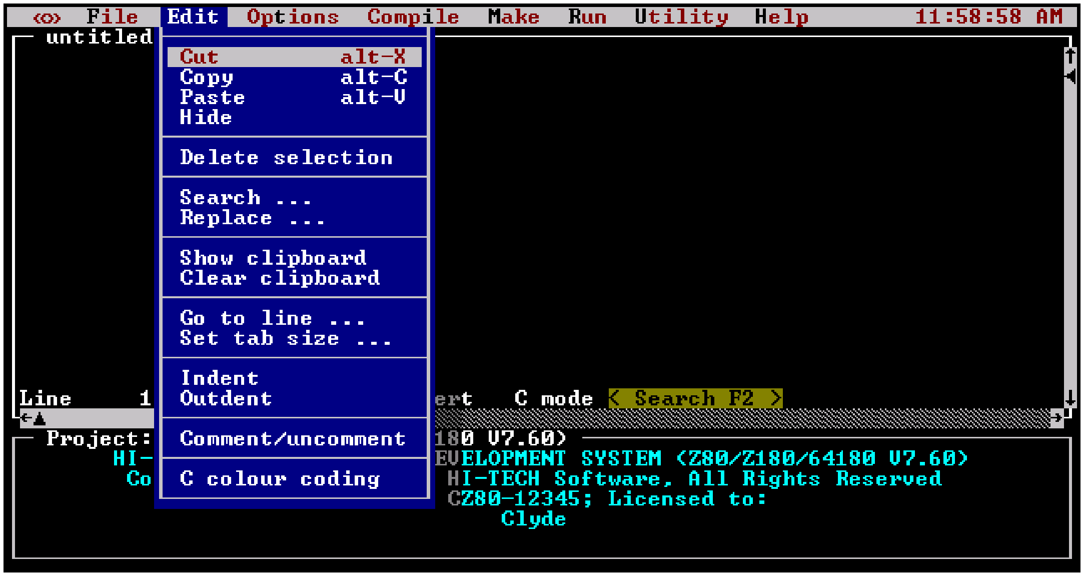
Pick list **ctrl-F4**

The pick list contains a list of the most recently-opened files. A file may be loaded from the pick list by selecting that file. The last file that was open may be retrieved by using the short-cut ctrl-F4.

3.5.3 Edit menu

The Edit menu contains items relating to the text editor and clipboard. The edit menu is shown inTable 3 - 7 on page 78.

Figure 3 - 7 HPDZ Edit Menu



```
HI-                                180 V7.60)
Co                                EVELOPMENT SYSTEM (Z80/Z180/64180 V7.60)
                                    HI-TECH Software, All Rights Reserved
                                    CZ80-12345; Licensed to:
                                        Clyde
```

Cut **alt-X**

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

Copy **alt-C**

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

Paste **alt-V**

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

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

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

Search ...
This option produces a dialog to allow you to enter a string for a search. You can select to search forwards or backwards by selecting the appropriate button. You can also decide if the search should be case sensitive and if a replacement string is to be substituted. You make these choices by clicking in the appropriate brackets.

Replace ...
This option is almost the same as the search option. It is used where you are sure you want to search and replace in the one operation. You can choose between two options. You can search and then decide whether to replace each time the search string is found. Alternatively, you can search and replace globally. If the global option is chosen, you should be careful in defining the search string as the replace can not be undone.

Show clipboard
This menu options hides or displays the clipboard editor window. If the clipboard window is already visible, it will be hidden. If the clipboard window is currently 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 ...
The **Go to line** command allows you to go directly to any line within the current edit file. You will be presented with a dialog prompting you for the line number. The title of the dialog will tell you the allowable range of line numbers in your source file.

Set tab size ...
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 also a 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.

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
This option toggles the colour coding of text in the editor window. It turns on and off the colours for the various types of text. A mark appears before this item when it is active. For a full description of colours used in HPDZ and how to select specific schemes, you should refer to Colours on page 57.

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 on page 81..

Memory model and chip type...
This option activates a dialog box which allows you to select the memory model and processor type you wish to use.

Available memory models are *small* and *banked* (large). For more information on memory models, see -Bs: Select Small Memory Model on page 101 and -Bl: Select Large Memory Model on page 101.

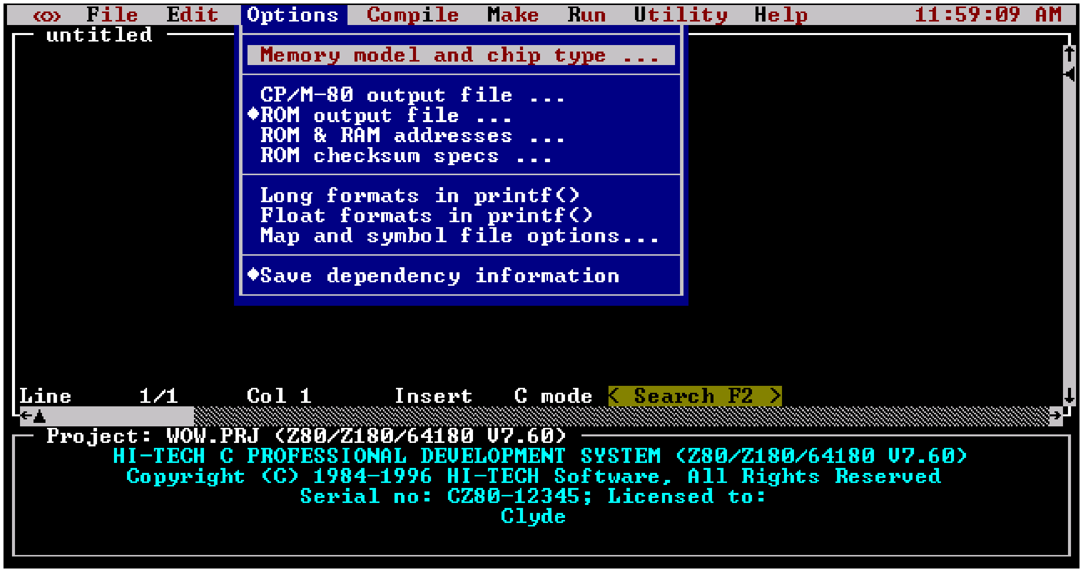
Processor types available are *Z80* and *Z180/64180*.

This dialog box also allows you to select whether the alternate register set is to be used in regular functions. For this to be effective, global optimization must also be used. With this in effect, the compiler will use the alternate register set for storing variables where appropriate. For more information, see -ALTREG: Use Alternate Register Set on page 100.

You can also select whether you want 8-bit I/O port addressing as opposed to the default 16-bit I? port addressing. See -P8: Use 8 bit port addressing on page 109 for more details.

CP/M-80 output file ...
You can select to output either a CP/M-80 .COM file or a library file.

Figure 3 - 8 Options Menu



```
 <o>  File   Edit   Options   Compile   Make   Run   Utility   Help          11:59:09 AM
 untitled
                   Memory model and chip type ...

                   CP/M-80 output file ...
                  ♦ROM output file ...
                   ROM & RAM addresses ...
                   ROM checksum specs ...

                   Long formats in printf()
                   Float formats in printf()
                   Map and symbol file options...

                  ♦Save dependency information

Line      1/1        Col 1        Insert    C mode  < Search F2 >
 Project: WOW.PRJ (Z80/Z180/64180 V7.60)
     HI-TECH C PROFESSIONAL DEVELOPMENT SYSTEM (Z80/Z180/64180 V7.60)
      Copyright (C) 1984-1996 HI-TECH Software, All Rights Reserved
             Serial no: CZ80-12345; Licensed to:
                          Clyde
```

ROM output file ...
The default output file type is Intel HEX. The other choices are: Motorola S-Record HEX, Binary Image, UBROF, Tektronix HEX, American Automation symbolic HEX, Intel OMF-51 and Bytecraft .COD. This option will also 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.

ROM & RAM addresses ...
This option allows you to enter the addresses of the ROM and RAM in your target system. For standard values refer to the instructions in the tutorial section on page 64.

The information entered into this dialog box is used by the linker to assign addresses to your code. Note that some of the RAM addresses may be set to zero to allow HPDZ to set them automatically. The addresses are used as follows:

ROM & RAM addresses

ROM and RAM addresses are applicable to both the small and the banked (large) memory models.

ROM address: This is the address in ROM where program code is to start. For virtually all Z80 systems this will be address 0, the lowest ROM address available. If compiling code which is to be downloaded into RAM using the LUCIFER debugger, this address will be the start of the downloadable RAM area.

The *rom* area starts with the reset vector, followed by any other interrupt vectors which have been initialized using the interrupt vector handling macros defined in *<intrpt.h>*.

RAM address: This is the starting address of RAM. For Z180 systems this should be the address within the first 64k that the RAM is to be mapped to (i.e. the RAM *logical* address).

RAM size: This is the size in bytes of RAM available, starting from the *ram* address.

NVRAM address: This is the address of a block of non-volatile RAM to be used for *persistent* variables. If all RAM is non-volatile or persistent variables are not used, this value should be zero. This will cause the non-volatile RAM area to be allocated from within the standard RAM area. The default value is zero.

For a Z80 system using small model, these four values (*ROM address*, *RAM address*, *RAM size* and *NVRAM address*) will be the only ones required.

Physical addresses

Physical addresses are only applicable for the banked (large) memory model.

RAM address: is the physical (20 bit) address at which the RAM is actually located in a Z180 system. To use the RAM it must be mapped down into the lower 64K of memory space. This value is required for large model code, and for Z180 code unless the RAM is already addressed in the lower 64K.

Banked area address: is the start of the ROM which is to be mapped into the banked area

Banked area logical addresses

Banked area logical addresses are only applicable for the banked (large) memory model.

Banked area address: is the logical starting address of the banked area, within the lower 64K address space. This and the next value, *banksize*, define a window in the bottom 64K which is mapped into ROM at various physical addresses.

Banked area size: is the size of the banked area which is mapped into ROM at various physical addresses.

In this dialog box, you can also choose whether initialised data is copied from ROM into RAM (default) or remains in ROM.

Long formats in printf()

This option is used to tell the linker that you wish to use the long printf() support library. The long library includes a version of printf() which supports the long output formats %1d, %1u and %1x. If you use this option the compiled application will increase in size. You should only use this option if you want to use long formats in printf(), it is not necessary if you merely want to perform long integer calculations. If you select this option a marker appears beside the line in the menu.

Float formats in printf()
This option 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. If you use this option the compiled application will be larger. It is not required to perform floating point calculations, so only use it if you wish to use floating point formats in printf().

Map and Symbol File Options ...
This dialog box allows you to set various options pertaining to debug information, the map file and the symbol file.

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, of an in-circuit emulator, you should enable this option.

Sort map by address
By default, the symbol table in the 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.

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.

Avocet format symbol file
Use this option to select generation of an avocet AVSIM compatible symbol file.

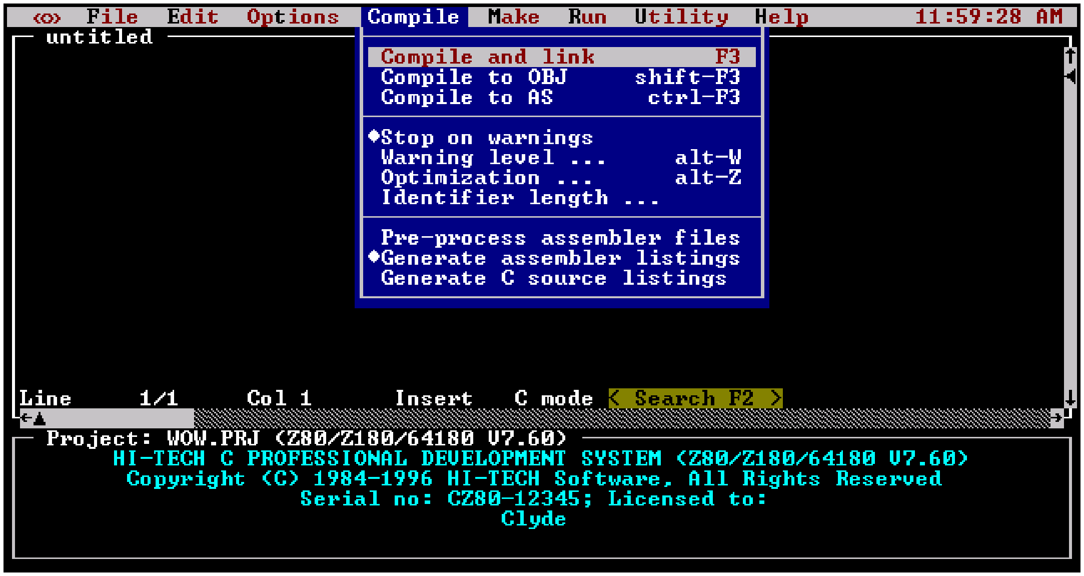
3.5.5 Compile menu

The **Compile** menu, shown in Figure 3 - 9 on page 84, 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 *objtohex.exe* 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 objtohex 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 the extension .OBJ.

Figure 3 - 9 HPDZ Compile Menu



```
 <○>  File  Edit  Options  Compile  Make  Run  Utility  Help        11:59:28 AM
 untitled
                          Compile and link          F3
                          Compile to OBJ      shift-F3
                          Compile to AS        ctrl-F3

                         ♦Stop on warnings
                          Warning level ...      alt-W
                          Optimization ...       alt-Z
                          Identifier length ...

                          Pre-process assembler files
                         ♦Generate assembler listings
                          Generate C source listings

 Line      1/1      Col 1        Insert    C mode < Search F2 >
 Project: WOW.PRJ (Z80/Z180/64180 V7.60)
     HI-TECH C PROFESSIONAL DEVELOPMENT SYSTEM (Z80/Z180/64180 V7.60)
      Copyright (C) 1984-1996 HI-TECH Software, All Rights Reserved
                Serial no: CZ80-12345; Licensed to:
                               Clyde
```

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 the 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 dialog which allows you set the compiler warning level, i.e. it determines how selective the compiler is about legal but dubious code. The range of currently implemented warning levels is -9 to 9, where 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 of the ZC command.

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

Identifier length...
By default C identifiers are considered significant only to 31 characters. This command will allow setting the number of significant characters to be used, between 31 and 255.

Pre-process assembler files
Selecting this item will make HPDZ pass assembler files through the pre-processor before assembling. This makes it possible to use C pre-processor macros and conditionals in assembler files. A mark appears before the item when it is selected.

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`.

Generate C source listings
Selecting this option will cause a C source listing for each C file compiled. The listing file will be named in the same way as an assembler file (described above) but will contain the C source code with line numbers, and with tabs expanded. The tab expansion setting is derived from the editor tab stop setting.

3.5.6 Make menu

The **Make** menu (Figure 3 - 10 on page 86) contains all of the commands required to use the HPDZ *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 **New project ...** command. After selecting the project file name, HPDZ will present several dialogs to allow you to set up the memory model.

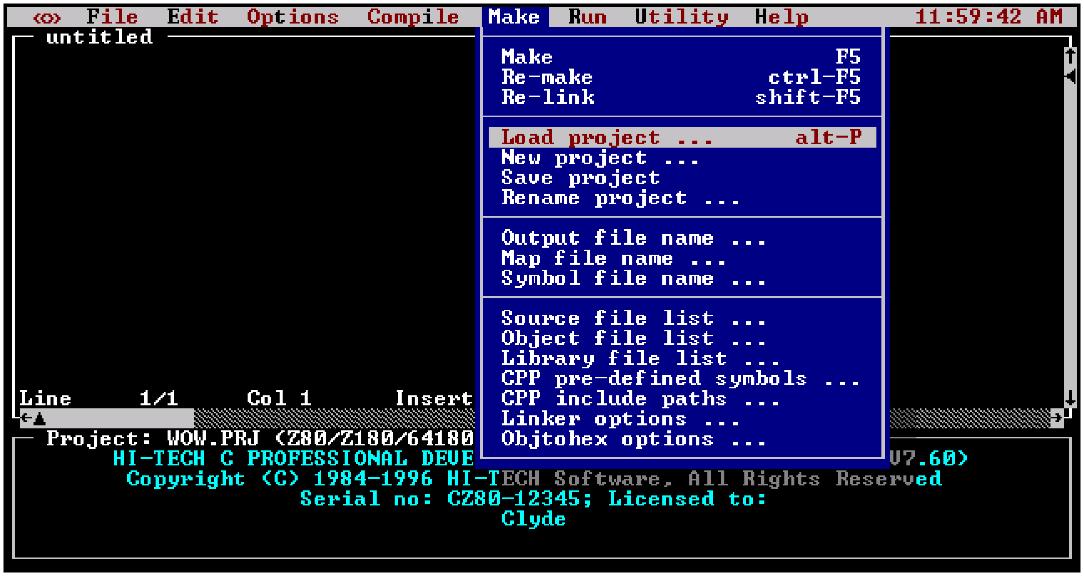
❒ Enter the list of source file names using the **Source file list ...** command.

❒ Set up any special libraries, pre-defined pre-processor symbols, object files or linker options using the other items in the **Make** menu.

❒ Save the project file using the **Save project** command.

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

Figure 3 - 10 HPDZ Make Menu



```
<◇>  File  Edit  Options  Compile  Make  Run  Utility  Help      11:59:42 AM
 untitled
                     Make                        F5
                     Re-make                ctrl-F5
                     Re-link               shift-F5

                     Load project ...         alt-P
                     New project ...
                     Save project
                     Rename project ...

                     Output file name ...
                     Map file name ...
                     Symbol file name ...

                     Source file list ...
                     Object file list ...
                     Library file list ...
                     CPP pre-defined symbols ...
Line       1/1      Col 1       Insert  CPP include paths ...
                     Linker options ...
 Project: WOW.PRJ (Z80/Z180/64180  Objtohex options ...
      HI-TECH C PROFESSIONAL DEVE                        V7.60)
       Copyright (C) 1984-1996 HI-TECH Software, All Rights Reserved
              Serial no: CZ80-12345; Licensed to:
                           Clyde
```

Make **F5**
The Make command re-compiles the current project. When Make is selected, HPDZ re-compiles any source files which have been modified since the last Make command was issued. HPDZ 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, HPDZ 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, HPDZ will print a message in the message window indicating that nothing was done.

HPDZ 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. In other words, if you modify a header file, any source files including that header file will be recompiled.

If you forget to use the source file list to select the files to be included, HPDZ will produce a dialog warning that no files have been selected. You will then have to select the **Done** button or press **escape**. This takes you back to the editor window.

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 ... **alt-P**
This command loads a pre-defined project file. You are presented with a file selection dialog allowing a `.PRJ` file to be selected and loaded. If this command is selected when the current project has been modified but not saved, you will be given a chance to save the project or abort the Load project command. After loading a project file, the message window title will be changed to display the project file name.

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 HPDZ will present several dialogs to allow you to configure the project. These dialogs will allow you to select: processor type, memory model and floating point type; output file type; ROM and RAM addresses; optimization settings; and map and symbol file options. 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 an .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 you to select 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.

Source file list ...
This option displays a dialog 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 the suffix `.C` and assembly language files the suffix `.AS`, so that HPDZ can determine where the files should be passed.

Object file list ...
This option allows any extra `.OBJ` files to be added to the project. Only enter one `.OBJ` file per line. Operation of this dialog is the same as the source file list dialog. This list will normally only contain one object file: the run-time start off module for the current code generation model. For example, if a project is generating small model code for a Z80, by default this list will contain the small model runtime startoff module `rtz80-s.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 for the Z80 and floating point printf in use, this list will contain the libraries `z80-sc.lib` and `z80-sf.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. Sometimes it is necessary to scan a user library more than once. In this case you should enter the name of the library more than once.

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 ZC. 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 files rely on their presence.

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

Linker options ...
This command allows the options passed to the linker by HPDZ 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 HPDZ to be modified. Normally you will not need to change these options as the generation of binary files and HEX files can be chosen in the Options menu. However, if you want to generate one of the unusual output formats which objtohex can produce, like COFF files, you will need to change the options using this command.

3.5.7 Run menu

The **Run** menu shown in Figure 3 - 11 on page 90 contains options allowing MS-DOS commands and user programs to be executed. It also contains the options to allow you to run code using the **LUCIFER** debugger.

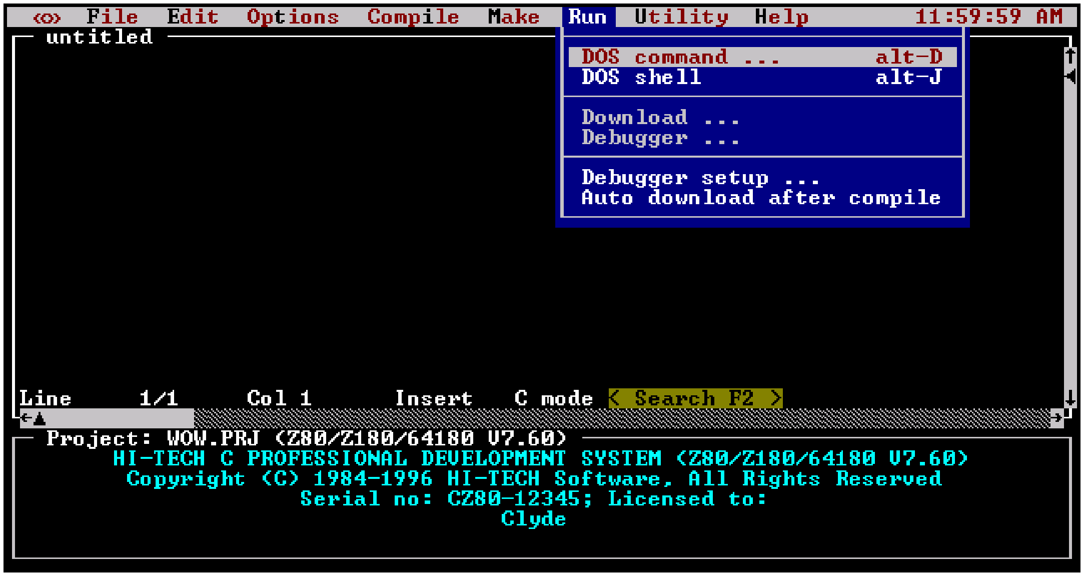
DOS command ... **alt-D**
This option allows a DOS command to be executed exactly like it had been entered at the COMMAND.COM prompt. This command could be an internal DOS command like DIR, or the name of a program to be executed. If you want to escape to the DOS command processor, use the DOS Shell command below.

Warning: do not use this option to load TSR programs.

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 HPDZ, type “exit” at the DOS prompt.

Download ...
This option runs the LUCIFER debugger, automatically downloads the current output file and loads the current symbol file. If the debugger has not been set up, this option will be unavailable. You should use the *Debugger setup* ... command instead. LUCIFER can only download Motorola HEX, Intel HEX and Binary type files. You should not attempt to download any other file types.

Figure 3 - 11 HPDZ Run Menu



<O> File Edit Options Compile Make Run Utility Help 11:59:59 AM
untitled
DOS command ... alt-D
DOS shell alt-J
Download ...
Debugger ...
Debugger setup ...
Auto download after compile
Line 1/1 Col 1 Insert C mode < Search F2 >

```
Project: WOW.PRJ (Z80/Z180/64180 V7.60)
     HI-TECH C PROFESSIONAL DEVELOPMENT SYSTEM (Z80/Z180/64180 V7.60)
      Copyright (C) 1984-1996 HI-TECH Software, All Rights Reserved
               Serial no: CZ80-12345; Licensed to:
                              Clyde
```

Debugger ...
This option runs the LUCIFER debugger with the current symbol file. This option does not download user code, so can be used to return to a suspended LUCIFER session.

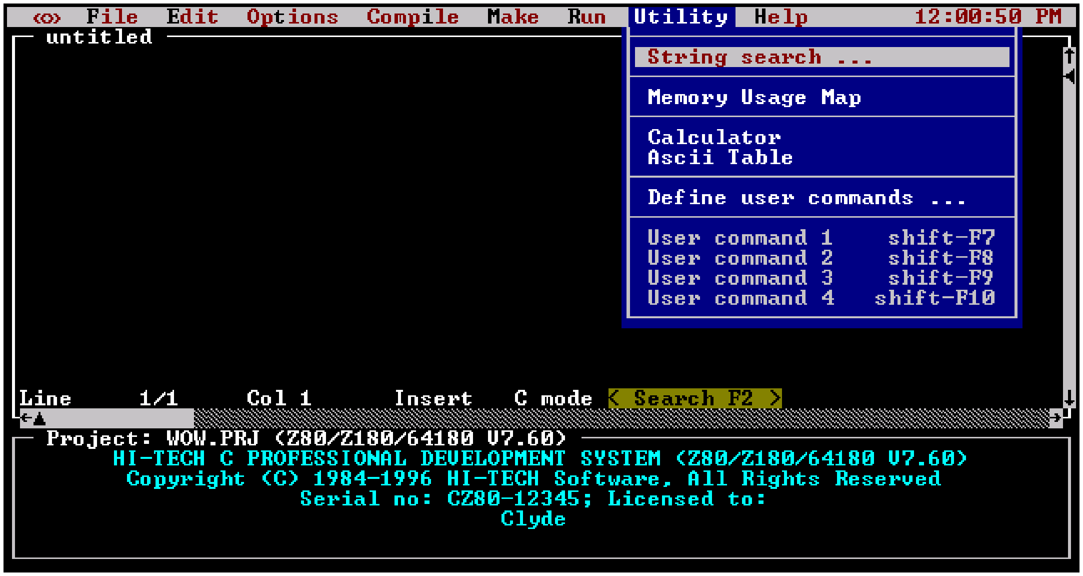
Debugger setup ...
This option activates a dialog box which allows you to select the serial port parameters for the LUCIFER debugger. Once you have set up the parameters they are saved in the project file with the other settings. The default settings for the Z80 version of LUCIFER are saved in the `LUCZ80_ARGS` environment variable. For more information refer to Lucifer Source Level Debugger on page 197.

Auto download after compile
This option allows you to enable or disable automatic downloading of code after compiling. When it is enabled and the LUCIFER debugger set up, at successful compilation HPDZ automatically invokes LUCIFER to download the current output file and load the current symbol file. When it is enabled a mark appears before the item

3.5.8 Utility menu

The **Utility** menu (Figure 3 - 12 on page 91) contains any useful utilities which have been included in HPDZ.

Figure 3 - 12 HPDZ Utility Menu



```
Project: WOW.PRJ (Z80/Z180/64180 V7.60)
HI-TECH C PROFESSIONAL DEVELOPMENT SYSTEM (Z80/Z180/64180 V7.60)
Copyright (C) 1984-1996 HI-TECH Software, All Rights Reserved
Serial no: CZ80-12345; Licensed to:
Clyde
```

String search ...
This option allows you to conduct a string search in a list of files. The option produces a dialog which enables you to type in the string you are seeking and then select a list of files to search. You can also select case sensitivity. It is possible to limit the search to a source file list or just the current project.

Memory usage map
This option displays a window which contains a detailed memory usage map of the last program which was compiled.

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 or select display of the table in octal, decimal or hexadecimal.

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...

Table 3 - 9 Macros usable in user commands

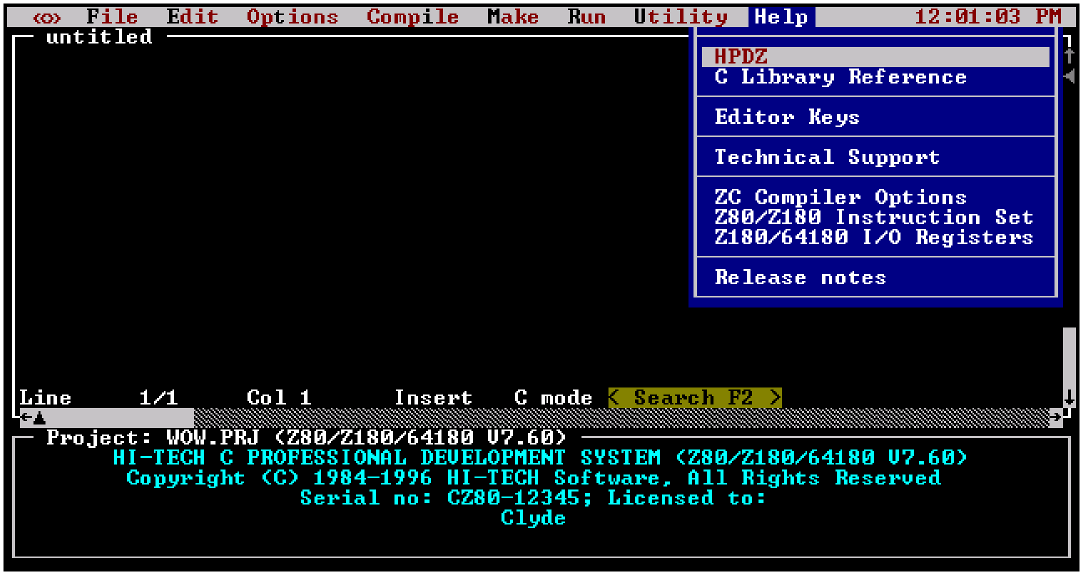
Macro name	Meaning
$(LIB)	Expands to the name of the system library file directory; eg C:\HPDZ\LIB\
$(CWD)	The current working directory
$(INC)	The name of the system include directory
$(EDIT)	The name of the file currently loaded into the editor. If the current file has been modified, this will be replaced 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, eg if the current project file is AUDIO.PRJ, this macro will expand to AUDIO with no dot or file type.

In the Utility menu are four user-definable commands. This item will invoke a dialog 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 - 9 on page 92. 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 HPDZ.

3.5.9 Help menu

The **Help** menu (Figure 3 - 13 on page 93) contains items allowing you to obtain help about any topics listed.

Figure 3 - 13 HPDZ Help Menu



```
Project: WOW.PRJ (Z80/Z180/64180 V7.60)
HI-TECH C PROFESSIONAL DEVELOPMENT SYSTEM (Z80/Z180/64180 V7.60)
Copyright (C) 1984-1996 HI-TECH Software, All Rights Reserved
Serial no: CZ80-12345; Licensed to:
Clyde
```

On startup, HPDZ 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 HT_xx_HLP. If this is not set, it will be derived from the full path name used when HPDZ was invoked. If the help directory cannot be located, none of the standard help entries will be available.

HPDZ
This option produces a window showing all the topics for which help is available. These include Checksum specifications, compiler optimizations, editor searching, memory models and chip types, ROM and RAM addresses and string search.

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 in a separate window. You can move around within the reference using the keypad cursor keys and the index can be re-entered using the INDEX button at the bottom of the window.

If you have a mouse, you can follow *hypertext* links by double clicking the mouse on any word. For example, if you are in the **printf** entry and double click on the reference to **fprintf**, you will be taken to the entry for **fprintf**.

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

Editor Keys
This option displays a list editor commands and the corresponding keys used to activate that command.

Technical Support
This option displays a list of dealers and their phone numbers for you to use should you require technical support.

ZC Compiler Options
This option displays a window showing all the ZC compiler options. They are displayed in a table showing the option and its meaning. You can scroll through the table using the normal scroll keys or the mouse.

Z80/Z180 Instruction Set
This option displays a table of the entire instruction set for the Z80/Z180 processors.

Z180/64180 I/O Registers
This option displays a table of the I/O registers for the Z180 and 64180 processors.

Release notes
This option displays the release notes for your program. You can scroll through the window using the normal scrolling keys or the mouse.

ZC Command Line Compiler Driver

ZC is invoked from the DOS command line to compile and/or link C programs. If you prefer to use an integrated environment then see the HPDZ chapter. ZC has the following basic command format:

ZC [options] files [libraries]

It is conventional to supply the options (identified by a leading dash '-') before the filenames, but in fact this is not essential. The options are discussed below. The files may be a mixture of source files (C or assembler) and object files. The order of the files is not important, except that it will affect the order in which code or data appears in memory. The libraries are a list of library names, or *-L* options (see page 106). Source files, object files and library files are distinguished by ZC solely by the file type or extension. Recognized file types are listed in Table 4 - 1 on page 95. This means, for example, that an assembler file must always have a file type of *.AS* (alphabetic case is not important).

Table 4 - 1 ZC file types

File Type	Meaning
.C	C source file
.AS	Assembler source file
.OBJ	Object code file
.LIB	Object library file

ZC will check each file argument and perform appropriate actions. C files will be compiled; assembler files will be assembled. At the end, unless suppressed by one of the options discussed later, all object files resulting from a compilation or assembly, or listed explicitly, will be linked with any specified libraries. Functions in libraries will be linked only if referenced.

Invoking ZC with only object files as arguments (i.e. no source files) will mean only the link stage is performed. It is typical in Makefiles to use ZC with a -C option to compile several source files to object files, then to create the final program, invoke ZC with only object files and libraries (and appropriate options).

4.0.1 Long command lines

Since DOS has a command line limitation of 128 characters, to invoke ZC with a long list of options and files you may create a command file containing the ZC command line, and invoke ZC with its input redirected from that file. With no command line options specified, ZC will read its standard input to get the argument list. For example a command file may contain:

```
-v -O -Otest.hex -A0,8000,8000 -bl
file1.obj file2.obj mylib.lib
```

If this was in the file *xyz.cmd* then ZC would be invoked as:

```
ZC < xyz.cmd
```

Since no command line arguments were supplied, ZC will read *xyz.cmd* for its command line.

4.0.2 Default Libraries

ZC will search the standard C library by default. This will always be done last, after any user-specified libraries. The particular library will be dependent on the memory model. The standard library contains a version of printf that supports only *int* length values. If you want to print long values with printf, or sprintf or related functions, you must specify a *-LL* option. This will search the "long" library. For floating point printf support, use the *-LF* option.

4.0.3 Standard Run-Time Startoff

ZC will also automatically provide the standard run-time startoff module appropriate for the memory model. If you require any special powerup initialization, rather than replace or modify the standard run-time startoff module, you should use the *powerup* routine feature (see page 145).

4.1 ZC Compiler Options

The compiler is configured primarily for generation of ROM code, but will optionally create executables for the CP/M operating system (via the -CPM option). ZC recognizes the compiler options listed in Table 4 - 2 on page 97. The ZC command also allows access to a number of advanced compiler features which are not available within the HPDZ integrated development environment.

4.1.1 -180: Generate Z180/64180 Code

This option selects code generation for the Z180/64180 processors, and specifies the appropriate libraries for that processor.

4.1.2 -64180: Generate Z180/64180 Code

This option is the same as the -180 option.

4.1.3 -A*spec*: Set ROM and RAM Addresses

The `-A` option is used to set the ROM and RAM addresses which will be used to link your code to absolute addresses. This option takes the following form for Z80 small model:

-Arom,ram,ramsize,nvram

and this form for other models and processors:

Table 4 - 2 ZC options

Option	Meaning
-180	Generate code for the Z180 processor
-64180	Generate code for the 64180 processor
-A*spec*	Specify memory addresses for linking
-AAHEX	Generate an American Automation symbolic HEX file
-ALTREG	Use alternate register set
-ASMLIST	Generate assembler .LST file for each compilation
-AV	Select AVOCET format symbol table
-AVSIM	Same as -AV
-BIN	Generate a Binary output file
-Bs	Select *small* memory model
-Bl	Select *large* memory model
-Bc	Select *CP/M* memory model
-C	Compile to object files only
-CLIST	Generate C source listing file
-CPM	Generate CP/M executable file
-CR*file*	Generate cross-reference listing
-D*macro*	Define pre-processor macro
-E	Use “editor” format for compiler errors
-E*file*	Redirect compiler errors to a file
-E+*file*	Append errors to a file
-G*file*	Generate source level symbol table
-H*file*	Generate symbol table without line numbers etc.
-HELP	Print summary of options
-I*path*	Specify a directory pathname for include files
-L*library*	Specify a library to be scanned by the linker
-L-*option*	Specify -*option* to be passed directly to the linker
-M*file*	Request generation of a MAP file
-MOTOROLA	Generate a Motorola S1/S9 HEX format output file
-N*length*	Set identifier length to *length* (default is 31 characters)
-O	Enable peephole optimization
-O*file*	Specify output filename
-OF	Optimize for speed
-OMF51	Produce an OMF-51 output file

Table 4 - 2 ZC options

Option	Meaning
-P	Preprocess assembler files
-P8	Use 8 bit port addressing
-P16	Use 16 bit port addressing
-PROTO	Generate function prototype information
-PSECTMAP	Display complete memory segment usage after linking
-q	Specify quiet mode
-ROMDATA	Leave initialised data in ROM
-ROM*ranges*	Specify ROM ranges for code
-S	Compile to assembler source files only
-SA	Compile to Avocet AVMAC assembler source files
-STRICT	Enable strict ANSI keyword conformance
-TEK	Generate a Tektronix HEX format output file
-UBROF	Generate an UBROF format output file
-UNSIGNED	Make default character type *unsigned*
-U*symbol*	Undefine a predefined pre-processor symbol
-V	Verbose: display compiler pass command lines
-W*level*	Set compiler warning level
-X	Eliminate local symbols from symbol table
-Z180	Generate code for the Z180 processor
-Zg	Enable global optimization in the code generator

-Arom,ram,ramsize,ramphys,banklogi,banksize,bankphys,nvram

where:

rom is the address in ROM where program code is to start. For virtually all Z80 systems this will be address 0, the lowest ROM address available. If compiling code which is to be downloaded into RAM using the LUCIFER debugger, this address will be the start of the downloadable RAM area. The *rom* area starts with the reset vector, followed by any other interrupt vectors which have been initialized using the interrupt vector handling macros defined in <*intrpt.h*>.

ram is the starting address of RAM. For Z180 systems this should be the address within the first 64k that the RAM is to be mapped to (i.e. the RAM *logical* address).

ramsize is the size in bytes of RAM available, starting from the *ram* address.

ramphys is the physical (20 bit) address at which the RAM is actually located in a Z180 system. To use

	the RAM it must be mapped down into the lower 64K of memory space. This value is required for large model code, and for Z180 code unless the RAM is already addressed in the lower 64K.
banklogi	is the logical starting address of the banked area, within the lower 64K address space. This and the next value, *banksize*, define a window in the bottom 64K which is mapped into ROM at various physical addresses.
banksize	is the size of the banked area which is mapped into ROM at various physical addresses.
bankphys	is the start of the ROM which is to be mapped into the banked area
nvram	is the address of a block of non-volatile RAM to be used for *persistent* variables. If all RAM is non-volatile or persistent variables are not used, this value should be zero. This will cause the non-volatile RAM area to be allocated from within the standard RAM area. The default value is zero.

For a Z80 system using small model, only four values (*rom, ram, ramsize* and *nvram*) will be required.

Note that for the Z180 all the logical and physical addresses must be multiples of 1000 hex (4096 decimal) because the Z180 memory management is performed in 4096 byte pages.

All values taken by the `-A` option are hexadecimal (base 16) numbers, and should be specified without a trailing “H”. Thus, the option

```
-A0,2000,2000
```

is correct, but the option:

```
-A0H,2000H,2000H
```

is not. Some examples of valid `-A` options follow:

A Z80 system with 32k ROM at 0H; and 32k RAM starting at 8000h would require:

```
-A0,8000,8000
```

A Z180 system with 128K of ROM starting at zero, and 32K of RAM starting at 40000 (256K) might require the following option:

```
-A0,4000,8000,40000,1000,2000,4000
```

In this example the banked area runs from 1000 to 2FFF, and is mapped into physical memory starting at 4000. Since there will be 1000 hex bytes of ROM still mapped at zero, this will leave 3000 hex bytes of ROM unused between this and the base of the banked ROM.

4.1.4 -AAHEX: Generate American Automation Symbolic Hex

The `-AAHEX` option directs ZC to generate an American Automation symbolic format HEX file, producing a file with the `.HEX` extension. This option has no effect if used with a `.BIN` file. The American Automation hex format is an enhanced Motorola S-Record format which includes symbol records at the start of the file. This option should be used if producing code which is to be debugged with an American Automation in-circuit emulator.

4.1.5 -ALTREG: Use Alternate Register Set

This option allows the alternate register set to be used in regular functions. Note that use of *fast interrupt* precludes the use of the alternate register set in any other way, but use in ordinary functions is compatible with the existing use of the alternate register set in library functions.

If a conflict occurs, there will be a link-time error stating that the symbol “Fast_interrupt_cant_be_used_with_long_or_float_or_altreg” is multiply defined. This indicates that *fast interrupt* functions have been declared, but the alternate register set is also being used for non-interrupt code.

For this to be effective, global optimization must also be used (via the `-Zg` option). With this in effect, the compiler will use the alternate register set for storing variables where appropriate.

4.1.6 -ASMLIST: Generate Assembler .LST Files

The `-ASMLIST` option tells ZC to generate an assembler `.LST` file for each compilation. The list file shows both the original C code, and the generated assembler code and the corresponding binary code. The listing file will have the same name as the source file, and a file type (extension) of `.LST`.

This option is not compatible with the -CLIST option.

4.1.7 -AV: Select Avocet Symbol File

The `-AV` option is used in conjunction with the `-H` option to generate Avocet style symbol tables for use with AVSIM simulators and certain in-circuit emulators. This option only sets the symbol table format, it does not tell the compiler to actually generate a symbol file. In order to generate an Avocet symbol file you should use the `-AV` option with the `-H` option. For example:

```
ZC -AV -Htest.sym -A8000,8000 test.c
```

will generate an Avocet style symbol table called `test.sym`. The `-AV` option should not be used with the ZC `-G` option as Avocet symbol tables make no provision for source level debug information.

4.1.8 -AVSIM: Select Avocet Symbol File

This is identical to the `-AV` option.

4.1.9 -BIN: Generate Binary Output File

The `-BIN` option tells ZC to generate a Binary image output file. The output file will be given type `.BIN`. Binary output may also be selected by specifying an output file of type `.BIN` using the `-O`*file* option.

4.1.10 -Bs: Select Small Memory Model

The `-Bs` option is used to select code generation using the *small memory model*. The small model uses a 64k address space. The libraries used with small model are listed in Table 4 - 3 on page 101.

Table 4 - 3 Small model libraries

Library	Purpose
RTZ80-S.OBJ	Small model run-time startoff
Z80-SC.LIB	Small model standard C library
Z80-SL.LIB	Small model printf library, long support
Z80-SF.LIB	Small model printf library, long and float support
RTZ801S.OBJ	Z180 Small model run-time startoff
Z801SC.LIB	Z180 Small model standard C library
Z801SL.LIB	Z180 Small model printf library, long support
Z801SF.LIB	Z180 Small model printf library, long and float support

4.1.11 -Bl: Select Large Memory Model

The `-Bl` option is used to select code generation using the *large memory model*, which uses bank switching. The runtime startoff module and libraries associated with large model are listed in Table 4 - 4 on page 101.

Table 4 - 4 Large model libraries

Library	Purpose
RTZ80-L.OBJ	Large model run-time startoff
Z80-LC.LIB	Large model standard C library
Z80-LL.LIB	Large model printf library, long support
Z80-LF.LIB	Large model printf library, long and float support
RTZ801L.OBJ	Z180 Large model run-time startoff
Z801LC.LIB	Z180 Large model standard C library
Z801LL.LIB	Z180 Large model printf library, long support
Z801LF.LIB	Z180 Large model printf library, long and float support

If large model is used with a Z180 processor, bank switching is handled by the Z180 MMU. If large model is used with a processor other than the Z180, you have to write your own bank switching routines. See Function Calling Conventions for Large Model on page 141 for more details.

4.1.12 -Bc: Select CP/M Memory Model

The `-Bc` option is used to select code generation using the *CP/M memory model*. The CP/M model allows a 64k address space and uses the CP/M libraries. The runtime startoff module and libraries used with medium model are listed inTable 4 - 5 on page 102. Note that the `-CPM` option is equivalent to the `-Bc` option.

Table 4 - 5 CP/M model libraries

Library	Purpose
RTZ80-C.OBJ	CP/M model run-time startoff
Z80-CC.LIB	CP/M model standard C library
Z80-CL.LIB	CP/M model printf library, long support
Z80-CF.LIB	CP/M model printf library, long and float support
RTZ801C.OBJ	Z180 CP/M model run-time startoff
Z801CC.LIB	Z180 CP/M model standard C library
Z801CL.LIB	Z180 CP/M model printf library, long support
Z801CF.LIB	Z180 CP/M model printf library, long and float support

4.1.13 -C: Compile to Object File

The `-C` option is used to halt compilation after generating object files. This option is frequently used when compiling multiple source files using a “make” utility. If multiple source files are specified to the compiler each will be compiled to a separate `.OBJ` file. To compile three source files `main.c`, `module1.c` and `asmcode.as` to object files you could use the command:

```
ZC -O -Zg -C main.c module1.c asmcode.as
```

The compiler will produce three object files `main.obj`, `module1.obj` and `asmcode.obj` which could then be linked to produce a Motorola HEX file using the command:

```
ZC -A0,8000,8000 main.obj module1.obj asmcode.obj
```

The compiler will accept any combination of `.C`, `.AS` and `.OBJ` files on the command line. Assembler source files will be passed directly to the assembler and object files will not be used until the linker is invoked. Unless the `-O`*file* option is used to specify an output file name and type the final output will be a Motorola hex file with the same “base name” as the first source or object file, the example above would produce a file called `main.hex`.

4.1.14 -CLIST: Produce C Listing File

This option will generate a listing file for each C source file, containing line numbers with tabs formatted to spaces on 8 character stops. The listing file will be called *file*.LST where *file* is the base name of the C source file.

This option is not compatible with the -ASMLIST option.

4.1.15 -CPM: Generate CP/M Executable File

The `-CPM` option is equivalent to the `-Bc` option, and selects CP/M code and output file format.

4.1.16 -CR*file*: Generate Cross Reference Listing

The `-CR` option will produce a cross reference listing. If the *file* argument is omitted, the "raw" cross reference information will be left in a temporary file, leaving the user to run the CREF utility. If a filename is supplied, for example `-CRtest.crf`, ZC will invoke CREF to process the cross reference information into the listing file, in this case `TEST.CRF`. If multiple source files are to be included in the cross reference listing, all must be compiled and linked with the one ZC command. For example, to generate a cross reference listing which includes the source modules `main.c`, `module1.c` and `nvram.c`, compile and link using the command:

```
      ZC -CRmain.crf main.c module1.c nvram.c
```

4.1.17 -D*macro*: Define Macro

The `-D` option is used to define a pre-processor macro on the command line, exactly as if it had been defined using a `#define` directive in the source code. This option may take one of two forms, **-D*macro*** which is equivalent to:

```
      #define  macro  1
```

or **-D*macro*=*text*** which is equivalent to:

```
      #define  macro  text
```

Thus, the command:

```
      ZC -Ddebug -Dbuffers=10 test.c
```

will compile `test.c` with macros defined exactly as if the C source code had included the directives:

```
      #define  debug   1
      #define  buffers 10
```

4.1.18 -E: Use “editor” Format for Compiler Errors

The `-E` option instructs the compiler to generate error messages in a format which is acceptable to some text editors. The default behaviour, if `-E` is not used, is to display compiler errors in a “human readable” format line with a caret and error message pointing out the offending characters in the source line, for example:

```
x.c: main()
     4: PORT_A = xFF;
                ^ undefined identifier: xFF
```

The standard format is perfectly acceptable to a person reading the error output but is not usable with editors which support compiler error handling. If the same source code were compiled using the `-E` option, the error output would be:

```
x.c 4 9: undefined identifier: xFF
```

indicating that the error occurred in file `x.c` at line 4, offset 9 characters into the statement. The second numeric value, the column number, is relative to the left-most non-space character on the source line. If an extra space or tab character were inserted at the start of the source line, the compiler would still report an error at line 4, column 9. Error output, either in standard or `-E` format, can be redirected into files using UNIX or DOS style standard output redirection. The error from the example above could have been redirected into a file called `errlist` using the command:

```
ZC -E x.c > errlist
```

Compiler errors can also be appended onto existing files using the redirect and append syntax. If the error file specified does not exist it will be created. To append compiler errors onto a file use a command like:

```
ZC -E x.c >> errlist
```

4.1.19 -E*file*: Redirect Compiler Errors to a File

Some editors do not allow the standard command line redirection facilities to be used when invoking the compiler. To work with these editors, ZC allows the error listing file name to be specified as part of the `-E` option. Error files generated using this option will always be in `-E` format. For example, to compile `x.c` and redirect all errors to `x.err`, use the command:

```
ZC -Ex.err x.c
```

The `-E` option also allows errors to be appended to an existing file by specifying a `+` at the start of the error file name, for example:

```
ZC -E+x.err y.c
```

If you wish to compile several files and combine all of the errors generated into a single text file, use the `-E` option to create the file then use `-E+` when compiling all the other source files. For example, to compile a number of files with all errors combined into a file called `project.err`, you could use the `-E` option as follows:

```
ZC -Eproject.err -O -Zg -C main.c

ZC -E+project.err -O -Zg -C part1.c

ZC -E+project.err -C asmcode.as
```

The file `project.err` will contain any errors from `main.c`, followed by the errors from `part1.c` and then `asmcode.as`, for example:

```
main.c 11 22: ) expected

main.c 63 0: ; expected

part1.c 5 0: type redeclared

part1.c 5 0: argument list conflicts with prototype

asmcode.as 14 0: Syntax error

asmcode.as 355 0: Undefined symbol _putint
```

4.1.20 -G*file*: Generate Source Level Symbol File

`-G` generates a source level symbol file for use with HI-TECH Software debuggers and simulators such as *Lucifer*. If no filename is given, the symbol file will have the same "base name" as the first source or object file, and an extension of `.SYM`. For example, `-GTEST.SYM` generates a symbol file called `TEST.SYM`. Symbol files generated using the `-G` option include source level information for use with source level debuggers.

This option should not be used in conjunction with Avocet style symbol tables (ZC option `-AV`) as the Avocet symbol table format makes no provision for source level debug information. Note that all source files for which source level debugging is required should be compiled with the `-G` option, for example:

```
ZC -G -C test.c

ZC -C module1.c

ZC -A0,30,2000 -Gtest.sym test.obj module1.obj
```

will include source level debugging information for `test.c` only because `module1.c` was not compiled with the `-G` option.

4.1.21 -H*file*: Generate Assembler Level Symbol File

The `-H` option generates a symbol file without any source level information. HI-TECH Software debuggers will only be able to perform assembler level debugging when using symbol files generated using `-H`. Normally the symbol file generated using this option will be a HI-TECH Software format symbol table, however if this option is used in conjunction with the ZC option `-AV`, an Avocet style symbol table will be generated. Avocet symbol tables are used by certain in-circuit emulators.

4.1.22 -HELP: Display Help

The `-HELP` option displays information on the ZC options.

4.1.23 -I*path*: Include Search Path

Use `-I` to specify an additional directory to use when searching for header files which have been included using the `#include` directive. The `-I` option can be used more than once if multiple directories are to be searched. The default include directory containing all standard header files will still be searched, after any user specified directories have been searched. For example:

```
      ZC -C -Ic:\include -Id:\myapp\include test.c
```

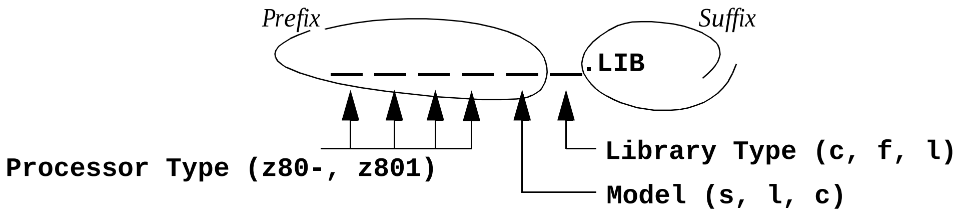
will search the directories `c:\include` and `d:\myapp\include` for any header files included using angle brackets.

4.1.24 -L*library*: Scan Library

The `-L` option is used to specify additional libraries which are to be scanned by the linker. Libraries specified using the `-L` option are scanned before the standard C library, allowing alternate versions of standard library functions to be accessed. For example, if using the Z80 small memory model, the floating point version of *printf()* can be linked in preference to the standard version by searching the library *z80-sf.lib* using the option `-Lf`. The argument to `-L` is a library keyword to which the standard library prefix and suffix `.LIB` are added. The library prefix depends on which processor and memory model are being used. The standard library prefixes are shown in Figure 4 - 1 on page 107, where: *Processor Type* is `'z80-'` for the Z80 or `'z801'` for the Z180; *Model* is `'s'` for small, `'l'` for large and `'c'` for CP/M; and the *Library Type* is `'c'` for the standard library, `'l'` for printf, supporting longs, and `'f'` for printf supporting floats and longs. (The *Library Type* is what can be specified using this option.)

All libraries must be located in the *LIB* subdirectory of the compiler installation directory. Libraries in other directories can only be accessed using HPDZ or by invoking the linker directly. The complete set of libraries and runtime startoff modules supplied with the compiler is listed in Table 4 - 3 to Table 4 - 5.

Figure 4 - 1 Library Prefixes and Suffixes



4.1.25 -L-*option*: Specify Extra Linker Option

The `-L` option can also be used to specify an extra "-" option which will be passed directly to the linker by ZC. If `-L` is followed immediately by any text starting with a "-" character, the text will be passed directly to the linker without being interpreted by ZC. For example, if the option `-L-FOO` is specified, the `-FOO` option will be passed on to the linker when it is invoked. The `-L` option is especially useful when linking code which contains extra program sections (or *psects*, as may be the case if the program contains assembler code or C code which makes use of the `#pragma psect` directive. If the `-L` option did not exist, it would be necessary to invoke the linker manually or use an HPDZ option to link code which uses extra psects. The `-L` option makes it possible to specify any extra psects simply by using an extra linker `-P` option. To give a practical example, suppose your code contains variables which have been mapped into a special RAM area using an extra psect called *xram*. In order to link this new psect at the appropriate address all you need to do is pass an extra linker `-P` option using the `-L` option. For example, if the special RAM area (*xram* psect) were to reside at address 2000h, you could use the ZC option `-L-Pxram=2000h` as follows:

```
ZC -Bl -L-Pxram=2000h -A0,4000,4000 prog.c xram.c
```

One commonly used linker option is *-N*, which sorts the symbol table in the map file in address rather than name order. This is passed to ZC as *-L-N*.

4.1.26 -M*file*: Generate Map File

The `-M` option is used to request the generation of a map file. If no filename is specified, the map information is displayed on the screen, otherwise the filename specified to `-M` will be used.

4.1.27 -MOTOROLA: Generate Motorola S-Record HEX File

The `-MOTOROLA` option directs ZC to generate a Motorola S-Record HEX file if producing a file with `.HEX` extension. This option has no effect if used with a `.BIN` file.

4.1.28 -N*length*: Specify Identifier Significant Length

By default identifiers are truncated at 31 characters, so that two identifiers that were the same in their first 31 characters would be considered identical. This is consistent with the 31 character minimum specified by the ANSI/ISO standard for C. Some applications may require identifiers to be distinguished by more than 31 characters. This option sets the significant number of characters to *length*. It may not be set outside the bounds 31-255. Use this option cautiously as use of identifiers longer than 31 characters will mean the code may not compile with other ANSI-conformant compilers.

4.1.29 -O: Invoke Optimizer

`-O` invokes the peephole optimizer after the code generation pass. Peephole optimization reduces the code size by removing redundant jump and register load instructions.

4.1.30 -O*file*: Specify Output File

This option allows the name and type of the output file to be specified to the compiler. If no `-O` option is specified, the output file will be named after the first source or object file. You can use the `-O` option to specify an output file of type HEX, BIN or UBR, containing HEX, Binary or UBROF respectively. For example:

```
      ZC -Otest.bin -A0,4000,4000 prog1.c part2.c
```

will produce a binary file named TEST.BIN.

This option will have no effect on *.obj* or *.as* files produced by the compiler.

4.1.31 -OF: Optimize for Speed

This option optimizes the code generated by the compiler for speed.

4.1.32 -OMF51: Produce OMF-51 Output File

This option will make the compiler generate an output (executable code) file in Intel OMF-51 (strictly speaking, AOMF-51) format. This format is used by some in-circuit emulators. It supports only a 64K address space each for code and data.

This option is unlikely to be of any use for the Z80 processor family.

4.1.33 -P: Pre-process Assembly Files

`-P` causes the assembler files to pre-processed before they are assembled. This allows assembler files to use #include, #if etc.

4.1.34 -P8: Use 8 bit port addressing

Z80 systems that use 8 bit port addressing can benefit from use of the 8 bit direct addressed *IN* and *OUT* instructions. This option enables the use of these instructions. The default is 16 bit port addressing. For the Z180 this option is neither necessary nor desirable. The Z180 option automatically causes the compiler to use *IN0* and *OUT0* instructions.

4.1.35 -P16: Use 16 bit port addressing

This is the default. See the *-P8* option for more information.

4.1.36 -PRE: Produce Pre-processed Source Code

`-PRE` is used to generate pre-processed C source files with an extension .PRE. Use this option if sending source code for technical support.

4.1.37 -PROTO: Generate Prototypes

`-PROTO` is used to generate `.PRO` files containing both ANSI and K&R style function declarations for all functions within the specified source files. Each .PRO file produced will have the same base name as the corresponding source file. Prototype files contain both ANSI C style prototypes and old style C function declarations within conditional compilation blocks. The *extern* declarations from each .PRO file should be edited into a global header file which is included in all the source files comprising a project. .PRO files may also contain *static* declarations for functions which are local to a source file. These *static* declarations should be edited into the start of the source file. To demonstrate the operation of the `-PROTO` option, enter the following source code as file `test.c`:

```
#include        <stdio.h>
int
add(arg1, arg2)
int *   arg1;
int *   arg2;
{
        return *arg1 + *arg2;
}

void
printlist(list, count)
int *   list;
int     count;
{
        while (count--)
```

```
                printf("%d ", *list++);
        putchar('\n');
}
```

If compiled with the command `ZC -PROTO test.c`, ZC will produce `test.pro` containing the following declarations which may then be edited as necessary:

```
/* Prototypes from test.c */
/* extern functions - include these in a header file */
#if     PROTOTYPES
 extern int add(int *, int *);
 extern void printlist(int *, int);
#else            /* PROTOTYPES */
 extern int add();
 extern void printlist();
#endif           /* PROTOTYPES */
```

4.1.38 -PSECTMAP: Display Complete Memory Usage

The `-PSECTMAP` option is used to display a complete memory and psect (*program section*) dump after linking the user code. The information provided by this option is more detailed than the standard memory usage map which is normally printed after linking. The `-PSECTMAP` option causes the compiler to print a listing of every compiler and user generated psect, followed by the standard memory usage map. For example:

```
Psect Usage Map:

Psect   | Contents                  | Memory Range
--------|---------------------------|-------------------------
lowtext | User defined psect        |  0071H -  00F3H
vectors | Interrupt vectors         |  0000H -  0070H
text    | Program and library code  |  00F4H -  0D25H
strings | Unnamed string constants  |  0D26H -  0D89H
const   | 'const' class data        |  0D8AH -  0EADH
bss     | Uninitialized RAM vars    |  4000H -  4025H

Memory Usage Map:

User:   0071H -  00F3H   0083H (131) bytes
```

```
      CODE:   0000H -  0070H   0071H (113) bytes
      CODE:   00F4H -  0EADH   0DBAH (3514) bytes
      RAM:    4000H -  4025H   0026H (38) bytes
```

4.1.39 -q: Quiet Mode

If used, this option must be the *first* option. It places the compiler in quite mode which suppresses the HI-TECH Software copyright notice from being output.

4.1.40 -ROMDATA

By default the program is linked so that initialized data (i.e. any static or global variables or arrays that are statically initialized but not qualified `const`) will be copied from ROM to RAM at start-up. This allows the data to be modified at run-time in accordance with standard C practice.

It is often preferred, however, for initialized data to remain in ROM, thus saving RAM space. the `-ROMDATA` option selects this alternative. Initialized data is then not modifiable at run-time. Constant strings and `const` data are not affected by this option, and will always remain in ROM.

4.1.41 -ROM*ranges*

Program code can be allocated into specific areas of the code memory space - the interrupt vectors and some other code must be allocated at a fixed address, and the ROM address specified in the -A option is always used for this. If a *-ROM* option is used, then rather than all other code being allocated upwards from above the vectors, it will be allocated into the specified ranges. The syntax is identical to the *-RAM* option, e.g.

`-ROM0-2FFF,4000-5FFF`

Note that it is not necessary to exclude areas used by the vectors from the ROM ranges, as the linker will do so automatically.

4.1.42 -S: Compile to Assembler Code

The `-S` option stops compilation after generating an assembler source file. An assembler file will be generated for each C source file passed on the command line. The command:

```
      ZC -O -Zg -S test.c
```

will produce an assembler source file called `test.as` which contains the code generated from `test.c`. The optimization options `-O` and `-Zg` can be used with `-S`, making it possible to examine the compiler output for any given set of options. This option is particularly useful for checking function calling conventions and “signature” values when attempting to write external assembly language routines.

4.1.43 -SA: Compile to Avocet assembler source files

This option directs the compiler to compile the source to Avocet AVMAC assembler source files.

4.1.44 -STRICT: Strict ANSI Conformance

The `-STRICT` option is used to enable strict ANSI conformance of all special keywords. HI-TECH C supports various special keywords (for example *port* for I/O port data types). If the `-STRICT` option is used, these keywords are changed to include a double underscore at the beginning (e.g. *__port*) so as to strictly conform to the ANSI standard. Be warned that use of this option may cause problems with some standard header files (e.g. *INTRPT.H*).

4.1.45 -TEK: Generate Tektronix HEX File

The `-TEK` option tells the compiler to generate a Tektronix format HEX file if producing a file with `.HEX` extension. This option has no effect if used with a `.BIN` file.

4.1.46 -U*macro*: Undefine a Macro

`-U`, the inverse of the `-D` option, is used to undefine predefined macros. This option takes the form **-D*macro***. For example, to remove the pre-defined macro *z80* use the option `-Uz80`.

4.1.47 -UBROF: Generate UBROF Format Output File

The `-UBROF` option tells the compiler to generate a UBROF format output file suitable for use with certain in-circuit emulators. The output file will be given an extension `.UBR`. UBROF output may also be selected by specifying an output file of type `.UBR` using the `-O` option. This option has no effect if used with a `.BIN` file.

4.1.48 -UNSIGNED: Make *char* Type Unsigned

`-UNSIGNED` will make the default *char* type *unsigned char.* The default behaviour of the compiler is to make all character values and variables *signed char* unless explicitly declared or cast to *unsigned char.* If `-UNSIGNED` is used, the default character type becomes *unsigned char* and variables will need to be explicitly declared *signed char.* The range of *signed char* is -128 to +127 and the range of *unsigned char* is 0 to 255.

4.1.49 -V: Verbose Compile

`-V` is the “verbose” option. The compiler will display the command lines used to invoke each of the compiler passes. This option may be useful for determining the exact linker options which should be used if you want to directly invoke the HLINK command.

4.1.50 -W*level*: Set Warning Level

`-W` is used to set the compiler warning level. Allowable warning levels range from -9 to 9. The warning level determines how picky the compiler is about dubious type conversions and constructs. The default warning level `-W0` will allow all normal warning messages. Warning level `-W1` will suppress the message “Func() declared implicit int”. `-W3` is recommended for compiling code originally written with other, less strict, compilers. `-W9` will suppress all warning messages. Negative warning levels `-W-1`, `-W-2` and `-W-3` enable special warning messages including compile-time checking of arguments to *printf()* against the format string specified.

4.1.51 -X: Strip Local Symbols

The option `-X` strips local symbols from any files compiled, assembled or linked. Only global symbols will remain in any object files or symbol files produced.

4.1.52 -Z180: Generate Z180 Code

This option is identical to the `-180` option and selects code generation for the Z180 processor, and specifies the appropriate libraries for that processor.

4.1.53 -Zg: Global Optimization

The `-Zg` option invokes global optimization during the code generation pass. This can result in significant reductions to code size and RAM usage.

Features and Runtime Environment

HI-TECH C supports a number of special features and extensions to the C language which are designed to ease the task of producing ROM-based applications. This chapter documents the compiler options and special features which are available. After reading and understanding this manual you should know how to:

❒ compile your C and assembler source files using the ZC command.

❒ link your C application for specific addresses and create binary images, HEX files, UBROF files or symbol files suitable for use with PROM programmers, debuggers, simulators and in-circuit emulators.

❒ configure the console I/O routines so that you can use <stdio.h> routines on your hardware.

❒ set up interrupt vectors and interrupt handlers using only C code.

❒ program memory mapped I/O devices using only C code.

❒ interface between C and assembler code using inline or external assembly language routines.

❒ understand the three memory models used to execute C code on a Z80 and the restrictions which are imposed on your C application.

5.1 Output File Formats

The compiler is able to directly produce a number of the output file formats which are used by common PROM programmers and in-circuit emulators. If you are using the HPDZ integrated environment compiler driver you can select Motorola Hex, Intel Hex, Binary, UBROF, Tektronix Hex, American Automation symbolic Hex, Intel OMF-51 or Bytecraft .COD using the “Output file type” menu item in the “Options” menu. The default behaviour of the ZC command is to produce Intel HEX output. If no output file name or type is specified, ZC will produce an Intel HEX file with the same base name as the first source or object file. Table 5 - 1 on page 116 shows the output format options available with ZC. With any of the output format options, the base name of the output file will be the same as the first source or object file passed to ZC. The “File Type” column lists the filename extension which will be used for the output file.

In addition to the options shown, the `-O` option may be used to request generation of binary or UBROF files. If you use the `-O` option to specify an output file name with a *.BIN* type, for example `-Otest.bin`, ZC will produce a binary file. Likewise, if you need to produce UBROF files, you can use the `-O` option to specify an output file with type *.UBR*, for example `-Otest.ubr`.

Table 5 - 1 Output file formats

Format Name	Description	ZC Option	File Type
Motorola HEX	S1/S9 type hex file	**-MOTOROLA**	**.HEX**
Intel HEX	Intel style hex records (default)		**.HEX**
Binary	Simple binary image	**-BIN**	**.BIN**
UBROF	"Universal Binary Image Relocatable Format"	**-UBROF**	**.UBR**
Tektronix HEX	Tektronix style hex records	**-TEK**	**.HEX**
American Automation HEX	Hex format with symbols for American Automation emulators	**-AAHEX**	**.HEX**
OMF-51	Intel Absolute Object Module Format	**-OMF**	**.OMF**
Bytecraft .COD	Bytecraft code format	**n/a**	**.COD**

Note that the Bytecraft code format is of limited use for the Z80.

5.2 Symbol Files

The ZC `-G` and `-H` options tell the compiler to produce a symbol file which can be used by debuggers and simulators to perform symbolic and source level debugging. The `-H` option produces symbol files which contain only assembler level information whereas the `-G` option also includes C source level information. If no symbol file name is specified, by default a file called *`file`*`.sym` will be produced, where *file* is the basename of the first source file on the command line. For example, to produce a symbol file called `test.sym` which includes C source level information:

```
ZC -Gtest.sym test.c
```

The symbol files produced by these options may be used with in-circuit emulators, and also the *Lucifer* debugger included with the compiler.

5.2.1 Avocet Symbol Tables

The ZC option `-AV` can be used in conjunction with the `-H` option to generate Avocet style symbol tables for use with the AVSIM simulator and certain in-circuit emulators. `-AV` should not be used with the `-G` option as the Avocet symbol table format does not support source level debugging information.

5.3 Predefined Macros

The compiler drivers define certain symbols to the preprocessor (CPP), allowing conditional compilation based on chip type, memory model etc. The symbols defined are listed in Table 5 - 2 on page 117. Each symbol, if defined, is equated to 1.

Table 5 - 2 Predefined CPP symbols

Symbol	Usage
HI_TECH_C	Always set - can be used to indicate that the compiler in use is HI-TECH C.
z80	Always set - can be used to indicate the code is compiled for the z80 family. Note that this symbol is *lower case*.
SMALL_MODEL	Set for small model
LARGE_MODEL	Set for large memory model.
CPM	Set for CP/M model
_HOSTED	Set when running under an operating system, i.e. when using the CP/M model

5.4 Supported Data Types

The ZC compiler supports basic data types of 1, 2 and 4 byte size. All multi-byte types follow *least significant byte first* format, also known as *little endian*. Word size values thus have the least significant byte at the lower address, and double word size values have the least significant byte and least significant word at the lowest address.

5.4.1 8 Bit Integer Data Types

HI-TECH C supports both *signed char* and *unsigned char* 8 bit integral types. The default *char* type is *signed char* unless the ZC `-UNSIGNED` option is used, in which case it is *unsigned char*. *Signed char* is an 8 bit two's complement signed integer type, representing integral values from -128 to +127 inclusive. *Unsigned char* is an 8 bit unsigned integer type, representing integral values from 0 to 255 inclusive. It is a common misconception that the C *char* types are intended purely for ASCII character manipulation. This is not true, indeed the C language makes no guarantee that the default character representation is even ASCII. The *char* types are simply the smallest of up to four possible integer sizes, and behave in all respects like integers. The reason for the name *char* is historical and does not mean that *char* can only be used to represent characters. It is possible to freely mix *char* values with *short*, *int* and *long* in C expressions. On the Z80 the *char* types will commonly be used for a number of purposes, as 8 bit integers, as storage for ASCII characters, and for access to I/O locations. The *unsigned char* type is the most efficient data type on the Z80 and maps directly onto the 8 bit bytes which are most efficiently manipulated by Z80 instructions. It is suggested that *char* types be used wherever possible so as to maximize performance and minimize code size.

5.4.2 16 Bit Integer Data Types

HI-TECH C supports four 16 bit integer types. *Int* and *short* are 16 bit two's complement signed integer types, representing integral values from -32,768 to +32,767 inclusive. *Unsigned int* and *unsigned short* are 16 bit unsigned integer types, representing integral values from 0 to 65,535 inclusive. 16 bit integer values are represented in *little endian* format with the least significant byte at the lower address. Both *int* and *short* are 16 bits wide as this is the smallest integer size allowed by the ANSI standard for C. 16 bit integers were chosen so as not to violate the ANSI standard. Allowing a smaller integer size, such as 8 bits would lead to a serious incompatibility with the C standard. 8 bit integers are already fully supported by the *char* types and should be used in place of *int* wherever possible.

5.4.3 32 Bit Integer Data Types

HI-TECH C supports two 32 bit integer types. *Long* is a 32 bit two's complement signed integer type, representing integral values from -2,147,483,648 to +2,147,483,647 inclusive. *Unsigned long* is a 32 bit unsigned integer type, representing integral values from 0 to 4,294,967,295 inclusive. 32 bit integer values are represented in *little endian* format with the least significant word and least significant byte at the lowest address. 32 bits are used for *long* and *unsigned long* as this is the smallest long integer size allowed by the ANSI standard for C.

5.4.4 Floating Point

Floating point is implemented using 32 bits, formatted as follows:

Table 5 - 3 32-bit floating point format

31	30 24	23 0
Sign - 1 bit	Exponent - 7 bits	Mantissa - 24 bits

The byte containing the sign and the eight exponent bits is at the highest address. The high order byte consists of a sign bit (bit 7) and a 7 bit excess 64 exponent (i.e. an exponent value of 0 is stored as 64). The remaining 3 bytes are a 24 bit mantissa, in twos complement form. The floating point number is always normalised.

Doubles are exactly the same as floats.

5.5 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 charPortvar @ 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.

This construct is primarily intended for equating the address of a C identifier with a microprocessor register. To place a user-defined variable at an absolute address, define it in a separate psect and instruct the linker to place this psect at the required address. See 5.26.3 on page 151.

5.6 Port Type Qualifier

The Z80 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:

```
;z.c: 8: io_port |= 0x40;
            IN    A,(0E0H)
            OR    40H
            OUT   (0E0H),A
;z.c: 9: *pptr = 0x10;
            LD    BC,(_pptr)
            LD    A,10H
            OUT   (C),A
```

The compiler allows I/O ports at addresses above 0FFh to be used. The ZC options `-P8` and `-P16` should be used to select generation of code for either 8 bit or 16 bit port addresses. `-P8` would be used on traditional Z80 systems where only 8 bits of the I/O address space are decoded. `-P16` (which is the default) should be used when generating code for Z80 systems which decode more than 8 address lines. Use of the `-P16` switch will force the `IN A,(C)` and `OUT (C),A` instructions to be used for all I/O as the direct forms of the `IN` and `OUT` instructions should not be used with 16 bit port decoding. If you are generating Z180 code (with the `-180` switch to ZC), the compiler may also use the `IN0` and `OUT0` instructions where appropriate, and it is not necessary to use either `-P16` or `-P8` in conjunction with `-180`.

5.7 Structures and Unions

HI-TECH C supports *struct* and *union* types of any size from one byte upwards. Structures and unions may be passed freely as function arguments and return values. Pointers to structures and unions are fully supported.

5.7.1 Bit Fields in Structures

HI-TECH C fully supports *bit fields* in structures. Bit fields are allocated starting with the least significant bit. Bit fields are allocated within 16 bit words, the first bit allocated will be the least significant bit of the least significant byte of the word. Bit fields are always allocated in 16 bit units, starting from the most significant bit. When a bit field is declared, it is allocated within the current 16 bit unit if it will fit, otherwise a new 16 bit word is allocated within the structure. Bit fields never cross the boundary between 16 bit words, but may span the byte boundary within a given 16 bit allocation unit. For example, the declaration:

```
struct {
        unsigned        hi : 1;
        unsigned        dummy : 14;
        unsigned        lo : 1;
} foo @ 0x10;
```

will produce a structure occupying 2 bytes from address 10h. The field hi will be bit 0 of address 10h, lo will be bit 7 of address 11h. The least significant bit of dummy will be bit 1 of address 10h and the most significant bit of dummy will be bit 6 of address 11h. If a bit field is declared in a structure that is assigned an absolute address, no storage will be allocated, and the fact that 16 bits are reserved is unimportant, so to model a byte location with bit fields, you may simply define the bits as though only one byte was occupied.

Unnamed bit fields may be declared to pad out unused space between active bits in control registers. For example, if `dummy` is never used the structure above could have been declared as:

```
struct {
        unsigned        hi : 1;
        unsigned           : 14;
        unsigned        lo : 1;
} foo @ 0x10;
```

5.8 *Const* and *Volatile* Type Qualifiers

HI-TECH C supports the use of the ANSI type qualifiers *const* and *volatile*. The *const* type qualifier is used to tell the compiler that an object has a constant value and will not be modified. If any attempt is made to modify an object declared *const*, the compiler will issue a warning. User defined objects declared const are placed in a special *psect* called *const*. For example:

```
const int version = 3;
```

The *volatile* type qualifier is used to tell the compiler that an object cannot be guaranteed to retain its value between successive accesses. This prevents the optimizer from eliminating apparently redundant references to objects declared *volatile* because it may alter the behaviour of the program to do so. All I/O ports and any variables which may be modified by interrupt routines should be declared *volatile*, for example:

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

5.9 Special Type Qualifiers

HI-TECH C supports special type qualifiers, *persistent* and *code* to allow the user to control placement of *static* and *extern* class variables into particular address spaces. If the ZC `-STRICT` option is used, these type qualifier are changed to *__persistent* and *__code*. These type qualifier may also be applied to pointers. These type qualifier may not be used on variables of class *auto*; if used on variables local to a function they must be combined with the *static* storage class specifier. You may not write:

```
void func(void)
{
      persistent intintvar;
      .. other code ..
}
```

because *intvar* is of class *auto*. To declare *intvar* as a *persistent* variable local to function *test()*, write:

```
static persistent int intvar;
```

5.9.1 *Persistent* Type Qualifier

By default, any C variables that are not explicitly initialized are cleared to zero on startup. This is consistent with the definition of the C language. However, there are occasions where it is desired for some data to be preserved across resets or even power cycles (on-off-on). The *persistent* type qualifier is used to qualify variables that should not be cleared on startup. In addition, any *persistent* variables will be stored in a different area of memory to other variables, and this area of memory may be assigned to a specific address (with the -A option to ZC, or in the HPDZ **Options**/**ROM and RAM addresses ...**). Thus if a small amount of non-volatile RAM is provided then *persistent* variables may be assigned to that memory. On the other hand if all memory is non-volatile, you may choose to have persistent variables allocated to addresses by the compiler along with other variables (but they will still not be cleared). One advantage of assigning an explicit address for persistent variables is that this can remain fixed even if you change the program, and other variables get allocated to different addresses. This would allow configuration information etc. to be preserved across a firmware upgrade.

There are some library routines provided to check and initialize persistent data - see page 358 for more information, and for an example of using persistent data.

5.9.2 *Code* Type Qualifier

The *code* type qualifier works with large model only and is used to place initialised static objects into a ROM bank with associated code. The initialised static object will be located in the *current ROM bank*. Objects declared to be *code* must be statically initialized, for example:

```
static code char      sometext[] = “abcdef”;
```

will locate `sometext[]` in the current ROM bank. Obviously this will only be accessible to code executing in this bank or in common code, so a pointer to *code* data should not be passed to any functions outside the current bank, including library functions.

5.10 Pointers

HI-TECH C supports several different classes of pointer, of both 16 and 32 bit size. The default pointer class is a 16 bit pointer which addresses a 64K address. The only 32 bit pointer is the *pointer to a function* in large model.

5.10.1 Combining Type Qualifiers and Pointers

The *const*, *volatile*, *persistent* and *code* type qualifiers may also be applied to pointers, controlling the behaviour of the object which the pointer addresses. When using these modifiers with pointer declarations, care must be taken to avoid confusion as to whether the modifier applies to the pointer, or the object addressed by the pointer. The rule is as follows: if the type qualifier is to the left of the “*” in the pointer declaration, it applies to the object which the pointer addresses. If the type qualifier is to the right of the “*”, it applies to the pointer variable itself. Using the *volatile* type qualifier to illustrate, the declaration:

```
volatile char *     nptr;
```

declares a pointer to a *volatile* character. The *volatile* type qualifier applies to the object which the pointer addresses because it is to the left of the “*” in the pointer declaration.

The declaration:

```
char * volatile     ptr;
```

behaves quite differently however. The *volatile* type qualifier is to the right of the “*” and thus applies to the actual pointer variable *ptr*, not the object which the pointer addresses. Finally, the declaration:

```
volatile char * volatile        nnptr;
```

will generate a volatile pointer to a volatile variable.

5.10.2 *Code* Pointers

The *code* pointer works with large model only and is used to point to initialised static objects placed into a ROM bank with associated code. The initialised static object will be located in the *current ROM bank*, and will only be accessible to code executing in this bank or in common code. Therefore a pointer to *code* data should not be passed to any functions outside the current bank, including library functions.

A common use of *code* constants and variables of class *pointer to code* is to access string constants such as menus and prompts which have been placed in ROM. The following code illustrates this technique:

```
#include <conio.h>
static code charhello[] = "Hello, world\n";

static void
code_puts(code char * cptr)
{
      char  ch;
      while (ch = *cptr++)
            putch(ch);
}
main()
{
      code_puts(hello);
}
```

Use of *code* constants and pointers can reduce RAM usage in the large memory model which, depending on the compile-time *-ROMDATA* option, copies initialized variables to RAM.

5.10.3 *Const* Pointers

Pointers to *const* should be used when indirectly accessing objects which have been declared using the *const* qualifier. *Const* pointers behave in nearly the same manner as the default pointer class in each memory model, the only difference being that the compiler forbids attempts to write via a pointer to *const*. Thus, given the declaration:

```
const char *    cptr;
```

the statement:

```
ch = *cptr;
```

is legal, but the statement:

```
*cptr = ch;
```

is not. *Const* pointers always access program ROM because *const* declared objects are stored in ROM.

5.11 Interrupt Handling in C

The compiler incorporates features allowing interrupts to be handled without writing any assembler code. The function qualifier *interrupt* may be applied to a function to allow it to be called directly from a hardware interrupt. The compiler will process *interrupt* functions differently to normal functions, generating code to save and restore any registers used and exit using the RETI instruction instead of a RET instruction at the end of the function. (If the ZC option `-STRICT` is used, the *interrupt* keyword becomes *__interrupt*. Wherever this manual refers to the *interrupt* keyword, assume *__interrupt* if you are using `-STRICT`.)

An *interrupt* function must be declared as type *interrupt void* and may not have parameters. It may not be called directly from C code, but it may call other functions itself, subject to certain limitations. An example of an *interrupt* function follows:

```
long  tick_count;
void interrupt
tc_int(void)
{
      ++tick_count;
}
```

The manner of setting interrupt vectors depends on whether you are using mode 0, 1 or 2 interrupt vectors. Note that the NMI is always handled as though it was a mode 0 or 1 interrupt, even if you are using mode 2 for other interrupts.

5.11.1 Interrupt Handling Macros

The standard header file `<intrpt.h>` contains several macros and functions which are useful when handling interrupts using C code. These are listed inTable 5 - 4 on page 125..

5.11.1.1 The ei() and di() Macros

The *ei()* and *di()* macros may be used to disable and enable maskable interrupts. It may be useful to disable interrupts while initializing or servicing I/O devices. *Di()* disables interrupts by clearing the interrupt enable flags using the `DI` instruction. Similarly, *ei()* enables interrupts by setting the interrupt enable flags using the `EI` instruction.

Table 5 - 4 Interrupt support macros and functions

Macro Name	Description	Example
ei	Enable interrupts	`ei();`
di	Disable interrupts	`di();`
im	Select interrupt mode	`im(2);`
set_vector	Initialise an interrupt vector.	set_vector(BRKINT, brkint_isr);
ROM_VECTOR	Initialize interrupt vector in ROM	`ROM_VECTOR(IV_T0, handler);`
RAM_VECTOR	Initialize interrupt vector in RAM	`RAM_VECTOR(IV_T0, handler);`
CHANGE_VECTOR	Alter vector in RAM	`CHANGE_VECTOR(IV_T0, handler);`
READ_RAM_VECTOR	Get current contents of vector in RAM	`iptr = READ_RAM_VECTOR(IV_T0);`

5.11.1.2 The im() Macro

The *im()* macro is provided to select the interrupt mode. This macro will generate a line of assembler code to select the appropriate interrupt mode. The I register will also be set for **im(2)**. This should be called in your main routine before enabling interrupts. The argument to *im()* is the interrupt mode number, 0, 1 or 2.

For example:

```
        im(2);
```

will select mode 2 interrupts.

5.11.1.3 The set_vector() Function

The *set_vector()* function allows an interrupt vector to be initialised. See the description of this function in the Library Functions chapter.

5.11.1.4 ROM_VECTOR

The ROM_VECTOR, RAM_VECTOR, CHANGE_VECTOR, and READ_RAM_VECTOR macros may only be used with *mode 2* interrupts.

The ROM_VECTOR, RAM_VECTOR and CHANGE_VECTOR macros are used to set up a “hard-coded” table of vectors in ROM, aligned on a 256 byte boundary. The high byte of the base address of this array is loaded into the I register at startup. The table is long enough to hold the highest vector defined.

The macro ROM_VECTOR will statically initialise an entry in this table. It takes the form:

```
            ROM_VECTOR(vector, handler);
```

where `vector` is the vector offset (i.e. the 8 bit value that will be supplied by the interrupting device in response to an interrupt acknowledge) and handler is the name of the function which will handle the interrupt.

ROM_VECTOR generates in-line assembler code, so the `vector` argument may be in any format acceptable to the assembler. Hexadecimal interrupt vector addresses may be passed either as C style hex (0xA0) or as assembler style hex (A0h).

5.11.1.5 RAM_VECTOR

The `RAM_VECTOR` macro sets up a "soft" interrupt vector which can be modified to point to a different *interrupt handler* if necessary. Note that for this to work, the *data* psect must be copied into RAM at run-time. If using `RAM_VECTOR`, the `-ROMDATA` compiler option must not be used, nor may the **Initialised data in RAM** HPDZ option be deselected.

As with **`ROM_VECTOR`**, **`RAM_VECTOR`** statically initialises an entry in the table of vectors. `RAM_VECTOR` takes the same arguments as **`ROM_VECTOR`** and can be used anywhere **`ROM_VECTOR`** is used.

5

5.11.1.6 CHANGE_VECTOR

The `CHANGE_VECTOR` macro is used to modify a vector which has been set up by `RAM_VECTOR` macro. This is accomplished by modifying the interrupt handler address in internal RAM. For example:

```
            di();
            CHANGE_VECTOR(0xA0, new_handler);
            ei();
```

will change the handler address used by vector 0xA0 to point to an interrupt function called *new_handler()*. The address of the vector word in internal RAM is found by a special symbol defined by `RAM_VECTOR`, so `CHANGE_VECTOR` should only be used in the same module and after `RAM_VECTOR` has been used. The vector address should be *identical* otherwise the symbols will not match and a compile-time error will result.

If a vector has been modified and you want to change it back to the original value, you will need to use `CHANGE_VECTOR` to change it back. Re-executing the code which contains the `RAM_VECTOR` macro will not reset the vector because `RAM_VECTOR` statically initializes the vector without generating any executable code. `CHANGE_VECTOR` is the only vector initialization macro which generates instructions which are actually executed at run-time; `ROM_VECTOR` and `RAM_VECTOR` just force initial values into the vectors.

5.11.1.7 READ_RAM_VECTOR

The `READ_RAM_VECTOR` macro may be used to read the value of a RAM based interrupt vector which has been set up by `RAM_VECTOR`. It must never be used on vectors which have been initialized using

ROM_VECTOR as garbage will be returned. READ_RAM_VECTOR can be used along with CHANGE_VECTOR to preserve an old interrupt handler address, set a new address and then restore the original address. For example:

```
volatile unsigned char  wait_flag;
interrupt void
wait_handler(void)
{
        ++wait_flag;
}
void
wait_for_serial_intr(void)
{
        interrupt void    (*old_handler)(void);

        di();
        old_handler = READ_RAM_VECTOR(0xA0);
        wait_flag = 0;
        CHANGE_VECTOR(0xA0, wait_handler);
        ei();
        while (wait_flag == 0)
                continue;
        di();
        CHANGE_VECTOR(0xA0, old_handler);
        ei();
}
```

5.11.2 Interrupt Modes

Mode 0 interrupts are 8080 compatible, i.e. the interrupting device supplies a single instruction (usually a JP or RST), while *mode 1* interrupts always vector to address 0x38 (like a **rst 38h** instruction). Thus both mode 0 and 1 interrupts (and the NMI) require a JP instruction, or other code, at the vector address.

Mode 2 interrupts, on the other hand, fetch a 2 byte address from a memory location in a 256 byte vector page. The high byte of the vector page address is obtained from an internal CPU register, and the low byte is supplied by the interrupting device. The 2 byte address fetched is loaded into the program counter.

5.11.2.1 Setting the Interrupt Mode

The *im()* macro will generate a line of assembler code to select the appropriate interrupt mode. This should be called in your main routine before enabling interrupts.

5.11.2.2 Interrupt Mode 0 and Mode 1

The *set_vector()* library routine is suitable for use with mode 0 or 1 interrupts, and should always be used for the NMI, which always vectors to location 0x66. *Set_vector()* depends on the runtime startoff code having set up `JP` instructions at the vector locations. If you want to directly vector base page interrupts to your interrupt service routines, you can modify the runtime startoff code to directly encode `JP` instructions which point to your code. The interrupt vectors are set up by the code sequence shown below (from the standard run-time startoff module `RTZ80-S.AS`). The run-time startoff modules for bankswitched code and the Z180 are similar.

```
psect vectors

globalstart,_main,_exit,__Hstack,__Hbss, __Lbss
globalpowerup, __Ldata, __Hdata, __Lconst

defb  0c3h  ; JP opcode
defw  init  ; cannot be optimized by ZAS -J option
defb  0,0
jp    bdosvec; jump to BDOS handler
jp    r08vec; jump to RST 08 handler
defb  0,0,0,0,0
jp    r10vec; jump to RST 10 handler
defb  0,0,0,0,0
jp    r18vec; jump to RST 18 handler
defb  0,0,0,0,0
jp    r20vec; jump to RST 20 handler
defb  0,0,0,0,0
jp    r28vec; jump to RST 28 handler
defb  0
jp    r2Cvec; jump to RST 0C handler (NSC800)
defb  0
jp    r30vec; jump to RST 30 handler
defb  0
jp    r34vec; jump to RST 0B handler (NSC800)
defb  0
jp    r38vec; jump to RST 38 handler
defb  0
jp    r3Cvec; jump to RST 0A handler (NSC800)
defb  0,0
defm  "Copyright (C) 1993 HI-TECH Software"
```

```
        defb  0,0
        jp    nmivec      ; jump to NMI handler
;
fakesp:
        defw  start
init:
        ld    sp,fakesp
        jp    powerup
```

The code sequence shown above is linked at address 0 in ROM and redirects all basepage interrupts to a table of JP instructions which are placed in RAM by the runtime startoff code. The *set_vector()* routine follows the JP instruction in the basepage and modifies the RAM based JP instruction to point to the specified routine. For example, the call

```
set_vector((isr  *)0x38,  handleint)
```

will place a JP to *handleint()* at the RAM location pointed to by the RST 38 vector, i.e. r38vec.

This technique makes it possible to dynamically modify interrupt vectors while your code is running. Many ROM applications require only a small number of fixed interrupt vectors. In this case, it may be desirable to edit the run-time startoff module and hard code the basepage vectors to point directly at the interrupt routines. For example, the RST 38 vector could be hard coded to point at *handleint()* by changing it to:

```
      GLOBAL _handleint
      JP     _handleint ;38- RST 38 handler
```

5.11.2.3 Interrupt Mode 2

Interrupt mode 2 is handled with the macros ROM_VECTOR, RAM_VECTOR, CHANGE_VECTOR, and READ_RAM_VECTOR as defined in the standard header file *<intrpt.h>*.

An example using mode 2 interrupts is shown in figure 10.

```
#include   <z180.h>
#include   <intrpt.h>
#include   <stdio.h>

#define    CLOCK   6144000     /* clock frequency in Hz */
#define    TICKS   50          /* ticks per second */
#define    COUNT   ((CLOCK/20)/TICKS)/* timer counts per tick */
#define    BRATE   0x01        /* 19200 baud @ 6.144Mhz */
```

```
static unsigned         time;
static unsigned char    ticks, flag;

static void interrupt
timer_int(void)
{
          if(++ticks == TICKS) {
                 ticks = 0;
                 time++;
                 flag = 1;
          }
          TCR;                /* read TCR to clear flag */
          TMDR0L;             /* and timer reg */
}

void
putch(char c)
{
          if(c == ’\n’) {
                 while (!(STAT0 & 2))
                       continue;
                 TDR0 = ’\r’;
          }
          while (!(STAT0 & 2))      /* wait for TDRE == 1 */
                 continue;
          TDR0 = c;                 /* write character */
}

main(void)
{
                 /* set up interrupt vector */
          ROM_VECTOR(PRT0_VEC, timer_int);
          im(2);                    /* set interrupt mode */
          STAT0 = 0;                /* reset ASCI channel 1 */
          CNTLA0 = 0x64;            /* Enable UART */
          CNTLB0 = BRATE;           /* baud rate */
          RLDR0L = COUNT;           /* set up timer */
          RLDR0H = COUNT >> 8;
```

```
                TCR = 0x11;   /* start timer, enable interrupt */
                ei();
                for(;;) {
                        while(!flag)      /* wait a second! */
                              continue;
                        flag = 0;
                        printf("%6d\n", time);  /* print the time */
                }
      }
```

5.11.3 Predefined Interrupt Vector Names

The header file `z180.h` includes declarations for all of the standard interrupt vectors for Z180 internal interrupts. These vector names may be used as the vector address argument to the `ROM_VECTOR`, *set_vector()*, `RAM_VECTOR`, `CHANGE_VECTOR` and `READ_RAM_VECTOR` macros.

The interrupt vectors defined in `z180.h` are listed in Table 5 - 5 on page 131. Interrupt vectors other than those in `z180.h` may be declared using pre-processor `#define` directives, or the vector address may be directly used with the vector macros.

Table 5 - 5 Z180 interrupt vectors

Name	Value	Use
INT1_VEC	0x00	External /INT1
INT2_VEC	0x02	External /INT2
INTCAP_VEC	0x12	Input capture
OUTCMP_VEC	0x14	Output compare
TIMOV_VEC	0x16	Timer Overflow
PRT0_VEC	0x04	PRT Channel 0
PRT1_VEC	0x06	PRT Channel 1
DMA0_VEC	0x08	DMA Channel 0
DMA1_VEC	0x0A	DMA Channel 1
CSIO_VEC	0x0C	Clocked serial I/O
ASCI0_VEC	0x0E	Async channel 0
ASCI1_VEC	0x10	Async channel 1

5.11.4 Handling Non Maskable Interrupts

The *nmi* function qualifier may be used to declare a function which handles the Z80 non-maskable interrupt. A function declared *nmi interrupt* will exit via a *retn* instruction instead of *reti*. Nmi interrupt functions should only be vectored from the NMI vector at 66h as it is not safe practice to terminate a normal interrupt with retn, or an NMI with reti, due to the way the interrupt enable flip flops are handled on the Z80.

5.11.5 Fast Interrupts

A function declared *fast interrupt* will preserve the general purpose registers by switching register banks using the *EXX* and *EX AF,AF'* instructions. Fast interrupt functions may not be used in programs which make use of the floating point or long (32 bit) arithmetic facilities in the C library as some of the library routines used by floating point code make use of the alternate register set. There is a link time check against this condition; the linker will fail with an appropriate error message if you attempt to use fast interrupts and floating point in the same program. Fast interrupt functions must never be nested; interrupts should be kept disabled for the duration of the function.

5.12 Mixing C and Z80 Assembler Code

Z80 assembly language code can be mixed with C code using three different techniques.

5.12.1 External Assembly Language Functions

Entire functions may be coded in assembly language, assembled by ZAS as separate *.AS* source files and combined into the binary image using the linker. This technique allows arguments and return values to be passed between C and assembler code. To access an external function, first include an appropriate C *extern* declaration in the calling C code. For example, suppose you need an assembly language function to provide access to the rotate left instruction on the Z80:

```
extern char    rotate_left(char);
```

declares an external function called *rotate_left()* which has a return value type of *char* and takes a single argument of type *char*. The actual code for *rotate_left()* will be supplied by an external *.AS* file which will be separately assembled with ZAS. The full Z80 assembler code for *rotate_left()* would be something like:

```
      GLOBAL  _rotate_left
      SIGNAT  _rotate_left,4153
      PSECT   text
_rotate_left:
      RLC     E
      LD      L,E
      RET
```

The name of the assembly language function is the name declared in C, with an underscore prepended. The *GLOBAL* pseudo-op is the assembler equivalent to the C *extern* keyword and the *SIGNAT* pseudo-op is used to enforce link time calling convention checking. Signature checking and the *SIGNAT* pseudo-op are discussed in more detail later in this chapter. Note that in order for assembly language functions to work properly they must look in the right place for any arguments passed and must correctly set up any return values. Local variable allocation, argument and return value passing mechanisms are discussed in detail later in the manual and should be understood before attempting to write assembly language routines.

5.12.2 #asm, #endasm and asm()

Z80 instructions may also be directly embedded in C code using the directives *#asm*, *#endasm* and *asm()*. The *#asm* and *#endasm* directives are used to start and end a block of assembler instructions which are to be embedded inside C code. The *asm()* directive is used to embed a single assembler instruction in the code generated by the C compiler. To continue our example from above, you could directly code a rotate left on a memory byte using either technique as the following example shows:

```
#include        <stdio.h>
unsigned char   var;
main()
{
        var = 1;
        printf("var = 0x%2.2X\n", var);
#asm
        LD      A,(_var)
        RLC     A
        LD      (_var),A
#endasm
        printf("var = 0x%2.2X\n", var);
        asm("LD   A,(_var)");
        asm("RLC  A");
        asm("LD   (_var),A");
        printf("var = 0x%2.2X\n", var);
}
```

When using inline assembler code, great care must be taken to avoid interacting with compiler generated code. If in doubt, compile your program with the ZC `-S` option and examine the assembler code generated by the compiler.

IMPORTANT NOTE: the *#asm* and *#endasm* construct is not syntactically part of the C program, and thus it does *not* obey normal C flow-of-control rules. For example, you cannot use a *#asm* block with an if statement and expect it to work correctly. If you use in-line assembler around any C constructs such as if, while, do etc. they you should use only the *asm("")* form, which is a C statement and will correctly interact with all C flow-of-control structures.

5.13 Signature Checking

The compiler automatically produces signatures for all functions. A signature is a 16 bit value computed from a combination of the function's return data type, the number of its parameters and other information affecting the calling sequence for the function. This signature is output in the object code of any function referencing or defining the function.

At link time the linker will report any mismatch of signatures. Thus if a function is declared in one module in a different way (for example, as *char* instead of *short*) then the linker will report an error.

It is sometimes necessary to write assembly language routines which are called from C using an *extern* declaration. Such assembly language functions need to include a signature which is compatible with the C prototype used to call them. The simplest method of determining the correct signature for a function is to write a dummy C function with the same prototype and compile it to assembly language using the ZC `-S` option. For example, suppose you have an assembly language routine called `_widget` which takes two *int* arguments and returns a *char* value. The prototype used to call this function from C would be:

```
extern char     widget(int, int);
```

Where a call to `_widget` is made in the C code, the signature for a function with two *int* arguments and a *char* return value would be generated. In order to match the correct signature the source code for widget needs to contain an ZAS *SIGNAT* pseudo-op which defines the same signature value. To determine the correct value, you would write the following code:

```
char widget(int arg1, int arg2)
{
}
```

and compile it to assembler code using

```
ZC -S x.c
```

The resultant assembler code includes the following line:

```
signat  _widget,8249
```

The *SIGNAT* pseudo-op tells the assembler to include a record in the *.OBJ* file which associates the value 8249 with symbol `_widget`. The value 8249 is the correct signature for a function with two *int* arguments and a *char* return value. If this line is copied into the *.AS* file where `_widget` is defined, it will associate the correct signature with the function and the linker will be able to check for correct argument passing. For example, if another *.C* file contains the declaration:

```
extern char     widget(long);
```

A different signature will be generated and the linker will report a signature mismatch which will alert you to the possible existence of incompatible calling conventions.

5.14 Linking Programs

The compiler will automatically invoke the linker unless requested to stop after producing assembler code (ZC `-S` option) or object code (ZC `-C` option). To specify your RAM address (for variables and stack) and your ROM address (for code and initialized constants), use the ZC `-A` option. If the `-A` option is not used the compiler will ask for the appropriate ROM and RAM addresses and sizes.

ZC and HPDZ by default generate *Intel hex* files. If you use the `-BIN` option or specify an output file with a *.BIN* filetype using the ZC -O option the compiler will generate a binary image instead. The file will contain code starting from the lowest initialized address in the program. For example:

```
ZC -v -oxx.bin -A0,8000,8000
```

will produce a binary file starting with the vectors, followed by user code, initialized data and library code. After linking, the compiler will automatically generate a memory usage map which shows the address and size of all memory areas which are used by the compiled code. For example:

```
Memory Usage Map:
User:  0069H -  00F2H   008AH (138) bytes
 ROM:  0000H -  0068H   0069H (105) bytes
 ROM:  00F3H -  0DC2H   0CD0H (3280) bytes
 RAM:  FE00H -  FEC6H   00C7H (199) bytes
```

More detailed memory usage information, listed in ascending order of individual psects, may be obtained by using the ZC `-PSECTMAP` option.

5.15 Memory Usage

The compiler makes few assumptions about memory. With the exception of variables declared using the *@address* construct, absolute addresses are not allocated until link time.

5.16 Register Usage

The IX register is used as a stack frame pointer, while IY may be used as a register variable. Register DE and BC are used for register based function argument passing. Registers HL and DE are used for function return values. These registers should be preserved by any assembly language routines which are called.

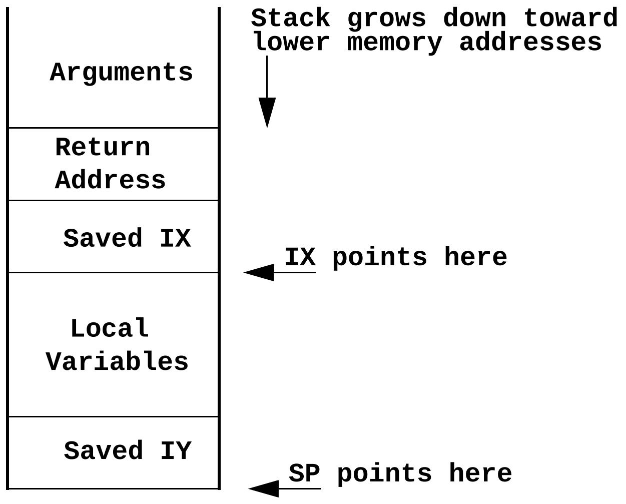
5.17 Stack Frame Organisation

On entry to _main and all other C functions, some code is executed to set up the *local stack frame*. This involves saving non-temporary registers (IX and IY for the Z80) and setting up the base pointer (IX) to point to the base of the new stack frame. IX and IY are saved only if the function uses them. Then the stack pointer is adjusted to allow the necessary space for local variables. Typical code on entry to a function would be:

```
PUSH  IX
LD    IX,0
ADD   IX,SP
LD    HL,-10
ADD   HL,SP
LD    SP,HL
PUSH  IY
```

This will allocate 10 bytes of stack space for local variables. The stack frame after this code will look like that shown in Figure 5 - 1.

Figure 5 - 1 Stack Frame after Function Entry



All references to the local data or parameters are made via IX. The first argument is located at **`(IX+4)`**. Note that if IX is not used (i.e. there are no stack based arguments or local variables accessed in the function) then IX will not be saved. Similarly IY will be saved only if it is used as a register variable.

A parameter may occupy any number of bytes, depending on its type. If a *char* is passed as an argument to a non-prototyped (i.e. K&R style) function, it is expanded to *int* length. Where a function is prototyped, a *char* argument will occupy only one byte, but two bytes will be allocated for it on the stack.

Local variables are accessed at locations (IX-1) downwards. IY may be used as a register variable. If global optimization is used, other registers may also be used as register variables, but are not saved since they are regarded as temporary registers across function calls.

Exit from the function is accomplished by reversing the entry code. This restores the old IX and IY, resets the stack pointer to point to the return address, pops the return address into a register, removes the stack based arguments, and jumps to the saved return address. Example exit code for a function with 4 bytes of stack arguments follows:

```
POP   IY
LD    SP,IX
POP   IX
POP   BC
POP   AF
POP   AF
PUSH  BC
RET
```

The code above saves the return address in the BC register temporarily, removes the 4 bytes of stack arguments by popping them into AF, then returns by pushing the return address onto the stack and executing the a RET instruction. Note that non-prototyped functions, and functions with variable argument lists, leave the stack argument removal to the calling function.

5.18 Function Argument Passing

Some function arguments are passed in the DE and BC registers. This occurs if the function in question has an ANSI-style prototype and either of the first two arguments are words or bytes, *and* the second argument is not the ellipsis symbol (...). In this case the first argument will be placed in DE (for a word) or E (for a byte) and the second argument will be placed in BC or C.

Depending on the function, register based parameters may or may not be stored on the stack at some stage during execution of the function. This feature has nothing to do with a *register* declaration of a parameter. Any arguments passed on the stack will be pushed in strict right to left order. The left most argument will occupy the lowest memory address on the stack.

For example, take the following ANSI-style function:

```
void
test(int a, int b, int c)
{
}
```

Test() will receive argument *a* in the DE register, argument *b* in the BC register and argument *c* on the stack. The call `test(1,2,3)` would generate the following code:

```
LD    HL,3
PUSH  HL
LD    DE,1
LD    BC,2
CALL  _test
```

Note that the *test* removes its own arguments from the stack, thus it is not necessary for the caller to remove argument c from the stack.

For functions declared with an *ANSI-style* prototype, the responsibility for removing the arguments from the stack lies with the function called, not the caller. For example, a function with 10 bytes of stack arguments would exit with code similar to the following:

```
POP   BC    ;BC = return address
LD    HL,10
ADD   HL,SP ;adjust SP by
LD    SP,HL ;10 bytes
PUSH  BC    ;ret addr -> stack
RET
```

Functions declared with *K&R-style* argument lists do not use register arguments and leave stack unjunking to the caller. Functions which return long or float values and use more than 16 bytes of stack arguments will also leave stack unjunking to the caller. A K&R-style function (that is, a function with old-style C arguments) will clear any local variables from the stack and exit using a RET instruction, leaving it to the caller to clear the arguments from the stack. For example, take the old style function:

```
void test(a,b,c)
int         a,b,c;
{
}
```

Test() will receive arguments *a*, *b* and *c* on the stack, pushed in right to left order with *a* at the lowest address. The stack arguments will be removed from the stack by the calling function. For example, the call `test(1,2,3)` would generate the following code:

```
LD    HL,3
PUSH  HL
LD    HL,2
PUSH  HL
LD    HL,1
PUSH  HL
CALL  _test
POP   BC
POP   BC
POP   BC
```

These rules should be kept in mind when writing assembly language routines which are called by C code. It is often helpful to write a dummy C function with the same argument types as your assembler function, and compile to assembler code with the ZC -S option, allowing you to examine the entry and exit code generated. In the same manner, it is useful to examine the code generated by a call to a function with the same argument list as your assembler function.

5.19 Function Return Values

Function return values are passed to the calling function as follows:

5.19.1 8 Bit Return Values

8 bit values (*char*, *unsigned char*) are returned in the L register. 8 bit return values are *not* sign extended into the HL register pair. For example, the statement:

```
ch = char_func()
```

where `char func()` returns a character will generate code as follows:

```
CALL  _char_func
LD    A,L
LD    (_ch),A
```

5.19.2 16 Bit Return Values

16 bit values (*int*, *unsigned int*, *short*, *unsigned short* and *pointer*) are returned in the HL register pair, with the least significant byte in the L register. For example, the statement:

```
i = int_func()
```

where `int func()` returns an integer will generate this code:

```
            CALL  _int_func
            LD    (_i),HL
```

5.19.3 32 Bit Return Values

32 bit values (*long*, *unsigned long*, *float* and *double*) are returned in register pairs DE and HL. The least significant word is in the DE register pair. The statement:

```
            l = long_func()
```

where `long func()` returns a 32 bit quantity will generate:

```
            CALL  _long_func
            LD    (_l),DE
            LD    (_l+2),HL
```

5.19.4 Structure Return Values

Composite return values (*struct* and *union*) are returned by copying the return value to a static buffer in the `bss` psect and returning a pointer to the buffer in the HL register pair. The structure pointer returned in HL is then used either to reference members of the structure or copy the return value to an lvalue. Consider the following C code:

```
            struct tst {
                  int   i, j, k;
            };
            extern struct tst    test(void);
            int       i;
            struct tst   s;
            main()
            {
                  s = test();
                  i = test().j;
            }
```

The compiler will generate the following code for the two assignments.

```
            ;z.c: 12: s = test();
                  CALL  _test
                  LD    DE,_s
                  LD    BC,6
                  LDIR
            ;z.c: 13: i = test().j;
                  CALL  _test
                  INC   HL
                  INC   HL
                  LD    C,(HL)
                  INC   HL
                  LD    B,(HL)
                  LD    (_i),BC
```

In the first case, the entire structure is being assigned to another variable so the return pointer in HL is used as the source pointer for a block move to the lvalue. In the second case only one member of the return value is referenced so the return pointer is used to access the `.j` field without making a local copy of the structure.

When using structure return values you should take into account the extra `bss` psect space allocated for the “hidden” copy of the return value. In the case above, the code generated for function `test()` will include a declaration of a 6 byte buffer used to set up the return value.

5.20 Function Calling Conventions for Large Model

When using the large (bankswitched) model, the calling conventions are similar to the small model except for the actual call. Rather than calling the function directly, register A is loaded with the bank number of the function to be called, HL is loaded with its address within that bank, and then a call is made to a routine in common memory which performs the necessary bank switching before jumping to the function. The current bank is saved on the stack.

On return from the function, the old bank number is retrieved from the stack before a return is made to the calling function. The code to do this on a Z80 is shown following. Note that there are two routines to return; one that removes arguments from the stack (*lretp*) and one that does not (*lret*).

When a function is called in this manner, the compiler accounts for the additional word on the stack when calculating argument offsets.

```
            globallcall, lret, lretp, __Lbasecode
            psect lowtext,class=CODE

      BBR   equ   39h         ;Bank base register


      ;     lcall - perform a far call

      lcall:ex    af,af'      ;save new bank number
```

```
        in0   a,(BBR)     ;get current bank
        push  af          ;save it
        ex    af,af'      ;restore new bank
        sub   __Lbasecode/1000h;adjust bank number
        out0  (BBR),a
        jp    (HL)        ;go to new routine

  lret: pop   af          ;restore bank number
        out0  (BBR),a     ;select it
        ret               ;back to caller

  lretp:push  bc          ;get adjustment
        exx               ;get some unused regs
        pop   hl          ;adjustment value
        pop   af          ;bank number
        pop   de          ;return address
        add   hl,sp
        ld    sp,hl       ;remove parameters
        push  de          ;push return address back
        exx
        out0  (BBR),a     ;restore bank
        ret               ;done.
```

5.20.1 *Near* and *Basenear* Functions in Large Model

When using the *large* (i.e. banked) model, functions are called using the mechanism described above by default. It is, however, possible to define functions that are called via a simple *call/ret* sequence, thus speeding up the code. The two ways to do this are with *near* functions and with *basenear* functions.

A *near* function can be called only from within the same bank, while a *basenear* function resides in the common area 0 and can be called from any bank. Near functions should be declared static and called and called only from within the same module. The following code shows an example of these kind of functions.

```
static nearint
read_port( void)
{
      while(STATUS & 0x80)
            ;
      return DATA;
}

basenear void
kick_dog( void)
{
      WATCHDOG = 1;
}
```

5.21 Stack and Heap Allocation

As previously mentioned, the stack grows downwards. On startup, the stack pointer is initialized to the top of available memory. For CP/M this is the base of the BDOS. For embedded code, this is defined by the *ram address* and *ram size* defined at link time. The stack is set to the sum of these two values.

The heap is the area of memory dynamically allocated via *sbrk()*, *calloc()* or *malloc()*. It grows upwards from the top of statically allocated memory. This is defined by the top of the *bss* psect, which the linker gives the name *__Hbss*. *Sbrk()* checks the amount of memory left between the top of the heap and the bottom of the stack, and if less than 1k bytes 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 appropriate library (e.g. `Z80-SC.lib`).

5.22 Local Variables

C supports two classes of local variables in functions: *auto* variables which are normally allocated on some sort of stack and *static* variables which are always given a fixed memory location.

5.22.1 Auto Variables

Auto variables are the default type of local variable. Unless explicitly declared to be *static* a local variable will be made *auto*. Auto variables are allocated on the stack and referenced by indexing off the IX register. The variables will not necessarily be allocated in the order declared - in contrast to parameters which are always in lexical order. Note that most type qualifiers cannot be used with auto variables, since there is no control over the storage location. Exceptions are *const* and *volatile*.

5.22.2 Static Variables

Static variables are allocated in the *bss* psect and occupy fixed memory locations which will not be overlapped by storage for other functions. *Static* variables are local in scope to the function which they are declared in, but may be accessed by other function via pointers. *Static* variables are guaranteed to retain their value between calls to a function, unless explicitly modified via a pointer. *Static* variables are not subject to any architectural limitations on the Z80.

5.23 Compiler Generated Psects

The compiler splits code and data objects into a number of standard program sections (referred to as “psects”). The HI-TECH assembler allows an arbitrary number of named psects to be included in assembler code. The linker will group all data for a particular psect into a single segment. If you are using ZC or HPDZ to invoke the linker, you don’t need to worry about the information documented here, except as background knowledge. If you want to run the linker manually, or write your own assembly language subroutines you should read this section carefully. The psects used by compiler generated code are:

vectors The *vectors* psect contains the reset vector and the NMI vector. *Vectors* is normally linked for address 0 in ROM so that the Z80 will fetch the reset vector contents from zero on reset. The code in this psect comes from the run-time startoff module.

lowtext is used in the large model for code which must go in the common area zero, for example, the startup code.

text is used for all executable code in the small model. User-written assembly language subroutines should also be placed in the *text* psect. When using the large model, C code will be placed in *ltext* psects, which will be distributed over various banks, but some library functions are placed in the *text* psect in the common area. All interrupt functions are placed in the text psect.

If multiple ROMs are being used (small model only) then the contents of the *text* psect may be spread over several memory ranges. Code in this psect comes from user modules and libraries.

ltext In large (bankswitched) model, program code from user modules (and library modules written in C) is placed into local psects called *ltext* rather than the *text* psect. This allows the linker to assign these psects to banks. Each *ltext* psect will have a different *load* and *link* address. The load address is the physical address in ROM, the link address will be within the banked area in the 64K logical address space. Since these are local psects, they cannot be referenced by name in a linker command line, so the psect class *basecode* is defined to hold these psects. This class is used in the linker command line to collectively refer to all *ltext* psects.

strings The *strings* psect is used for all unnamed string constants, such as string constants passed as arguments to routines like *printf()* and *puts()*. This psect is linked into ROM, since it does not need to be modifiable.

const is used for all initialized constants of class *const*. For example:

```
const char      masks[] = { 1,2,4,8,16,32,64,128 };
```

This psect is linked into ROM, since it does not need to be modifiable.

im2vecs When using mode 2 interrupts, it is necessary to have a block of memory allocated for the mode 2 interrupt vector table. This table must start on a 256 byte boundary, and can be up to 256 bytes long. This psect represents that block of memory. It will be linked into ROM after all other psects in ROM, and aligned on a 256 byte boundary.

basecode is a zero-length psect which specifies the physical address of the bankswitched ROM. In conjunction with the bank area size, it is used to generate address for banked code, and bank numbers for use at run-time.

baseram is another zero-length psect used to specify the physical address of RAM. This allows the Z180 run-time startoff code to set up the mapping of common area 1 to the RAM.

ramstart is another zero-length psect which is provided to specify the start of the RAM area (common area 1 in the Z180 bankswitched model). Depending on other link options, either the *data* or *bss* psects will start at the beginning of RAM.

data The *data* psect is used to contain all statically initialized data except those in classes *code* or *const*.

The *data* psect is linked into ROM, since it is initialised, and is copied into RAM at startup unless link-time options specify otherwise. If the *data* psect remains in ROM, no RAM will be occupied by it, but it cannot be modified at run-time.

bss The *bss* psect is used for all uninitialized static and extern variables which reside in RAM. *Bss* is cleared to all zeros by the runtime startoff code before *main()* is invoked.

stack is a psect used to specify the top of RAM memory. It does not contain any data and will always be of zero length. The base address of this psect (from the assembler value *__Lstack*) is loaded into the stack pointer in the run-time start-off code.

heap is another zero-length psect used to determine the highest RAM location used. The run-time heap used for allocating dynamic memory will start from this location, and grow up towards the stack (which grows downwards).

nvram This psect is used to store *persistent* variables. It can be assigned an absolute address at link time, or by default the compiler driver will concatenate it with the *bss* psect. It is not cleared or otherwise modified at startup.

5.24 Runtime Startoff Modules

The starting address of a C program is usually the lowest code address; for ROM applications this is the lowest address in ROM, usually 0h. For CP/M the start address is the base of the TPA, 100h. The global symbol *start* is at this address. The code located at the start address performs some initialisation, notably clearing of the *bss* (uninitialized data) psect and initialising the stack pointer. In the case of CP/M it also calls a routine to set up the *argv* and *argc* values which are passed to `main()`. Depending on compile-time options, this routine may or may not expand wild cards in file names (? and *) and perform I/O redirection.

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".

The run-time startup code is provided by a standard module found in the **LIB** directory, chosen from one of those listed in Table 5 - 6 on page 146.

Table 5 - 6 Standard run-time startoff modules

Model	Z80 Startoff Module	Z180 Startoff Module
Small	RTZ80-S.OBJ	RTZ180S.OBJ
CP/M	RTZ80-C.OBJ	RTZ180C.OBJ
Large	RTZ80-L.OBJ	RTZ180L.OBJ

See the assembler file corresponding to the object file for the startup module source code.

5.24.1 The *powerup* routine

Some hardware configurations require special initialization, often within the first few cycles of execution after reset. Rather than having to modify the run-time startoff module to achieve this there is a hook to the reset vector provided via the *powerup* routine. This is a user-supplied assembler module that will be executed immediately on reset. Often this can be embedded in a C module as embedded assembler code.

The *powerup* routine is called with the stack pointer set to point at a fake return address embedded in the runtime startoff code, thus it does not require any RAM to be working for the stack. The *powerup* routine should be written assuming that little or no RAM is working and should only use system resources after it has tested and enabled them. The following example code, embedded in a C file, waits for dynamic RAM to stabilise after reset, then performs a RAM test. If the test passes, it returns (via the hard-coded return address), otherwise it illuminates a LED and hangs:

```
#asm
;
        PSECT text
        GLOBALpowerup
;
```

```
; Sample powerup routine - tests RAM, leaves
; a LED on and hangs if there is a RAM fault
;
; Assumes RAM: 2000H to 3FFFH, LED: port 0FEH
;
LED   EQU   0FEH
RAMSTARTEQU 2000H
RAMSIZEEQU  2000H
;
; Loop, allowing DRAM time to stabilise
;
tbytes:DEFB 55H,0AAH,0FFH,0
;
powerup:LD  BC,0FFFFH
1:    DEC   BC
      LD    A,B
      OR    C
      JR    NZ,1b
;
; Main test loop - test RAM with each of 4
; test values: 55H,0AAH,0FFH and 0
;
      LD    B,4
      LD    HL,tbytes
2:    LD    A,1
      OUT   (LED),A;LED off
      LD    A,(HL);get test val
      EXX         ;preserve HL & B
      LD    HL,RAMSTART
      LD    DE,RAMSTART+1
      LD    BC,RAMSIZE-1
      LD    (HL),A
      EX    AF,AF';preserve A
      LDIR        ;fill memory
;
; loop for a while, leaving time for
; refresh problems to show up
;
```

```
      LD    BC,0FFFFH
3:    DEC   BC
      LD    A,B
      OR    C
      JR    NZ,3b
;
      LD    A,0
      OUT   (LED),A;LED off
      EX    AF,AF';restore A
;
; Loop through memory testing against
; the value stored.
;
      LD    HL,RAMSTART
      LD    BC,RAMSIZE
4:    CP    (HL)  ;check for match
      JR    NZ,ramerr
      INC   HL
      DEC   BC
      LD    A,B
      OR    C
      JR    NZ,4b
;
      EXX         ;restore regs
      INC   HL    ;next test val
      DJNZ  2b
      RET
;
; ramerr: turn LED on and hang
;
ramerr:LD   A,1
      OUT   (LED),A
hang: JR    hang
;
#endasm
```

The standard libraries contain default *powerup* routines as shown in Table 5 - 7 on page 149.

Table 5 - 7 Default powerup actions

Model/Processor	Library	Default Powerup Action
Small / Z80	Z80-SC.lib	Nothing
Large / Z80	Z80-LC.lib	No default: you *must* provide your own routine
Small / Z180	Z801SC.lib	Sets up common area 1 to reference RAM
Large / Z180	Z801LC.lib	Sets up bank area and common area 1

5.24.2 Using Linker Defined Symbols

In order for the runtime startoff code to clear the *bss* psect and copy the *data* psect, it must determine the load address, link address and size of these psects. This is achieved using special linker generated symbols.

The link address of a psect can be obtained from the value of a global symbol with name `__L`*`name`* where *name* is the name of the psect. For example, `__Lbss` is the low bound of the *bss* psect. The highest address of a psect (i.e. the link address plus the size) is symbol `__H`*`name`*. If the psect has different load and link addresses, as may be the case if the *data* psect is linked for RAM operation, the load address is `__B`*`name`*.

5.24.2.1 Clearing the *bss* Psect

In the standard Z80 runtime model, the *bss* psect has the same link and load addresses, so `__Bbss` is not defined. *Bss* is cleared by zeroing `__Hbss-__Lbss` bytes of memory, starting at address `__Lbss`.

5.24.2.2 Copying the *data* Psect

The *data* psect is linked both in RAM and in ROM. The runtime startoff code must copy the ROM image to RAM. This is achieved by copying `__Hdata-__Ldata` bytes from address `__Bdata` in ROM to `__Ldata` in RAM.

5.24.2.3 Initialising the Stack

When ZC or HPDZ invokes the linker, it includes a psect called *stack* in the linker options. The *stack* psect does not contain any data objects and is simply used as a convenient method of specifying the initial value of the stack pointer. The address given to the *stack* psect by ZC or HPDZ is the user specified RAM address plus the user specified RAM size. For example, if you use the ZC option -A0,8000,4000, the *stack* psect will be given address 8000H + 4000h = 0C000h. The run-time startoff code loads the value of `__Lstack` into the stack pointer before executing any user code.

5.24.3 Customizing the Runtime Startoff Code

If you find that you are running out of ROM space, you may wish to reduce the ROM size further by customizing the runtime startoff code. The standard runtime startoff modules contains code to clear the *bss* psect and copy the *data* psect. Code space be saved by creating a custom version of the runtime startoff code which performs only those initializations required by your application. The source code for the runtime startoff code may be found in the **SOURCES** subdirectory, if installed. Several different versions of the runtime startoff code are supplied, these are listed in Table 5 - 6 on page 146.

5.24.3.1 Copyright Message

The runtime startoff code includes a HI-TECH C Software copyright message which is linked at the start of ROM. If you are running out of code space you may wish to delete the copyright message, saving a small amount of ROM. If ROM space permits, the HI-TECH Software copyright message should be left intact. You may also wish to add your own copyright message.

5.24.3.2 Using the New Runtime Startoff Code

Once you have modified the runtime startoff code to suit your needs, reassemble it using the command:

```
ZAS -x RTZ80xx.AS
```

where RTZ80xx.AS is the name of the startoff module which you have modified. The runtime startoff code should be copied to the library subdirectory. For most installations it will be C:\HT-Z80\LIB. Once the modified startoff code is present in the library directory, you need only relink your application to use your modifications. If you are using customized runtime startoff code, you need to be careful when making further modifications to your application. If you are running a minimal startoff module and add code which uses the *bss* psect or RAM based *data* you may have problems. Without the *bss* clear code, any uninitialized variables in external RAM will not be cleared to zero before *main()* is invoked. More importantly, if the initialized data is used, but the *data* copy code is missing, statically initialized variables will not be set to their correct values at startup.

5.25 Optimizing Code for the Z80

Care needs to be taken to avoid writing code which will be large or inefficient. To improve execution speed and reduce code size, some or all of these suggestions can be used:

❒ Use 8 bit quantities (*char* or *unsigned char*) where appropriate rather than 16 bit quantities, as this consumes less space and can be accessed more quickly.

5.26 Pragma Directives

There are certain compile-time directives that can be used to modify the behaviour of the compiler. These are implemented through the use of the ANSI standard *pragma* facility. The format of a pragma is:

```
#pragma keyword options
```

where **keyword** is one of a set of keywords, some of which are followed by certain options. A list of the keywords is given in Table 5 - 8 on page 151. Each keyword is discussed below.

Table 5 - 8 Pragma directives

Directive	Meaning	Example
jis	Enable JIS character handling in strings	`#pragma jis`
nojis	Disable JIS character handling (default)	`#pragma nojis`
printf_check	Enable printf-style format string checking	`#pragma printf_check(printf)`
psect	Rename compiler-defined psect	`#pragma psect text=mytext`
strings	Define constant string qualifiers	`#pragma strings code`

5.26.1 The #pragma jis and nojis Directives

If your code includes strings with two-byte characters in the JIS encoding for Japanese and other national characters, the **#pragma jis** directive will enable proper handling of these characters, specifically not interpreting a back-slash (\) character when it appears as the second half of a two byte character. The **nojis** directive disables this special handling. JIS character handling is off by default.

5.26.2 The #pragma printf_check Directive

Certain library functions accept a format string followed by a variable number of arguments in the manner of *printf()*. Although the format string is interpreted at run-time, it can be compile-time checked for consistency with the remaining arguments. This directive enables this checking for the named function, e.g. the system header file <*stdio.h*> includes the directive **#pragma printf_check(printf)** to enable this checking for *printf()*. You may also use this for any user-defined function that accepts printf-style format strings. Note that the warning level must be set to -1 or below for this option to have effect.

5.26.3 The #pragma psect Directive

Normally the object code generated by the compiler is broken into the standard psects as already documented. This is fine for most applications, but sometimes it is necessary to redirect variables or code into different psects when a special memory configuration is desired. For example, if the hardware includes an area of memory which is battery backed, it may be desirable to redirect certain variables from *bss* into a psect which is not cleared at startup (although this particular function is provided as a standard feature). Code and data for any of the standard C psects may be redirected using a `#pragma psect` directive. For example, if all executable code generated by a particular C source file is to be placed into a psect called *altcode*,the following directive should be used:

```
#pragma psect  text=altcode
```

This directive tells the compiler that anything which would normally be placed in the `text` psect should now be placed in the `altcode` psect. Any given psect should only be redirected once in a particular source file, and all psect redirections for a particular source file should be placed at the top of the file, below any #includes and above any other declarations. For example, to declare a group of uninitialized variables which are all placed in a psect called *xram*, the following technique should be used:

```
---File XRAM.C
#pragma psect   bss=xram
char    buffer[20];
int     var1, var2, var3;
```

Any files which need to access the variables defined in XRAM.C should #include the following header file:

```
--File XRAM.H
extern char     buffer[20];
extern int      var1, var2, var3;
```

The #pragma psect directive allows code and data to be split into arbitrary memory areas. Definitions of code or data for non-standard psects should be kept in separate source files as documented above. When linking code which uses non-standard psect names, you will not be able to use the ZC -A option to specify the link addresses for the new psects, instead you will need to use the ZC -L option to specify an extra linker option, use the linker manually or use an HPDZ project to compile and link your code. If you want a nearly standard configuration with the addition of only an extra psect like *xram*, you can use the ZC -L option to add an extra -P specification to the linker command. For example:

```
ZC -L-Pxram=1000h/20000h -A0,8000,8000 test.obj xv.obj
```

will link TEST.OBJ and NV.OBJ with a standard configuration of ROM at 0h, RAM at 8000h, and the extra *xram* psect at 1000h in RAM, but not overlapping any valid ROM load address. If you are using the HPDZ integrated environment you can set up a project file by selecting **Start New Project**, add the names of your four source files using **Source Files ...** and then modify the linker options to include any new psects by selecting **Linker Options ...**.

5.26.4 The #pragma strings Directive

Any user-defined variables can be qualified by a number of type qualifiers (see Special Type Qualifiers on page 121) but constant strings (i.e. anonymous strings embedded in expressions) normally are unqualified. This means they will be put into the data segment. To control this behaviour, the **#pragma psect strings** directive allows you to specify a set of qualifiers to be applied to all subsequent constant strings. If a qualifier is specified, it will be added to any qualifiers specified previously. Using the directive without a qualifier will remove all qualifiers from any subsequent strings, i.e. restore to normal.

For example., to qualify strings with *code* you should use the example given in Table 5 - 8 on page 151. Note that all constant strings will then have type *pointer to code char* and will not be usable where a simple *pointer to char* is expected.

5.26.5 The #pragma switch Directive

The compiler generates several different kinds of code for *switch* statements. Usually the compiler will choose the smallest code for a given switch. The major methods are a *direct* switch, where a jump table is indexed by the value being switched on, and a *simple* switch, where a sequence of comparisons are done. A direct switch operates in constant time and will usually be faster on average than a simple switch, but can be quite large for a sparse set of case labels. If you have a particular need for a deterministic time switch, you can select a direct switch with this directive. A **#pragma switch** directive will have effect only for the immediate next switch statement, and only if this appears in the same function. The possible arguments to this directive are **auto**, which restores behaviour to the default, **direct**, which selects a direct switch, and **simple** which selects a simple switch. See the example in Table 5 - 8 on page 151.

5.27 Standard I/O Functions and Serial I/O

A number of the standard I/O functions are provided in the C library with the ZC compiler, specifically those functions intended to read and write formatted text on standard output and input. A list of the available functions is in Table 5 - 9 on page 153. More details of these functions are in the Library

Table 5 - 9 Supported STDIO functions

Function name	Purpose
puts(char * s)	Writes a string to stdout, appends newline
char * gets(char * buf)	Gets a line of text from stdin to buf, removes newline
printf(char * s, ...)	Formatted printing to stdout
putchar(int c)	Puts a single character to stdout
scanf(char *, ...)	Reads formatted input from stdin
sprintf(char * buf, char * s, ...)	Writes formatted text to buf
sscanf(char * buf, char * s, ...)	Reads formatted text from buf
vprintf(char * s, va_arg list)	Version of printf taking argument list
vscanf(char * s, va_arg list)	Version of scanf taking argument list
vsprintf(char * buf, char * s, va_arg list)	Version of sprintf taking argument list
vsscanf(char * buf, char * s, ...)	Version of sscanf taking argument list

Functions chapter. You must link either *sersio.c* (for Z80 SIO) or *ser180.c* (for Z180) before any characters written or read using these functions will be sent or received. Look at these files, which are located in the *sources* directory, to ensure they support your hardware.

If the code is run under the Lucifer debugger, link in *ser180.c*. The host portion of the debugger will act as a dumb terminal. If you wish to send or receive characters via some other means (e.g. to an LCD display) you should modify one or other of the modules in *sersio.c* or *ser180.c*.

To replace one of these modules with your own version, copy the source code from the SOURCES directory to a file in your working directory, make whatever changes are needed, and include this file in your project - either in the Source files list in HPDZ or on the command line to ZC. You must retain all the functions present in the module, even if some are unused (in which case they can be empty functions). Failure to do so may lead to multiply defined symbol messages at link time.

The Z80 Macro Assembler

The HI-TECH Software Z80 Macro Assembler assembles source files for the Z80 and Z180 family of microprocessors. This chapter describes the usage of the assembler and the directives (assembler pseudo-ops and controls) accepted by the assembler.

The HI-TECH assembler package includes a linker, librarian, cross reference generator and an object code converter.

6.1 Assembler usage

The assembler is called ZAS and is available to run under the UNIX and MS-DOS operating systems. Note that the assembler will not produce any messages unless there are errors or warnings - there are no "assembly completed" messages.

The usage of the assembler is similar under all of these operating systems. All command line options are recognised in either upper or lower case. The basic command format is shown:

```
zas [ options ] files ...
```

Files is a space separated list of one or more assembler source files. Where more than one source file is specified the assembler treats them as a single module, i.e. a single assembly will be performed on the concatenation of all the source files specified. The files must be specified in full, no default extensions or suffixes are assumed.

Options is an optional space separated list of assembler options, each with a minus sign (`-`) as the first character. A full list of possible options is given in Table 6 - 1 on page 156, and a full description of each option follows.

6.2 Assembler options

The command line options recognised by ZAS are as follows:

`-`*C* A cross reference file will be produced when this option is used. This file, called `srcfile.crf` where `srcfile` is the base portion of the first source file name, will contain raw cross reference information. The cross reference utility CREF must then be run to produce the formatted cross reference listing.

`-`*E* The default format for an error message is in the form:

```
filename:line: message
```

Table 6 - 1 ZAS assembler options

Option	Meaning	Default
-C	Produce cross-reference	No cross reference
-E***format***	Set error format	
-I	List macro expansions	Don't list macros
-J	Optimise jumps to branches	Don't optimise jumps
-L***listfile***	Produce listing	No listing
-N	Ignore arithmetic overflow	
-O***outfile***	Specify object name	srcfile.OBJ
-P***length***	Specify listing form length	66
-S	No size error messages	
-U	No undef'd symbol messages	
-V	Produce line number info	No line numbers
-W***width***	Specify listing page width	80
-X	No local symbols in OBJ file	

where the error of type *message* occurred on line *line* of the file *filename*. The `-E2` option will produce a less-readable format which is used by HPD.

`-`*I* This option forces listing of macro expansions and unassembled conditionals which would otherwise be suppressed by a `*LIST OFF` assembler control. The `-L` option is still necessary to produce a listing.

`-`*J* This requests the assembler to attempt to assemble jumps and conditional jumps as relative branches where possible. Only those conditional jumps with branch equivalents will be optimised, and jumps will only be optimised to branches where the target is in branch range. Note that the use of this option slows the assembly down, as the assembler must make an additional pass over the input code.

`-`*Llistfile* This option requests the generation of an assembly listing. If *listfile* is specified then the listing will be written to that file, otherwise it will be written to the standard output.

`-`*N* This suppresses the normal check for arithmetic overflow.

The assembler follows the *"Z80 Assembly Language Handbook"* in its treatment of overflow, and in certain instances this can lead to an error where in fact the expression does evaluate to what the user intended. The `-N` option may be used to override the overflow checking.

`-`*O* By default the assembler determines the name of the object file to be created by stripping any suffix or extension (i.e. the portion after the last dot) from the first source file name and appending .OBJ. The `-O` option allows the user to override the default and specify and explicit filename for the object file.

-*Plength*	The default listing pagelength is 66 lines (11 inches at 6 lines per inch). The `-P` option allows a different page length to be specified.
-*S*	If a byte-size memory location is intialized with a value which is too large to fit in 8 bits, then the assembler will generate a "Size error" message. Use of the `-S` option will suppress this type of message.
-*U*	Undefined symbols encountered during assembly are treated as external, however an error message is issued for each undefined symbol unless the `-U` option is given. Use of this option suppresses the error messages only, it does not change the generated code.
-V	This option will include in the object file produced by the assembler line number and file name information for the use of a debugger. Note that the line numbers will be assembler code lines - when assembling a file produced by the compiler, there will be *line* and *file* directives inserted by the compiler so this option is not required.
-*Wwidth*	This option allows specification of the listfile paper width, in characters. *Width* should be a decimal number greater than 41. The default width is 80 characters.
-*X*	The object file created by the assembler contains symbol information, including local symbols, i.e. symbols that are neither public or external. The `-X` option will prevent the local symbols from being included in the object file, thereby reducing the file size.

6.3 Z80 Assembly language

The source language accepted by the HI-TECH Software Z80 Macro Assembler is described below. All opcode mnemonics and operand syntax are strictly Z80 assembly language.

6.3.1 Character set

The character set used is standard 7 bit ASCII. Alphabetic case is significant for identifiers, but not opcodes and reserved words. Tabs are treated as equivalent to spaces.

6.3.2 Constants

6.3.2.1 Numeric Constants

The assembler performs all arithmetic as signed 32 bit. Errors will be caused if a quantity is too large to fit in a memory location. The default radix for all numbers is 10. Other radices may be specified by a trailing base specifier as given in Table 6 - 2 on page 158.

Hexadecimal numbers must have a leading digit (e.g. 0ffffh) to differentiate them from identifiers. Hexadecimal constants are accepted in either upper or lower case.

Table 6 - 2 ZAS numbers and bases

Radix	Format
Binary	digits 0 and 1 followed by *B*
Octal	digits 0 to 7 followed by *O*, *Q*, *o* or *q*
Decimal	digits 0 to 9 followed by *D*, *d* or nothing
Hexadecimal	digits 0 to 9, A to F preceded by *Ox* or followed by *H or h*

Note that a binary constant must have an upper case B following it, as a lower case b is used for temporary (numeric) label backward references.

Real numbers are accepted in the usual format for DEFF directives only. The exponent and mantissa of a real number must be decimal. When using the DEFF directive, real numbers are stored in 32 bit HI-TECH C format. A real number should include a decimal point, but the exponent and sign are optional. If the exponent is present, it should follow the mantissa without any intervening white space.

6.3.2.2 Character Constants

A character constant is a single character enclosed in single quotes (**'**). Multi character constants may be used only as an operand to a **DEFM** pseudo-op.

6.3.2.3 Opcode Constants

Any Z80 opcode may be used as a constant in an expression. The value of the opcode in this context will be the byte that the opcode would have assembled to if used in the normal way. If the opcode is a 2-byte opcode (**CB** or **ED** prefix byte) only the second byte of the opcode will be used. This is particularly useful when setting up jump vectors. For example:

```
ld    a,jp  ;a jump instruction
ld    (0),a ;0 is jump to warm boot
ld    hl,boot;done here
ld    (1),hl
```

6.3.3 Delimiters

All numbers and identifiers must be delimited by white space, non-alphanumeric characters or the end of a line.

6.3.4 Special characters

There are a few characters that are special in certain contexts. Within a macro body, the character @ is used for token concatenation. In a macro argument list, the angle brackets < and > are used to quote macro arguments.

6.3.5 Identifiers

Identifiers are user-defined symbols representing memory locations or numbers. A symbol may contain any number of characters drawn from the alphabetics, numerics and the special characters dollar ($), question mark (?) and underscore (_). The first character of an identifier may not be numeric. The case of alpahabetics is significant, e.g. *Fred* is not the same symbol as *fred*. Some examples of identifiers are shown here:

```
An_identifier
an_identifier
an_identifier1
$$$
?$_12345
```

6.3.5.1 Significance of Identifiers

Users of other assemblers that attempt to implement forms of data typing for identifiers should note that this assembler attaches no significance to any symbol, and places no restrictions or expectations on the usage of a symbol. The names of *psects* (program sections) and ordinary symbols occupy separate, overlapping name spaces, but other than this, the assembler does not care whether a symbol is used to represent bytes, words or chicken sheds. No special syntax is needed or provided to define the addresses of bits or any other data type, nor will the assembler issue any warnings if a symbol is used in more than one context. The instruction and addressing mode syntax provide all the information necessary for the assembler to generate correct code.

6.3.5.2 Assembler Generated Identifiers

Where a LOCAL directive is used in a macro block, the assembler will generate a unique symbol to replace each specified identifier in each expansion of that macro. These unique symbols will have the form *??nnnn* where *nnnn* is a 4 digit number. The user should avoid defining symbols with the same form.

6.3.5.3 Location Counter

The current location within the active program section is accessible via the symbol $.

6.3.5.4 Register symbols

All registers are available by using their standard names, e.g. BC de etc. Case of register names is not significant. It is not possible to equate a symbol to a register.

6.3.5.5 Labels

A label is a name at the beginning of a statement, terminated by a colon, which is assigned a value equal to the current offset within the current psect (program section). A label may be any symbol (including numeric labels). A label is not the same as a macro name, which also appears at the beginning of the line in a macro declaration, but is not followed by a colon.

6.3.5.6 Temporary labels

The assembler implements a system of temporary labels (as distinct from the local labels used in macros) which relieves the programmer from creating new labels within a block of code. A temporary label is defined as a numeric string, and may be referenced by the same numeric string with either an ‘f’ or ‘b’ suffix. When used with an ‘f’ suffix, the label reference is the first label with the same number found by looking *f*orward from the current location, and conversely a ‘b’ will cause the assembler to look *b*ackward for the label.

For example:

```
entry_point:;This is referenced from far away
      ld    b,10
1:    dec c
      jr    nz,2f ;Zero?, branch forward to 2:
      ld    c,8
      djnz 1b     ;Decrement, branch back to 1:
      jr    1f    ;This does not branch to the
                  ;same label as the djnz.
2:    call fred   ;Get here from the jr nz,2f
1:    ret         ;Get here from the jr 1f
```

Note that even though there are two 1: labels, no ambiguity occurs, since each is referred to uniquely. The *djnz 1b* refers to a label further back in the source code, while *jr 1f* refers to a label further forward. In general, to avoid confusion, it is recommended that within a routine you do not duplicate numeric labels.

6.3.6 Strings

A string is a sequence of characters not including carriage return or newline, enclosed within matching quotes. Either single (’) or double (") quotes may be used, but the opening and closing quotes must be the same. A string used as an operand to a DB directive may be any length, but a string used as operand to an instruction must not exceed 1 or 2 characters, depending on the size of the operand required.

6.3.7 Expressions

Expressions are made up of numbers, symbols, strings and operators. Operators can be unary (one operand, e.g. **`.NOT.`**) or binary (two operands, e.g. **`+`**). The operators allowable in expressions are listed in Table 6 - 3 on page 161. The usual rules governing the syntax of expressions apply. Operators starting with a dot "**`.`**" should be delimited by spaces, thus

```
        label .and. 1
```

is valid but

```
        label.and.1
```

is not.

Table 6 - 3 Operators

Operator	Purpose
&	Bitwise AND
*	Multiplication
+	Addition
-	Subtraction
/	Division
<	Signed less than
=	Equality
>	Signed greater than
<=	Signed less than or equal to
>=	Signed greater than or equal to
<>	Signed not equal to
\	Not
^ or |	Bitwise or
and or .and.	Bitwise AND
eq or .eq.	Equality test
gt or .gt.	Signed greater than
high or .high.	High byte of operand
low or .low.	Low byte of operand
lt or .lt.	Signed less than
mod or .mod.	Modulus
not or .not.	Bitwise compliment
or or .or.	Bitwise or
shl or .shl.	Shift left
shr or .shr.	Shift right
ult or .ult.	Unsigned less than
ugt or .ugt.	Unsigned greater than
xor or .xor.	Exclusive or
seg	Segment (bank number) of address

The operators listed may all be freely combined in both constant and relocatable expressions. The HI-TECH linker permits relocation of complex expressions, so the results of expressions involving relocatable identifiers may not be resolved until link time.

6.3.8 Statement format

Legal statement formats are shown in table Table 6 - 4 on page 162. The second form is only legal with

Table 6 - 4 ZAS statement formats

label:	opcode	operands	;comment
name	pseudo-op	operands	;comment
;comment only			

certain directives, such as MACRO, SET and EQU. The *label* field is optional and if present should contain one identifier. The *name* field is mandatory and should also contain one identifier. Note that a label, if present, is followed by a colon. There is no limitation on what column or part of the line any part of the statement should appear in.

6.3.9 Program sections

Program sections, or *psects*, are a way of grouping together parts of a program even though the source code may not be physically adjacent in the source file, or even where spread over several source files. Unless defined as ABS (absolute), psects are relocatable.

A psect is identified by a name and has several attributes. The psect directive is used to define psects. It takes as arguments a name and an optional comma-separated list of flags. See the section PSECT on page 165 for full information. The assembler associates no significance to the name of a psect.

The following is an example showing some executable instructions being placed in the *text* psect, and some data bytes being placed in the *data* psect.

```
      psect text, global
alabel:
      ld    hl,astring
      call  putit
      ld    hl,anotherstring
      psect data, global
astring:
      defm  ’A string of chars’
      defb  0
anotherstring:
      defm  ’Another string’
```

```
        defb  0
        psect text
putit:
        ld    a,(hl)
        or    a
        ret   z
        call  outchar
        inc   hl
        jr    putit
```

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 `ld hl,anotherstring` 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, that is, 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, that is their address will be determined by the assembler.

Relocatable expressions may be combined freely in expressions.

6.3.10 Extended Condition Codes

The assembler recognises several additional condition codes as listed in Table 6 - 5 on page 163.

Table 6 - 5 Extended condition codes

Code	Meaning	Equivalent
alt	Arithmetic less-than	m
llt	Logical less-than	c
age	Arithmetic greater-than or equal-to	p
lge	Logical greater-than or equal-to	nc
di,ei	Use after **`ld a,i`** for testing state of interrupt enable flag - enabled or disabled respectively.	

6.3.11 Assembler directives

Assembler directives, or *pseudo-ops*, are used in a similar way to opcodes, but either do not generate code, or generate non-executable code, i.e. data bytes. The directives are listed in table Table 6 - 6 on page 164, and detailed below**.**

Table 6 - 6 ZAS directives (pseudo-ops)

Directive	Purpose
GLOBAL	Make symbols accessible to other modules or allow reference to other modules' symbols
END	End assembly
PSECT	Declare or resume program section
ORG	Set location counter
EQU	Define symbol value
DEFL	Define or re-define symbol value
DEFB, DB	Define constant byte(s)
DEFW	Define constant word(s)
DEFF	Define constant real(s)
DEFM	Define message
DEFS, DS	Reserve storage
IF, COND	Conditional assembly
ELSE	Alternate conditional assembly
ENDC	End conditional assembly
MACRO	Macro definition
ENDM	End macro definition
LOCAL	Define local tabs
REPT	Repeat a block of code n times
IRP	Repeat a block of code with a list
IRPC	Repeat a block of code with a character list
SIGNAT	Define function signature

6.3.11.1 GLOBAL

GLOBAL declares a list of symbols which, if defined within the current module, are made public, otherwise are references to symbols in external modules. Example:

```
GLOBAL    lab1,lab2,lab3
```

6.3.11.2 END

`END` is optional, but if present should be at the very end of the program. It will terminate the assembly. If an expression is supplied as an argument, that expression will be used to define the start address of the program. Whether this is of any use will depend on the linker. Example:

```
END   start_label
```

6.3.11.3 PSECT

The PSECT directive declares or resumes a program section. It takes as arguments a name and optionally a comma separated list of flags. The allowed flags are listed in Table 6 - 7 on page 165 and detailed below. Once a psect has been declared it may be resumed later by simply giving its name as an argument to another psect directive; the flags need not be repeated.

Table 6 - 7 PSECT flags

Flag	Meaning
ABS	Psect is absolute
CLASS	Specify class name for psect
GLOBAL	Psect is global (default)
LOCAL	Psect is not global
OVRLD	Psect will overlap same psect in other modules
PURE	Psect is to be read-only
RELOC	Start psect on specified boundary
SELECTOR	Page number for banked areas
SIZE	Maximum size of psect

❒ `ABS` defines the current psect as being absolute, i.e. it is to start at location 0. This does not mean that this module's contribution to the psect will start at 0, since other modules may contribute to the same psect.

❒ The `CLASS` flag specifies a class name for this psect. Class names are used to allow local psects to be referred to by a class name at link time, since they cannot be referred to by their own name. Class names are also useful where the linker address range feature is to be used.

❒ A psect defined as `GLOBAL` will be combined with other global psects of the same name from other modules at link time. `GLOBAL` is the default.

❒ A psect defined as `LOCAL` will not be combined with other local psects at link time, even if there are others with the same name. A local psect may not have the same name as any global psect, even one in another module.

❒ A psect defined as `OVRLD` will have the contribution from each module overlaid, rather than concatenated at run time. `OVRLD` in combination with `ABS` defines a truly absolute psect, i.e. a psect within which any symbols defined are absolute.

❒ The `PURE` flag instructs the linker that this psect will not be modified at run time and may therefore, for example, be placed in ROM. This flag is of limited usefulness since it depends on the linker and target system enforcing it.

- ❐ The `RELOC` flag allows specification of a requirement for alignment of the psect on a particular boundary, e.g. `RELOC=100h` would specify that this psect must start on an address that is a multiple of 100h.
- ❐ The `SELECTOR` flag is calculated by the linker from the load address of a psect, either automatically by a default rule, or by an explicit rule passed with a `-g` option to the linker.
- ❐ The `SIZE` flag allows a maximum size to be specified for the psect, e.g. SIZE=100h. This will be checked by the linker after psects have been combined from all modules.

Some examples of the use of the PSECT directive follow:

```
PSECT fred
PSECT bill,size=100h,global
PSECT joh,abs,ovrld,class=CODE
```

6.3.11.4 ORG

`ORG` changes the value of the location counter within the current psect. This means that the addresses set with `ORG` are relative to the base of the psect, which is not determined until link time.

The argument to `ORG` must be either an absolute value, or a value referencing the current psect. In either case the current location counter is set to the value of the argument. For example:

```
ORG   100h
```

In order to use the `ORG` directive to set the location counter to an absolute value, an absolute, overlaid psect must be used:

```
psect absdata, abs, ovrld
org   addr
```

where *addr* is an absolute address.

6.3.11.5 EQU

This pseudo-op defines a symbol and equates its value to an expression. For example

```
assemblyEQU 123h
```

The identifier `assembly` will be given the value 123h. `EQU` is legal only when the symbol has not previously been defined. The operand may also be a register name, in which case the symbol will become a synonym for the register.

6.3.11.6 DEFL

`DEFL` (define label) is identical to `EQU` except that it may be used to re-define a symbol.

6.3.11.7 DEFB, DB

DEFB and DB are identical and are used to initialize storage as bytes. The argument is a list of expressions, each of which will be assembled into one byte. DEFB and DB may also take a multi-character string as an argument. Each character of the string will be assembled into one memory location.

An error will occur if the value of an expression is too big to fit into the memory location, e.g. if the value 1020 is given as an argument to DB.

Examples:

```
      alabel:DB   ’X’,1,2,3,4,"A string",0
```

6.3.11.8 DEFW

DEFW operates in a similar fashion to DEFB, except that it assembles expressions into words. An error will occur if the value of an expression is too big to fit into a word.

Example:

```
      DEFW  -1, 3664H, ‘A’, 3777Q
```

6.3.11.9 DEFF

DEFF initializes 4 bytes of memory as real numbers. Each number will be stored in 32 bit HI-TECH C format. For example:

```
      pi:   DEFF  3.14159
            DEFF  3.3,3e10,-23
```

6.3.11.10 DEFS, DS

This directive reserves, but does not initialize, memory locations. The single argument is the number of bytes to be reserved. Examples:

```
      alabel:DEFS 23    ;Reserve 23 bytes of memory
      xlabel:DEFS 2+3   ;Reserve 5 bytes of memory
```

6.3.11.11 IF, COND, ELSE and ENDC

These directives implement conditional assembly. IF and COND are identical. The argument to IF should be an absolute expression. If it is non-zero, then the code following it up to the next matching ELSE will be assembled. If the expression is zero then the code up to the next matching ELSE will be skipped.

At an ELSE the sense of the conditional compilation will be inverted, while an ENDC will terminate the conditional assembly block. Example:

```
IF    CPM
call  5
ELSE
call  os_func
ENDC
```

In this example, if *CPM* is non-zero, the first *call* instruction will be assembled but not the second. Conversely if *CPM* is zero, the second *call* will be assembled but the first will not. Conditional assembly blocks may be nested.

6.3.11.12 MACRO and ENDM

These directives provide for the definition of macros. The MACRO directive should be preceded by the macro name and optionally 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:

```
print MACRO string
      psect data
999:  db    string,’$’
      psect text
      ld    de,999b
      ld    c,9
      call  5
      ENDM
```

When used, this macro will expand to the 3 instructions in the body of the macro, with the formal parameters substituted by the arguments. Thus:

```
print ’hello world’
```

expands to:

```
      psect data
999:  db    ‘hello world','$'
      psect text
      ld    de,999b
      ld    c,9
      call  5
```

The & character may be used to delimit an argument used in the coding of the macro, thus permitting the concatenation of macro parameters with other text, but is removed in the actual macro expansion.

The & character need not be used if commas and spaces delimit the argument, but the & character should be used at the start and end if no other delimiter is available.

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 (`;;`).

6.3.11.13 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:

```
copy  MACRO source,dest,count
      LOCAL nocopy
      push af
      push bc
      ld    bc,source
      ld    a,b
      or    c
      jr    z,nocopy
      push de
      push hl
      ld    de,dest
      ld    hl,source
      ldir
      pop   hl
      pop   de
nocopy:pop  bc
      pop   af
      ENDM
```

when expanded will include a unique assembler generated label in place of *nocopy*. For example, `copy (recptr),buf,(recsize)` will expand to:

```
      push af
      push bc
      ld    bc,(recsize)
      ld    a,b
      or    c
      jr    z,??0001
      push de
```

```
            push hl
            ld    de,buf
            ld    hl,(recptr)
            ldir
            pop   hl
            pop   de
      ??0001:pop  bc
                  pop   af
```

if invoked a second time, the label *nocopy* would expand to *??0002*.

6.3.11.14 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
            ld    (hl),0
            inc   hl
            ENDM
```

will expand to

```
            ld    (hl),0
            inc   hl
            ld    (hl),0
            inc   hl
            ld    (hl),0
            inc   hl
```

6.3.11.15 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   string,<'hello world',13,10>,'arg2'
            LOCAL str
            psect data
      str:  db    string,'$'
            psect text
            ld    c,9
            ld    de,str
            call  5
            ENDM
```

would expand to

```
            psect data
      ??0001:db   ‘hello world',13,10,'$'
            psect text
            ld    c,9
            ld    de,??0001
            call  5
            psect data
      ??0002:db   ‘arg2','$'
            psect text
            ld    c,9
            ld    de,??0002
            call  5
```

Note the use of LOCAL labels and angle brackets in the same manner as with conventional macros.

The `IRPC` directive is similar, except it substitutes one character at a time from a string of non-space characters. For example:

```
            IRPC  char,ABC
            ld    c,2
            ld    e,'char'
            call  5
            ENDM
```

will expand to:

```
            ld    c,2
            ld    e,'A'
            call  5
            ld    c,2
            ld    e,'B'
            call  5
            ld    c,2
            ld    e,'C'
            call  5
```

6.3.11.16 SIGNAT

This directive is used to associate a 16 bit signature value with a label. At link time the linker checks that all signatures defined for a particular label are the same and produces an error if they are not. The `SIGNAT` directive is used by the HI-TECH C compiler to enforce link time checking of function prototypes and calling conventions.

Use the `SIGNAT` directive if you want to write assembly language routines which are called from C. For example:

```
        SIGNAT_fred,8192
```

will associate the signature value 8192 with symbol `_fred`. If a different signature value for `_fred` is present in any object file, the linker will report an error.

6.3.12 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 a space) in an argument then angle brackets (`<` and `>`) may be used to quote the 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 normally a comment indicator.

If an argument is preceded by a percent sign (`%`), that argument will be evaluated as an 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.

6.3.13 Assembler controls

Assembler controls may be included in the assembler source to control such things as listing format. Each assembler control starts with the asterisk (`*`) character. These keywords have no significance anywhere else in the program. Some keywords are followed by a parameter.

A list of keywords is given in Table 6 - 8 on page 172, and each is described further below.

Table 6 - 8 ZAS assembler controls

Control name	Meaning
*EJECT	Start a new page in the listing
*HEADING *string*	Define the subtitle for the listing
*INCLUDE *file*	Textually include another source file
*LIST *on|off*	Turn the listing on or off
*TITLE *string*	Define the title for the listing

6.3.13.1 *EJECT

**EJECT* causes a new page to be started in the listing. A control`-L` (form feed) character will also cause a new page when encountered in the source.

6.3.13.2 *HEADING *string*

The **HEADING* assembler control defines a subtitle for the listing. The subtitle appears at the top of each page, but can be changed from time to time. Here is an example:

```
*heading Initialisation Phase.
```

6.3.13.3 *INCLUDE *file*

This assembler control can be used to textually includes another source file. For example:

```
*include frame.inc
```

will include the file *frame.inc* at that point when assembling.

6.3.13.4 *LIST *on|off*

This directive turns listing on and off. With listing off, none of the generated code or the source code that produces it is listed in the listing file. Examples:

```
*List    on
*List    off
```

6.3.13.5 *TITLE *string*

This control keyword defines a title which appears at the top of every listing page. The title can be set only once in a file. For example:

```
      *title Heater Control Program
```

Linker and Utilities Reference Manual

7.1 Introduction

HI-TECH 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 source file may be compiled separately and finally all the object files linked together into a single executable program.

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 (HPD or command line) will automatically invoke the linker with 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 HI-TECH C is generic to a wide variety of compilers for several different processors. Not all features described in this chapter are applicable to all compilers.

7.2 Relocation and Psects

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

7.3 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 initialised data, and *bss* psects, containing uninitialised but reserved data.

The difference between the data and bss psects may be illustrated by considering two external variables; one is initialised to the value 1, and the other is not initialised. 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 initialised at run time to zero. The first will however occupy space in the program file, and will maintain its initialised 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.

7.4 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.

7.5 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.

7.6 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 initialised 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.

7.7 Operation

A command to the linker takes the following form:

```
    hlink1 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 recognised by the linker are listed in Table 7 - 1 on page 177 and discussed in the following paragraphs.

Table 7 - 1 Linker Options

Option	Effect
-A*class=low-high,...*	Specify address ranges for a class
-C*x*	Call graph options
-C*psect=class*	Specify a class name for a global psect
-C*baseaddr*	Produce binary output file based at *baseaddr*
-D*class=delta*	Specify a class delta value
-D*symfile*	Produce old-style symbol file
-E*errfile*	Write error messages to *errfile*
-F	Produce .OBJ file with only symbol records
-G*spec*	Specify calculation for segment selectors
-H*symfile*	Generate symbol file
-H+*symfile*	Generate enhanced symbol file
-I	Ignore undefined symbols
-J*num*	Set maximum number of errors before aborting
-K	Prevent overlaying function parameter and auto areas
-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
-Nc	Sort symbol table in map file by class address order
-Ns	Sort symbol table in map file by space 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
-Q*processor*	Specify the processor type (for cosmetic reasons only)

1. In earlier versions of HI-TECH C the linker was called LINK.EXE

Table 7 - 1 Linker Options

Option	Effect
-S	Inhibit listing of symbols in symbol file
-S*class=limit[,bound]*	Specify address limit, and start boundary for a class of psects
-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

7.7.1 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.

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

Normally psects are linked according to the information given to a -P option (see below) but sometimes it is desired to have a class of psects 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
```

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.

Where there are a number of identical, contiguous address ranges, they may be specified with a repeat count, e.g.

```
        -ACODE=0-FFFFhx16
```

specifies that there are 16 contiguous ranges each 64k bytes in size, starting from zero. Even though the ranges are contiguous, no code will straddle a 64k boundary. The repeat count is specified as the character ‘x’ or ‘*’ after a range, followed by a count.

7.7.3 -C*x*

These options allow control over the call graph information which may be included in the map file produced by the linker. The -CN option removes the call graph information from the map file. The -CC option only include the critical paths of the call graph. A function call that is marked with a ‘*’ in a full call graph is on a critical path and only these calls are included when the -CC option is used. A call graph is only produced for processors and memory models that use a compiled stack.

7.7.4 -C*psect=class*

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

7.7.5 -D*class=delta*

This option allows the delta value for psects that are members of the specified class to be defined. The delta value should be a number and represents the number of bytes per addressable unit of objects within the psects. Most psects do not need this option as they are defined with a delta value.

7.7.6 -D*symfile*

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

7.7.7 -E*errfile*

Error messages from the linker are written to standard error (file handle 2). Under DOS there is no convenient way to redirect this to a file (the compiler drivers will redirect standard error if standard output is redirected). This option will make the linker write all error messages to the specified file instead of the screen, which is the default standard error destination.

7.7.8 -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.

This option can be used when producing more than one hex file for situations where the program is contained in different memory devices located at different addresses. The files for one device are compiled using this linker option to produce a symbol-only object file; this is then linked with the files for the other device. The process can then be repeated for the other files and device.

7.7.9 -G*spec*

When linking programs using segmented, or bank-switched 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.

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 bank-switched 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*, * 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.

7.7.10 -H*symfile*

This option will instruct the linker to generate a symbol file. The optional argument *symfile* specifies a file to receive the symbol file. The default file name is *l.sym*.

7.7.11 -H+*symfile*

This option will instruct the linker to generate an *enhanced* symbol file, which provides, in addition to the standard symbol file, class names associated with each symbol and a segments section which lists each class name and the range of memory it occupies. This format is recommended if the code is to be run in conjunction with a debugger. The optional argument *symfile* specifies a file to receive the symbol file. The default file name is *l.sym*.

7.7.12 -J*errcount*

The linker will stop processing object files after a certain number of errors (other than warnings). The default number is 10, but the -J option allows this to be altered.

7.7.13 -K

For compilers that use a compiled stack, the linker will try and overlay function auto and paramter areas in an attempt to reduce the total amount of RAM required. For debugging purposes, this feature can be disabled with this option.

7.7.14 -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.

7.7.15 -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 running 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.

7.7.16 -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.

7.7.17 -M*mapfile*

This option 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 in Section 7.9 on page 184.

7.7.18 -N, -Ns and-Nc

By default the symbol table in the link map will be sorted by name. The -N option will cause it to be sorted numerically, based on the value of the symbol. The -Ns and -Nc options work similarly except that the symbols are grouped by either their Space type, or Class type.

7.7.19 -O*outfile*

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

7.7.20 -P*spec*

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

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

There are several variations, but essentially each psect is listed with its desired link and load addresses, and a minimum value. All values may be omitted, in which case a default will apply, depending on previous values.

The minimum value (*min*) is preceded by a + sign, if present. It sets a minimum value for the link or load address. The address will be calculated as described below, but if it is less than the minimum then it will be set equal to the minimum.

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.

Any psects allocated by a -P option will have their load address range subtracted from any address ranges specified with the -A option. This allows a range to be specified with the -A option without knowing in advance how much of the lower part of the range, for example, will be required for other psects.

7.7.21 -Q*processor*

This option allows a processor type to be specified. This is purely for information placed in the map file. The argument to this option is a string describing the processor.

7.7.22 -S

This option prevents symbol information relating from being included in the symbol file produced by the linker. Segment information is still included.

7.7.23 -S*class=limit[, bound]*

A class of psects may have an upper address *limit* associated with it. The following example places a limit on the maximum address of the CODE class of psects to one less than 400h.

```
-SCODE=400h
```

Note that to set an upper limit to a psect, this must be set in assembler code (with a *LIMIT=* flag on a psect directive).

If the *bound* (boundary) argument is used, the class of psects will start on a multiple of the bound address. This example places the FARCODE class of psects at a multiple of 1000h, but with an upper address limit of 6000h:

```
-SFARCODE=6000h,1000h
```

7.7.24 -U*symbol*

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

7.7.25 -V*avmap*

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

7.7.26 -W*num*

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

-*W9* will suppress all warning messages. -*W0* is the default. Setting the warning level to -9 (-*W-9*) will give the most comprehensive warning messages.

7.7.27 -X

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

7.7.28 -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.

7.8 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 HLINK 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, HLINK 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
```

7.9 Map Files

The map file contains information relating to the relocation of psects and the addresses assigned to symbols within those psects. 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. This is optionally followed by call graph information, depended on the processor and memory model selected. 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 summarises the psects from the object files. The SEGMENTS section summarises 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.

```
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
```

7.9.1 Call Graph Information

A call graph is produced for chip types and memory models that use a compiled stack, rather than a hardware stack, to facilitate parameter passing between functions and auto variables defined within a function. When a compiled stack is used, functions are not re-entrant since the function will use a fixed area of memory for its local objects (parameters/auto variables). A function called **foo**, for example, will use symbols like **?_foo** for parameters and **?a_foo** for auto variables. Compilers such as the PIC, 6805 and V8 use compiled stacks. The 8051 compiler uses a compiled stack in small and medium memory models. The call graph shows information relating to the placement of function parameters and auto variables by the linker. A typical call graph may look something like:

```
Call graph:

*_main size 0,0 offset 0
      _init size 2,3 offset 0
            _ports size 2,2 offset 5
*     _sprintf size 5,10 offset 0
*           _putch
      INDIRECT 4194
            INDIRECT 4194
                  _function_2 size 2,2 offset 0
                  _function size 2,2 offset 5
*_isr->_incr size 2,0 offset 15
```

The graph shows the functions called and the memory usage (RAM) of the functions for their own local objects. In the example above, the symbol **_main** is associated with the function **main**. It is shown at the far left of the call graph. This indicates that it is the root of a call tree. The run-time code has the **FNROOT** assembler directive that specifies this. The size field after the name indicates the number of parameters and auto variables, respectively. Here, **main()** takes no parameters and defines no auto variables. The offset field is the offset at which the function's parameters and auto variables have been placed from the beginning of the area of memory used for this purpose. The run-time code contains a **FNCONF** directive which tells the compiler in which psect parameters and auto variables should reside. This memory will be shown in the map file under the name **COMMON**.

Main() calls a function called **init**. This function uses a total of two bytes of parameters (it may be two chars or one int; that is not important) and has three bytes of auto variables. These figures are the total of bytes of *memory* consumed by the function. If the function was passed a two-byte int, but that was done via a register, then the two bytes would not be included in this total. Since **main()** did not use any of the local object memory, the offset of **init()**'s memory is still at 0.

The function **init** itself calls another function called **ports**. This function uses two bytes of parameters and another two bytes of auto variables. Since **ports()** is called by **init()**, its local variables cannot be overlapped with those of **init()**'s, so the offset is 5, which means that **ports()**'s local objects were placed immediately after those of **init()**'s.

The function **`main`** also calls s**`printf()`**. Since the function **`sprintf`** is not active at the same time as **`init()`** or **`ports()`**, their local objects can be overlapped and the offset is hence set to 0. **`Sprintf()`** calls a function **`putch`**, but this function uses no memory for parameters (the char passed as argument is apparently done so via a register) or locals, so the size and offset are zero and are not printed.

`Main()` also calls another function indirectly using a function pointer. This is indicated by the two **`INDIRECT`** entries in the graph. The number following is the signature value of functions that could potentially be called by the indirect call. This number is calculated from the parameters and return type of the functions the pointer can indirectly call. The names of any functions that have this signature value are listed underneath the **`INDIRECT`** entries. Their inclusion does not mean that they were called (there is no way to determine that), but that they could potentially be called.

The last line shows another function whose name is at the far left of the call graph. This implies that this is the root of another call graph tree. This is an interrupt function which is not called by any code, but which is automatically invoked when an enabled interrupt occurs. This interrupt routine calls the function **`incr()`**, which is shown shorthand in the graph by the "**`->`**" symbol followed by the called function's name instead of having that function shown indented on the following line. This is done whenever the calling function does not takes parameters, nor defines any variables.

Those lines in the graph which are starred (**`*`**) are those functions which are on a critical path in terms of RAM usage. For example, in the above, (**`main()`** is a trivial example) consider the function **`sprintf`**. This uses a large amount of local memory and if you could somehow rewrite it so that it used less local memory, it would reduce the entire program's RAM usage. The functions **`init`** and **`ports`** have had their local memory overlapped with that of **`sprintf()`**, so reducing the size of these functions' local memory will have no affect on the program's RAM usage. Their memory usage could be increased, as long as the total size of the memory used by these two functions did not exceed that of **`sprintf()`**, with no additional memory used by the program. So if you have to reduce the amount of RAM used by the program, look at those functions that are starred.

If, when searching a call graph, you notice that a function's parameter and auto areas have been overlapped (i.e. **`?a_foo`** was placed at the same address as **`?_foo`**, for example), then check to make sure that you have actually called the function in your program. If the linker has not seen a function actually called, then it overlaps these areas of memory since that are not needed. This is a consequence of the linker's ability to overlap the local memory areas of functions which are not active at the same time. Once the function is called, unique addresses will be assigned to both the parameters and auto objects.

If you are writing a routine that calls C code from assembler, you will need to include the appropriate assembler directives to ensure that the linker sees the C function being called.

7.10 Librarian

The librarian program, LIBR, has the function of combining several object files into a single file known as a library. The purposes of combining several such object modules are several.

- ❒ fewer files to link
- ❒ faster access
- ❒ uses less disk space

In order to make the library concept useful, it is necessary for the linker to treat modules in a library differently from object files. If an object file is specified to the linker, it will be linked into the final linked module. A module in a library, however, will only be linked in if it defines one or more symbols previously known, but not defined, to the linker. Thus modules in a library will be linked only if required. Since the choice of modules to link is made on the first pass of the linker, and the library is searched in a linear fashion, it is possible to order the modules in a library to produce special effects when linking. More will be said about this later.

7.10.1 The Library Format

The modules in a library are basically just concatenated, but at the beginning of a library is maintained a directory of the modules and symbols in the library. Since this directory is smaller than the sum of the modules, the linker is speeded up when searching a library since it need read only the directory and not all the modules on the first pass. On the second pass it need read only those modules which are required, seeking over the others. This all minimises disk I/O when linking.

It should be noted that the library format is geared exclusively toward object modules, and is not a general purpose archiving mechanism as is used by some other compiler systems. This has the advantage that the format may be optimized toward speeding up the linkage process.

7.10.2 Using the Librarian

The librarian program is called LIBR, and the format of commands to it is as follows:

```
libr options k file.lib file.obj ...
```

Interpreting this, LIBR is the name of the program, *options* is zero or more librarian options which affect the output of the program. *k* is a key letter denoting the function requested of the librarian (replacing, extracting or deleting modules, listing modules or symbols), *file.lib* is the name of the library file to be operated on, and *file.obj* is zero or more object file names.

The librarian options are listed in Table 7 - 2.

The key letters are listed inTable 7 - 3.

Table 7 - 2 Librarian Options

Option	Effect
-P*width*	specify page width
-W	suppress non-fatal errors

Table 7 - 3 Librarian Key Letter Commands

Key	Meaning
r	Replace modules
d	Delete modules
x	Extract modules
m	List modules
s	List modules with symbols

When replacing or extracting modules, the *file.obj* arguments are the names of the modules to be replaced or extracted. If no such arguments are supplied, all the modules in the library will be replaced or extracted respectively. Adding a file to a library is performed by requesting the librarian to replace it in the library. Since it is not present, the module will be appended to the library. If the *r* key is used and the library does not exist, it will be created.

Under the *d* key letter, the named object files will be deleted from the library. In this instance, it is an error not to give any object file names.

The *m* and *s* key letters will list the named modules and, in the case of the *s* keyletter, the symbols defined or referenced within (global symbols only are handled by the librarian). As with the *r* and *x* key letters, an empty list of modules means all the modules in the library.

7.10.3 Examples

Here are some examples of usage of the librarian. The following lists the global symbols in the modules a.obj, b.obj and c.obj:

```
LIBR s file.lib a.obj b.obj c.obj
```

This command deletes the object modules a.obj, b.obj and 2.obj from the library file.lib:

```
LIBR d file.lib a.obj b.obj 2.obj
```

7.10.4 Supplying Arguments

Since it is often necessary to supply many object file arguments to LIBR, and command lines are restricted to 127 characters by CP/M and MS-DOS, LIBR will accept commands from standard input if no command line arguments are given. If the standard input is attached to the console, LIBR will prompt for input. Multiple line input may be given by using a backslash as a continuation character on the end of a line. If standard input is redirected from a file, LIBR will take input from the file, without prompting. For example:

```
libr
libr> r file.lib 1.obj 2.obj 3.obj \
libr> 4.obj 5.obj 6.obj
```

will perform much the same as if the .obj files had been typed on the command line. The libr> prompts were printed by LIBR itself, the remainder of the text was typed as input.

```
libr <lib.cmd
```

Libr will read input from lib.cmd, and execute the command found therein. This allows a virtually unlimited length command to be given to LIBR.

7.10.5 Listing Format

A request to LIBR to list module names will simply produce a list of names, one per line, on standard output. The s keyletter will produce the same, with a list of symbols after each module name. Each symbol will be preceded by the letter D or U, representing a definition or reference to the symbol respectively. The -P option may be used to determine the width of the paper for this operation. For example LIBR -P80 s file.lib will list all modules in file.lib with their global symbols, with the output formatted for an 80 column printer or display.

7.10.6 Ordering of Libraries

The librarian creates libraries with the modules in the order in which they were given on the command line. When updating a library the order of the modules is preserved. Any new modules added to a library after it has been created will be appended to the end.

The ordering of the modules in a library is significant to the linker. If a library contains a module which references a symbol defined in another module in the same library, the module defining the symbol should come after the module referencing the symbol.

7.10.7 Error Messages

LIBR issues various error messages, most of which represent a fatal error, while some represent a harmless occurrence which will nonetheless be reported unless the -W option was used. In this case all warning messages will be suppressed.

7.11 Objtohex

The HI-TECH linker is capable of producing simple binary files, or object files as output. Any other format required must be produced by running the utility program OBJTOHEX. This allows conversion of object files as produced by the linker into a variety of different formats, including various hex formats. The program is invoked thus:

```
objtohex options inputfile outputfile
```

All of the arguments are optional. If *outputfile* is omitted it defaults to *l.hex* or *l.bin* depending on whether the -b option is used. The *inputfile* defaults to *l.obj*.

The options for OBJTOHEX are listed in Table 7 - 4 on page 191. Where an address is required, the format is the same as for HLINK:.

Table 7 - 4 Objtohex Options

Option	Meaning
-A	Produce an ATDOS .ATX output file
-B*base*	Produce a binary file with offset of *base*. Default file name is *l.obj*
-C*ckfile*	Read a list of checksum specifications from *ckfile* or standard input
-D	Produce a .COD file
-E	Produce an MS-DOS .EXE file
-F*fill*	Fill unused memory with bytes of value *fill* - default value is 0FFh
-I	Produce an Intel HEX file with linear addressed extended records.
-L	Pass relocation information into the output file (used with .EXE files)
-M	Produce a Motorola HEX file (S19, S28 or S37 format)
-N	Produce an output file for Minix
-P*stk*	Produce an output file for an Atari ST, with optional stack size
-R	Include relocation information in the output file
-S*file*	Write a symbol file into *file*
-T	Produce a Tektronix HEX file. -TE produces an extended TekHEX file.
-U	Produce a COFF output file
-UB	Produce a UBROF format file
-V	Reverse the order of words and long words in the output file
-x	Create an x.out format file

7.11.1 Checksum Specifications

The checksum specification allows automated checksum calculation. The checksum specification takes the form of several lines, each line describing one checksum. The syntax of a checksum line is:

```
addr1-addr2 where1-where2 +offset
```

All of *addr1*, *addr2*, *where1*, *where2* and *offset* are hex numbers, without the usual H suffix. Such a specification says that the bytes at addr1 through to addr2 inclusive should be summed and the sum placed in the locations where1 through where2 inclusive. For an 8 bit checksum these two addresses should be the same. For a checksum stored low byte first, where1 should be less than where2, and vice versa. The +offset is optional, but if supplied, the value offset will be used to initialise the checksum. Otherwise it is initialised to zero. For example:

```
0005-1FFF 3-4 +1FFF
```

This will sum the bytes in 5 through 1FFFH inclusive, then add 1FFFH to the sum. The 16 bit checksum will be placed in locations 3 and 4, low byte in 3. The checksum is initialised with 1FFFH to provide protection against an all zero rom, or a rom misplaced in memory. A run time check of this checksum would add the last address of the rom being checksummed into the checksum. For the rom in question, this should be 1FFFH. The initialization value may, however, be used in any desired fashion.

7.12 Cref

The cross reference list utility CREF is used to format raw cross-reference information produced by the compiler or the assembler into a sorted listing. A raw cross-reference file is produced with the -CR option to the compiler. The assembler will generate a raw cross-reference file with a -C option (most assemblers) or by using an OPT CRE directive (6800 series assemblers) or a XREF control line (PIC assembler). The general form of the CREF command is:

```
cref options files
```

where *options* is zero or more options as described below and *files* is one or more raw cross-reference files. CREF takes the options listed in Table 7 - 5 on page 193. Each option is described in more detail in the following paragraphs.

7.12.1 -F*prefix*

It is often desired to exclude from the cross-reference listing any symbols defined in a system header file, e.g. <stdio.h>. The -F option allows specification of a path name prefix that will be used to exclude any symbols defined in a file whose path name begins with that prefix. For example, **-F\** will exclude any symbols from all files with a path name starting with \\.

Table 7 - 5 Cref Options

Option	Meaning
-F*prefix*	Exclude symbols from files with a pathname or filename starting with *prefix*
-H*heading*	Specify a heading for the listing file
-L*len*	Specify the page length for the listing file
-O*outfile*	Specify the name of the listing file
-P*width*	Set the listing width
-S*stoplist*	Read file *stoplist* and ignore any symbols listed.
-X*prefix*	Exclude any symbols starting with the given *prefix*

7.12.2 -H*heading*

The -H option takes a string as an argument which will be used as a header in the listing. The default heading is the name of the first raw cross-ref information file specified.

7.12.3 -L*len*

Specify the length of the paper on which the listing is to be produced, e.g. if the listing is to be printed on 55 line paper you would use a -L*55* option. The default is 66 lines.

7.12.4 -O*outfile*

Allows specification of the output file name. By default the listing will be written to the standard output and may be redirected in the usual manner. Alternatively *outfile* may be specified as the output file name.

7.12.5 -P*width*

This option allows the specification of the width to which the listing is to be formatted, e.g. -P*132* will format the listing for a 132 column printer. The default is 80 columns.

7.12.6 -S*stoplist*

The -S option should have as its argument the name of a file containing a list of symbols not to be listed in the cross-reference. Multiple stoplists may be supplied with multiple -S options.

7.12.7 -X*prefix*

The -X option allows the exclusion of symbols from the listing, based on a prefix given as argument to -X. For example if it was desired to exclude all symbols starting with the character sequence *xyz* then the option -X*xyz* would be used. If a digit appears in the character sequence then this will match any digit in the symbol, e.g. -X**X0** would exclude any symbols starting with the letter *X* followed by a digit.

CREF will accept wildcard filenames and I/O redirection. Long command lines may be supplied by invoking CREF with no arguments and typing the command line in response to the *cref*> prompt. A backslash at the end of the line will be interpreted to mean that more command lines follow.

7.13 Memmap

MEMMAP has been individualized for each processor. The MEMMAP program that appears in your \bin directory will conform with the following criteria; *XX*map.exe where *XX* stands for the processor type. From here on, we will be referring to *XX*map.exe as MEMMAP, as to cover all processors.

At the end of compilation and linking, HPD and the command line compiler produce a summary of memory usage. If, however, the compilation is performed in separate stages and the linker is invoked explicitly, this memory information is not displayed. The MEMMAP program reads the information stored in the map file and produces either a summary of psect address allocation or a memory map of program sections similar to that shown by HPD and the command line compiler.

7.13.1 Using MEMMAP

A command to the memory usage program takes the form:

```
memmap options file
```

Options is zero or more MEMMAP options which are listed in Table 7 - 6 on page 194. *File* is the name

Table 7 - 6 Memmap options

Option	Effect
-P	Print psect usage map
-W*wid*	Specifies width to which address are printed

of a map file. Only one map file can be processed by MEMMAP.

7.13.1.1 -P

The default behaviour of MEMMAP is to produce a segment memory map. This output is similar to that printed by HPD and the command line compiler after compilation and linking. This behaviour can be changed by using the -P option. This forces a psect usage map to be printed. The output in this case will be similar to that shown by the HPD's **Memory Usage Map** item under the **Utility** menu or if the -PSECTMAP option is used with the command line compiler.

7.13.1.2 -W*wid*

The width to which addresses are printed can be adjusted by using the -W option. The default width is determined in respect to the processor's address range. Depending on the type of processor used, determines the default width of the printed address, for example a processor with less than or equal to 64k will have a default width of 4. Whereas a processor with greater than 64k may have a default value of 6 digits.

Lucifer Source Level Debugger

Lucifer is a source level remote debugger for use with the HI-TECH C compilers. It consists of a program which runs on a host machine (usually MS-DOS or UNIX) and communicates with a Z80-, Z180-, 64180- or NSC800-based microcontroller system via a serial line.

The host program provides the user interface, including source code display, disassembly, displaying memory etc. The target system must have logic to read and write memory and registers, and implement single stepping. With each version of Lucifer a small program is provided which can be compiled and placed in a ROM in a target system to implement these features. The standard host program is set up to communicate with this ROM program via a serial line.

8.1 Using Lucifer

To use Lucifer you will need to have the compiler generate a symbol file, with symbol name, line number and file name symbols included. This can be produced using the ZC -G option. If you use the -H option you will get a symbol file which can be used by Lucifer, but which does not contain any source code level information.

You can also use HPDZ to produce HEX and symbol files suitable for use with Lucifer. The LUCIFER directory contains *luctest.prj*, a sample project file which you can use as a guide to compiling your own programs with HPDZ. See the Using HPDZ chapter for more details. The basic option needed is **Options/Source level debug info**. Lucifer can be invoked from the **Run/Download ...** menu item. Otherwise once you have produced HEX and symbol files, invoke Lucifer as follows:

```
lucz80 -sSPEED -pCOMn test
```

If you are using an MS-DOS system, *COMn* should be COM1 or COM2. Lucifer will access a standard serial port addressed as either port. For UNIX simply specify the name of the serial port connected to your target, for example *-p/dev/tty006*. The default baud rate is 38400 for both MS-DOS and UNIX. The default serial port is *COM1* for MS-DOS and */dev/ttya* for UNIX. The -s (speed) and -p (port) options may be used to access ports other than the default. For UNIX */dev* may be left off the device name and will automatically added, thus *-ptty0* and *-p/dev/tty0* will access the same device. For example, under MS-DOS, **`lucz80 -s9600 -pCOM2`** will access COM2 at 9600 baud.

New default speed and port values may be set using the environment variable *LUCZ80_ARGS*. LUCZ80_ARGS may specify any mixture of valid Lucifer '**`-`**' options except filename options. For example, if you want the default options to be 4800 baud on port COM2, add the following line to your AUTOEXEC.BAT file:

```
SET LUCZ80_ARGS=-s4800 -pCOM2
```

In addition to the speed and port options Lucifer takes two optional arguments which are, in order of appearance, the name of the symbol file to use and the name of the .HEX or .BIN file to download. If no download file is specified, Lucifer will automatically search for .SYM, .HEX and .BIN files which the same base name as the symbol name given. Thus the command: **`lucz80 test`** would automatically locate and use *test.sym* and *test.hex* or *test.bin*. If you do not want to autoload your HEX or BIN file, give your symbol file a different base name to your HEX file.

When downloading binary files, Lucifer normally prompts for the download address. When downloading directly from the command line you can override this prompting by adding the option *-Baddr[:end]* to the LUCZ80 command line. For example, if you want to download a file *test.bin* at address $2000, you could use the command:

```
lucz80 -b2000 test
```

The optional *:end* value is the address at which the download should be terminated. For example, if you want to load the first $2000 bytes of `test.bin` from address $4000 to address $6000, use the command:

```
lucz80 -b4000:6000 test
```

Lucifer should announce itself, then attempt to communicate with the target. If successful it prints a message sent by the target, identifying itself, e.g:

```
Z80/64180 Lucifer v3.20, RST = CF
```

Lucifer will then display a prompt `:` and wait for commands. For a list of commands, type *?* and press return. Note that all commands should be in lower case.

8.2 Symbol names in expressions

Where Lucifer commands take numeric values or addresses as arguments; symbol names, register names and line numbers may be used. Symbols should be entered in exactly the same case as they were defined in the source code. Note that Lucifer cannot access auto variables or parameters by name. Where an expression is required, it may be of the forms in Table 8 - 1 on page 198.

Table 8 - 1 Lucifer expression forms

Expression form	Example
symbol_name	main
symbol+hexnum	barray+20
$hexnum	$2000
:linenumber	:10
regname	A5

By default, in the *b* (breakpoint) command any decimal number will be interpreted, as a line number. However, in the *u* (unassemble) command any number will be interpreted, by default, as a hex number representing an address. 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.

8.2.1 Auto Variables and Parameters

Auto variables and parameters cannot be accessed by name with Lucifer. To examine the contents of an auto variable or parameter, the best approach is to disassemble (with the *u* command) a line of code referencing the variable, and dump the corresponding memory location (e.g. *sp+4*).

8.3 Lucifer command set

Lucifer recognizes the commands listed in Table 8 - 2 on page 200.

8.3.1 The B command: set or display breakpoints

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

```
: b 10
Set breakpoint at _main+$28
:
```

Breakpoints can also include a semicolon separated list of Lucifer commands, which will be executed when the breakpoint is encountered. This makes it possible to create breakpoints which stop, display a value and then restart execution. For example, the command **`: b 10 @f pi;g`** creates a breakpoint which stops, displays the value of global variable *pi* and then continues execution.

8.3.2 The C command: display instruction at PC

The *c* command is used to display the assembler instruction and C source line addressed by the current value of the program counter. This is useful if you have been using other Lucifer commands and aren't quite sure where in the program the program counter is pointing. For example:

```
: c
10:     printf("answer = %d\n",j);
_main+$22       MOV     r0,_j
```

Table 8 - 2 Lucifer command set

Command	Meaning
B *[line|addr]*	Set or display breakpoints
C	Display instruction at PC
D *[addr [addr]]*	Display memory contents
E *fn|file*	Examine C source code
G *[addr]*	Commence execution
I	Toggle instruction trace mode
L *file*	Load a HEX file
M *addr val1 [val2 ...]*	Modify memory
P	Toggle input prompting mode
Q	Exit to operating system
R *[breakpnt]*	Remove breakpoints
S	Step one line
T	Trace one instruction
U *[addr]*	Disassemble machine instructions
W *file addr length*	Upload binary
X *[reg1 val1 [reg2 val2 ...]]*	Examine or change registers
@ *type [indirection]expr*	Display C variables
. *[breakpoint]*	Set a breakpoint and go
; *[line]*	Display from a source line
=	Display next page of source
-	Display previous page of source
/ *[string]*	Search source file for a string
! *command*	Execute a DOS command

8.3.3 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, 16 bytes are dumped from the address reached by the last d command. If one address is specified, 16 bytes are dumped from the address given. If two addresses are specified, the contents of memory from the first address to the second address are displayed. Dump addresses given can be symbols, line numbers, register names or absolute memory addresses.

8.3.4 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. For example:

```
: 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:}
:
```

8.3.5 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.

8.3.6 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. For example, with instruction trace disabled, step behaves like this:

```
: s
result = 20
Stepped to
10:}
:
```

With instruction trace enabled, step will instead behave like this:

```
: s
_memtest+30H    push    r4
_memtest+32H    mov     r0,#04A2H
_memtest+36H    push    r0
```

8

```
_memtest+38H    fcall   _printf
_memtest+3CH    adds    r7,#4
result = 20
Stepped to
10:}
:
```

Note that the library function *printf()* was not traced and thus operated properly and at full speed.

8.3.7 The L command: load a hex file

The *l* command is used to load object files into the target system. Lucifer correctly handles Motorola S-record format object files, Intel HEX files and binary images.

8.3.8 The M command: modify memory

The *m* command is used to write one or more values or ascii strings into memory at a specified address. This command takes the form: *m addr val1 [val2 ...]* where `addr` is the address to write to and all following arguments are values or strings to write to memory. Strings may use either single or double quotes. For example: **`: m buf "hello" 13 10 ’world’ 0`**

8.3.9 The P command: toggle input prompting mode

The *p* command is used to toggle input prompting. If input prompting is enabled, Lucifer will display a prompt “Target wants input:” when the target program executes an input function (*gets()*, *scanf()*, etc.). If input prompting is disabled, input prompting is left to the target program.

8.3.10 The Q command: exit to operating system

The *q* command is used to exit from Lucifer to the operating system. Note: the `q` command does not stop the target system (that is, the Lucifer monitor running on the target system), so it is possible to re-enter Lucifer without re-initializing the target.

8.3.11 The R command: remove breakpoints

The *r* command is used to remove breakpoints which have been set with the *b* command. If no arguments are given the user is prompted for each breakpoint in turn. For example:

```
: r
Remove _main+$28 ? y
Remove _main+$44 ? n
Remove _test ? n
: r main+$44
Removed breakpoint _main+$44
:
```

8.3.12 The S command: step one line

The `s` command is used to step by one line of C or assembler code. For example:

```
: 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:}
:
```

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. This avoids the slow process of single stepping through complex library routines like *printf()*. Normal library console I/O works correctly during single stepping using the `s` command. Where no line number information is available, such as inside library routines, the `s` command becomes an assembler step like the `t` command.

8.3.13 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.

8.3.14 The U command: disassemble machine instructions

The *u* command disassembles object code from the target system's memory. For example:

```
: u
9: printf("result = %d\n", result);
_memtest+30H    push    r4,r5
_memtest+32H    mov     r0,#04A2H
```

```
_memtest+36H    push    r0
_memtest+38H    fcall   _printf
_memtest+3CH    adds    r7,#6
```

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: by default, any values specified will be interpreted as absolute addresses. If you want to specify a line number, prefix it with a colon.

8.3.15 The W command: upload binary

The *w* command is used to upload and write a chunk of target memory as a binary file. This command takes three arguments: filename, start address and length. The start address and length values are in hex. For example, if the Lucifer monitor ROM were at $7000 to $7FFF in the target system, it could be uploaded to a binary file with the command:

```
: w lucrom.bin 7000 1000
........
Uploaded 4096 (0x1000) bytes to lucrom.bin
:
```

8.3.16 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.

8.3.17 The @ command: display C variables

The @ command is used to examine the contents of memory interpreted as one of the standard C types. The form of the @ command is: *@t/[*]expr* where *t* is the type of the variable to be displayed, * consists of zero or more indirection operators (“*” or “*n**”), and *expr* is the address of the variable to be displayed. Table 8 - 3 on page 205, shows the available @ command variants.

For example, to display a long variable `longvar` in hex: **`@lx longvar`**

To display a character, pointed to by a pointer `cptr`: **`@c *cptr`**

To de-reference ihandle: a pointer to a pointer to an unsigned int: **`@iu **ihandle`**

Table 8 - 3 Lucifer @ command variants

Command	Type	Displays
@c	char	character and value
@cu	unsigned char	decimal
@cx	unsigned char	hexadecimal
@co	unsigned char	octal
@i	int	decimal
@iu	unsigned int	decimal
@ix	unsigned int	hexadecimal
@io	unsigned int	octal
@l	long	decimal
@lu	unsigned long	decimal
@lx	unsigned long	hexadecimal
@lo	unsigned long	octal
@f	float	decimal
@np	near pointer	symbol+offset
@p	pointer	symbol+offset
@s	string	string chars

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

8.3.18 The . command: set a breakpoint and go

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

```
: . 10
Target wants input: 7
result = 20
Breakpoint
10:}
main+$28        RET
:
```

8.3.19 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. For example:

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

8.3.20 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.

8.3.21 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.

8.3.22 The / command: search source file for a string

The */* command is used to search the current source file for occurrences 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 occurrences of a string.

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

8.3.23 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.

8.3.24 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. In most 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`**.

If return is pressed after a breakpoint or **`.`** command has executed, it is equivalent to disassembling from the breakpoint address.

8.4 User input and output with Lucifer

The standard versions of the console I/O routines *putch()*, *getch()*, *getche()* and *init_uart()* are configured to work automatically with Lucifer. Code which is downloaded under Lucifer may use the standard I/O routines like *printf()* without any library modifications. Once you have finished debugging your code, you will need to insert into the library console, I/O routines suitable for your hardware. You can use the *getch.c* file in the *sources* directory as a starting point.

8.5 Installing Lucifer on a target

In order to use Lucifer on your target system, you will need to compile the Lucifer monitor and place it in a ROM. If your Z80 system already has a monitor in ROM, it is also possible to download the Lucifer target code into RAM. In most cases you will be able to use the Lucifer monitor program supplied without much modification.

8.5.1 Modifying the target code

Normally the only changes required will be the serial port drivers and the single step code.

Three versions of the target code are supplied; *jtarget.c* which is configured for the JED STD-801 board, *ztarget.c* which is configured for use with a generic Z80 with a Z80-SIO serial port, and *targ180.c* which is set up for use with a Z180 or 64180 processor, using one of the on-board serial ports.

There are extensive comments in these source files documenting any changes that might be needed for different hardware.

Error Messages

This chapter lists all possible error messages from the HI-TECH C compiler, with an explanation of each one. The name of the applications that could have produced the error are listed in brackets opposite the error message. The tutorial chapter describes the function of each application.

'.' expected after '..' ***(Parser)***
The only context in which two successive dots may appear is as part of the ellipsis symbol, which must have 3 dots.

'case' not in switch ***(Parser)***
A case statement has been encountered but there is no enclosing switch statement. A case statement may only appear inside the body of a switch statement.

'default' not in switch ***(Parser)***
A label has been encountered called "default" but it is not enclosed by a switch statement. The label "default" is only legal inside the body of a switch statement.

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

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

: no match** ***(Preprocessor, Parser)
This is an internal compiler error. Contact HI-TECH Software technical support with details.

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

-s, too few values specified in * ***(Preprocessor)***
The list of values to the preprocessor -S option is incomplete. This should not happen if the preprocessor is being invoked by the compiler driver or HPD.

-s, too many values, * unused ***(Preprocessor)***
There were too many values supplied to a -S preprocessor option.

... illegal in non-prototype arg list ***(Parser)***
The ellipsis symbol may only appear as the last item in a prototyped argument list. It may not appear on its own, nor may it appear after argument names that do not have types.

: expected ***(Parser)***
A colon is missing in a case label, or after the keyword "default". This often occurs when a semicolon is accidentally typed instead of a colon.

; expected ***(Parser)***
A semicolon is missing here. The semicolon is used as a terminator in many kinds of statements, e.g. do .. while, return etc.

= expected ***(Code Generator, Assembler)***
An equal sign was expected here.

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

#elif may not follow #else ***(Preprocessor)***
If a #else has been used after #if, you cannot then use a #elif in the same conditional block.

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

#else may not follow #else ***(Preprocessor)***
There can be only one #else corresponding to each #if.

#else must be in an #if ***(Preprocessor)***
#else can only be used after a matching #if.

#endif must be in an #if ***(Preprocessor)***
There must be a matching #if for each #endif. Check for the correct number of #ifs.

#error: * ***(Preprocessor)***
This is a programmer generated error; there is a directive causing a deliberate error. This is normally used to check compile time defines etc.

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

#if ... sizeof: bug, unknown type code * ***(Preprocessor)***
The preprocessor has made an internal error in evaluating a sizeof() expression. Check for a malformed type specifier.

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

#if bug, operand = * ***(Preprocessor)***
The preprocessor has tried to evaluate an expression with an operator it does not understand. This is an internal error.

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

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

#if value stack overflow ***(Preprocessor)***
The preprocessor filled up its expression evaluation stack in a #if expression. Simplify the expression - it probably contains too many parenthesized subexpressions.

#if, #ifdef, or #ifndef without an argument ***(Preprocessor)***
The preprocessor directives #if, #ifdef and #ifndef must have an argument. The argument to #if should be an expression, while the argument to #ifdef or #ifndef should be a single name.

#include syntax error ***(Preprocessor)***
The syntax of the filename argument to #include is invalid. The argument to #include must be a valid file name, either enclosed in double quotes ("") or angle brackets (< >). For example:

```
#include "afile.h"
#include <otherfile.h>
```

Spaces should not be included, and the closing quote or bracket must be present. There should be nothing else on the line.

#included file * was converted to lower case ***(Preprocessor)***
The #include file name had to be converted to lowercase before it could be opened.

] expected ***(Parser)***
A closing square bracket was expected in an array declaration or an expression using an array index.

{ expected ***(Parser)***
An opening brace was expected here.

} expected ***(Parser)***
A closing brace was expected here.

a parameter may not be a function ***(Parser)***
A function parameter may not be a function. It may be a pointer to a function, so perhaps a "*" has been omitted from the declaration.

absolute expression required ***(Assembler)***
An absolute expression is required in this context.

add_reloc - bad size ***(Assembler)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

ambiguous format name '*' ***(Cromwell)***
The output format specified to Cromwell is ambiguous.

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

argument -w* ignored ***(Linker)***
The argument to the linker option -w is out of range. For warning levels, the range is -9 to 9. For the map file width, the range is greater than or equal to 10.

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

argument redeclared: * ***(Parser)***
The specified argument is declared more than once in the same argument list.

argument too long ***(Preprocessor, Parser)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

arithmetic overflow in constant expression ***(Code Generator)***
A constant expression has been evaluated by the code generator that has resulted in a value that is too big for the type of the expression, e.g. trying to store the value 256 in a "char".

array dimension on * ignored ***(Preprocessor)***
An array dimension on a function parameter has been ignored because the argument is actually converted to a pointer when passed. Thus arrays of any size may be passed.

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

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

assertion *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

assertion failed: * *(Linker)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

attempt to modify const object *(Parser)*
Objects declared "const" may not be assigned to or modified in any other way.

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

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

bad ’-p’ format *(Linker)*
The "-P" option given to the linker is malformed.

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

-Aclass=low-high

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

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

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

bad arg * to tysize ***(Parser)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad arg to im ***(Assembler)***
The opcode "IM" only takes the constants 0, 1 or 2 as an argument.

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

bad bit number ***(Assembler, Optimiser)***
A bit number must be an absolute expression in the range 0-7.

bad bitfield type ***(Parser)***
A bitfield may only have a type of int.

bad character const ***(Parser, Assembler, Optimiser)***
This character constant is badly formed.

bad character in extended tekhex line * ***(Objtohex)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad checksum specification ***(Linker)***
A checksum list supplied to the linker is syntatically incorrect.

bad combination of flags ***(Objtohex)***
The combination of options supplied to objtohex is invalid.

bad complex range check ***(Linker)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

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

bad confloat - * ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad conval - * ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad dimensions ***(Code Generator)***
The code generator has been passed a declaration that results in an array having a zero dimension.

bad dp/nargs in openpar: c = * ***(Preprocessor)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

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

bad gn ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

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

bad int. code ***(Code Generator)***
The code generator has been passed input that is not syntatically correct.

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

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

bad min (+) format in spec ***(Linker)***
The minimum address specification in the linker’s -p option is badly formatted.

bad mod ’+’ for how = * ***(Code Generator)***
Internal error - Contact HI-TECH.

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

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

bad op * to revlog ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad op * to swaplog ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

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

bad operand ***(Optimiser)***
This operand is invalid. Check the syntax.

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

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

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

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

bad relocation type ***(Assembler)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad repeat count in -a spec ***(Linker)***
The repeat count given in a -A specification is invalid: it should be a valid decimal number.

bad ret_mask ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

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

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

-Gnxc+o

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

-Gnx4+16

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

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

bad size in index_type ***(Parser)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad size list ***(Parser)***
The first pass of the compiler has been invoked with a -Z option, specifying sizes of types, that is badly formed.

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

bad string * in psect pragma ***(Code Generator)***
The code generator has been passed a "pragma psect" directive that has a badly formed string. "Pragma psect" should be followed by something of the form "oldname=newname".

bad sx ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad u usage ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad variable syntax ***(Code Generator)***
There is an error in the intermediate code file. This could be caused by running out of disk space for temporary files.

bad which * after i ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

binary digit expected ***(Parser)***
A binary digit was expected. The format for a binary number is 0Bxxx where xxx is a string containing zeroes and/or ones, e.g.

0B0110

bit field too large (* bits) ***(Code Generator)***
The maximum number of bits in a bit field is the same as the number of bits in an "int".

bit range check failed * ***(Linker)***
The bit addressing was out of range.

bitfield comparison out of range ***(Code Generator)***
This is the result of comparing a bitfield with a value when the value is out of range of the bitfield. For example, comparing a 2-bit bitfield to the value 5 will never be true as a 2-bit bitfield has a range from 0 to 3,

bug: illegal __ macro * ***(Preprocessor)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

call depth exceeded by * ***(Linker)***
The call graph shows that functions are nested to a depth greater than specified.

can’t allocate memory for arguments ***(Preprocessor, Parser)***
The compiler could not allocate any more memory. Try increasing the size of available memory.

can’t allocate space for port variables: * ***(Code Generator)***
"Port" variables may only be declared "extern" or have an absolute address associated via the "@ address" construct. They may not be declared in such a way that would require the compiler to allocate space for them.

can’t be both far and near ***(Parser)***
It is illegal to qualify a type as both far and near.

can’t be long ***(Parser)***
Only "int" and "float" can be qualified with "long". Thus combinations like "long char" are illegal.

can’t be register ***(Parser)***
Only function parameters or auto (local) variables may be declared "register".

can’t be short ***(Parser)***
Only "int" can be modified with short. Thus combinations like "short float" are illegal.

can’t be unsigned ***(Parser)***
There is no such thing as an unsigned floating point number.

can’t call an interrupt function ***(Parser)***
A function qualified "interrupt" can’t be called from other functions. It can only be called by a hardware (or software) interrupt. This is because an interrupt function has special function entry and exit code that is appropriate only for calling from an interrupt. An "interrupt" function can call other non-interrupt functions.

can’t create * ***(Code Generator, Assembler, Linker, Optimiser)***
The named file could not be created. Check that all directories in the path are present.

can’t create cross reference file * *(Assembler)*
The cross reference file could not be created. Check that all directories are present. This can also be caused by the assembler running out of memory.

can’t create temp file *(Linker)*
The compiler was unable to create a temporary file. Check the DOS Environment variable TEMP (and TMP) and verify it points to a directory that exists, and that there is space available on that drive. For example, AUTOEXEC.BAT should have something like:

SET TEMP=C:\TEMP

where the directory C:\TEMP exists.

can’t create temp file * *(Code Generator)*
The compiler could not create the temporary file named. Check that all the directories in the file path exist.

can’t enter abs psect *(Assembler)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

can’t find op *(Assembler, Optimiser)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

can’t find space for psect * in segment * *(Linker)*
The named psect cannot be placed in the specified segment. This either means that the memory associated with the segment has been filled, or that the psect cannot be positioned in any of the available gaps in the memory. Split large functions (for CODE segements) in several smaller functions and ensure that the optimizers are being used.

can’t generate code for this expression *(Code Generator)*
This expression is too difficult for the code generator to handle. Try simplifying the expression, e.g. using a temporary variable to hold an intermediate result.

can’t have ’port’ variable: * *(Code Generator)*
The qualifier "port" can be used only with pointers or absolute variables. You cannot define a port variable as the compiler does not allocate space for port variables. You can declare an external port variable.

can’t have ’signed’ and ’unsigned’ together *(Parser)*
The type modifiers signed and unsigned cannot be used together in the same declaration, as they have opposite meaning.

can’t have an array of bits or a pointer to bit *(Parser)*
It is not legal to have an array of bits, or a pointer to bit.

can’t have array of functions *(Parser)*
You can’t have an array of functions. You can however have an array of pointers to functions. The correct syntax for an array of pointers to functions is "int (* arrayname[])();". Note that parentheses are used to associate the star (*) with the array name before the parentheses denoting a function.

can’t initialize arg *(Parser)*
A function argument can’t have an initialiser. The initialisation of the argument happens when the function is called and a value is provided for the argument by the calling function.

can’t mix proto and non-proto args *(Parser)*
A function declaration can only have all prototyped arguments (i.e. with types inside the parentheses) or all K&R style args (i.e. only names inside the parentheses and the argument types in a declaration list before the start of the function body).

can’t open *(Linker)*
A file can’t be opened - check spelling.

can’t open * *(Code Generator, Assembler, Optimiser, Cromwell)*
The named file could not be opened. Check the spelling and the directory path. This can also be caused by running out of memory.

can’t open avmap file * *(Linker)*
A file required for producing Avocet format symbol files is missing. Try re-installing the compiler.

can’t open checksum file * *(Linker)*
The checksum file specified to objtohex could not be opened. Check spelling etc.

can’t open command file * *(Preprocessor, Linker)*
The command file specified could not be opened for reading. Check spelling!

can’t open error file * *(Linker)*
The error file specified using the -e option could not be opened.

can’t open include file * *(Assembler)*
The named include file could not be opened. Check spelling. This can also be caused by running out of memory, or running out of file handles.

can’t open input file * *(Preprocessor, Assembler)*
The specified input file could not be opened. Check the spelling of the file name.

can’t open output file * *(Preprocessor, Assembler)*
The specified output file could not be created. This could be because a directory in the path name does not exist.

can’t reopen * ***(Parser)***
The compiler could not reopen a temporary file it had just created.

can’t seek in * ***(Linker)***
The linker can’t seek in the specified file. Make sure the output file is a valid filename.

can’t take address of register variable ***(Parser)***
A variable declared "register" may not have storage allocated for it in memory, and thus it is illegal to attempt to take the address of it by applying the "&" operator.

can’t take sizeof func ***(Parser)***
Functions don’t have sizes, so you can’t take use the "sizeof" operator on a function.

can’t take sizeof(bit) ***(Parser)***
You can’t take sizeof a bit value, since it is smaller than a byte.

can’t take this address ***(Parser)***
The expression which was the object of the "&" operator is not one that denotes memory storage ("an lvalue") and therefore its address can not be defined.

can’t use a string in an #if ***(Preprocessor)***
The preprocessor does not allow the use of strings in #if expressions.

cannot get memory ***(Linker)***
The linker is out of memory! This is unlikely to happen, but removing TSR’s etc. is the cure.

cannot open ***(Linker)***
A file cannot be opened - check spelling.

cannot open include file * ***(Preprocessor)***
The named include file could not be opened for reading by the preprocessor. Check the spelling of the filename. If it is a standard header file, not in the current directory, then the name should be enclosed in angle brackets (<>) not quotes.

cast type must be scalar or void ***(Parser)***
A typecast (an abstract type declarator enclosed in parentheses) must denote a type which is either scalar (i.e. not an array or a structure) or the type "void".

char const too long ***(Parser)***
A character constant enclosed in single quotes may not contain more than one character.

character not valid at this point in format specifier ***(Parser)***
The printf() style format specifier has an illegal character.

checksum error in intel hex file *, line * ***(Cromwell)***
A checksum error was found at the specified line in the specified Intel hex file. The file may have been corrupted.

circular indirect definition of symbol * ***(Linker)***
The specified symbol has been equated to an external symbol which, in turn, has been equated to the first symbol.

class * memory space redefined: */* ***(Linker)***
A class has been defined in two different memory spaces. Either rename one of the classes or, if they are the same class, place them in the same memory space.

close error (disk space?) ***(Parser)***
When the compiler closed a temporary file, an error was reported. The most likely cause of this is that there was insufficient space on disk for the file.

common symbol psect conflict: * ***(Linker)***
A common symbol has been defined to be in more than one psect.

complex relocation not supported for -r or -l options yet ***(Linker)***
The linker was given a -R or -L option with file that contain complex relocation. This is not yet supported.

conflicting fnconf records ***(Linker)***
This is probably caused by multiple run-time startoff module. Check the linker arguments, or "Object Files..." in HPD.

constant conditional branch ***(Code Generator)***
A conditional branch (generated by an "if" statement etc.) always follows the same path. This may indicate an expression with missing or badly placed parentheses, causing the evaluation to yield a value different to what you expected, or it may be because you have written something like "while(1)". To produce an infinite loop, use "for(;;)".

constant conditional branch: possible use of = instead of == ***(Code Generator)***
There is an expression inside an if or other conditional construct, where a constant is being assigned to a variable. This may mean you have inadvertently used an assignment (=) instead of a compare (==).

constant expression required ***(Parser)***
In this context an expression is required that can be evaluated to a constant at compile time.

constant left operand to ? ***(Code Generator)***
The left operand to a conditional operator (?) is constant, thus the result of the tertiary operator ?: will always be the same.

constant operand to || or && ***(Code Generator)***
One operand to the logical operators || or && is a constant. Check the expression for missing or badly placed parentheses.

constant relational expression ***(Code Generator)***
There is a relational expression that will always be true or false. This may be because e.g. you are comparing an unsigned number with a negative value, or comparing a variable with a value greater than the largest number it can represent.

control line * within macro expansion ***(Preprocessor)***
A preprocessor control line (one starting with a #) has been encountered while expanding a macro. This should not happen.

declaration of * hides outer declaration ***(Parser)***
An object has been declared that has the same name as an outer declaration (i.e. one outside and preceding the current function or block). This is legal, but can lead to accidental use of one variable when the outer one was intended.

declarator too complex ***(Parser)***
This declarator is too complex for the compiler to handle. Examine the declaration and find a way to simplify it. If the compiler finds it too complex, so will anybody maintaining the code.

default case redefined ***(Parser)***
There is only allowed to be one "default" label in a switch statement. You have more than one.

deff not supported in cp/m version ***(Assembler)***
The CP/M hosted assembler does not support floating point.

def[bmsf] in text psect ***(Optimiser)***
The assembler file supplied to the optimizer is invalid.

degenerate signed comparison ***(Code Generator)***
There is a comparision of a signed value with the most negative value possible for this type, such that the comparision will always be true or false. E.g. char c;

if(c >= -128)

will always be true, because an 8 bit signed char has a maximum negative value of -128.

degenerate unsigned comparison ***(Code Generator)***
There is a comparision of an unsigned value with zero, which will always be true or false. E.g.

unsigned char c;
if(c >= 0)

will always be true, because an unsigned value can never be less than zero.

delete what ? ***(Libr)***
The librarian requires one or more modules to be listed for deletion when using the ’d’ key.

did not recognize format of input file ***(Cromwell)***
The input file to Cromwell is required to be COD, Intel HEX, Motorola HEX, COFF, OMF51, P&E or HI-TECH.

digit out of range ***(Parser, Assembler, Optimiser)***
A digit in this number is out of range of the radix for the number, e.g. using the digit 8 in an octal number, or hex digits A-F in a decimal number. An octal number is denoted by the digit string commencing with a zero, while a hex number starts with "0X" or "0x".

dimension required ***(Parser)***
Only the most significant (i.e. the first) dimension in a multi-dimension array may not be assigned a value. All succeeding dimensions must be present.

direct range check failed * ***(Linker)***
The direct addressing was out of range.

directive not recognized ***(Assembler)***
An assembler directive is unrecognized. Check spelling.

divide by zero in #if, zero result assumed ***(Preprocessor)***
Inside a #if expression, there is a division by zero which has been treated as yielding zero.

division by zero ***(Code Generator)***
A constant expression that was being evaluated involved a division by zero.

double float argument required ***(Parser)***
The printf format specifier corresponding to this argument is %f or similar, and requires a floating point expression. Check for missing or extra format specifiers or arguments to printf.

duplicate -d or -h flag ***(Linker)***
The a symbol file name has been specified to the linker for a second time.

duplicate -m flag ***(Linker)***
The linker only likes to see one -m flag, unless one of them does not specify a file name. Two map file names are more than it can handle!

duplicate case label ***(Code Generator)***
There are two case labels with the same value in this switch statement.

duplicate label * ***(Parser)***
The same name is used for a label more than once in this function. Note that the scope of labels is the entire function, not just the block that encloses a label.

duplicate qualifier ***(Parser)***
There are two occurrences of the same qualifier in this type specification. This can occur either directly or through the use of a typedef. Remove the redundant qualifier.

duplicate qualifier key * ***(Parser)***
This qualifier key (given via a -Q option) has been used twice.

duplicate qualifier name * ***(Parser)***
A duplicate qualifier name has been specified to P1 via a -Q option. This should not occur if the standard compiler drivers are used.

end of file within macro argument from line * ***(Preprocessor)***
A macro argument has not been terminated. This probably means the closing parenthesis has been omitted from a macro invocation. The line number given is the line where the macro argument started.

end of string in format specifier ***(Parser)***
The format specifier for the printf() style function is malformed.

end statement inside include file or macro ***(Assembler)***
An END statement was found inside an include file or a macro.

entry point multiply defined ***(Linker)***
There is more than one entry point defined in the object files given the linker.

enum tag or { expected ***(Parser)***
After the keyword "enum" must come either an identifier that is or will be defined as an enum tag, or an opening brace.

eof in #asm ***(Preprocessor)***
An end of file has been encountered inside a #asm block. This probably means the #endasm is missing or misspelt.

eof in comment ***(Preprocessor)***
End of file was encountered inside a comment. Check for a missing closing comment flag.

eof inside conditional ***(Assembler)***
END-of-FILE was encountered while scanning for an "endif" to match a previous "if".

eof inside macro def'n ***(Assembler)***
End-of-file was encountered while processing a macro definition. This means there is a missing "endm" directive.

eof on string file ***(Parser)***
P1 has encountered an unexpected end-of-file while re-reading its file used to store constant strings before sorting and merging. This is most probably due to running out of disk space. Check free disk space.

error closing output file ***(Code Generator, Optimiser)***
The compiler detected an error when closing a file. This most probably means there is insufficient disk space.

error dumping * ***(Cromwell)***
Either the input file to Cromwell is of an unsupported type or that file cannot be dumped to the screen.

error in format string ***(Parser)***
There is an error in the format string here. The string has been interpreted as a printf() style format string, and it is not syntactically correct. If not corrected, this will cause unexpected behaviour at run time.

evaluation period has expired ***(Driver)***
The evaluation period for this compiler has expired. Contact HI-TECH to purchase a full licence.

expand - bad how ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

expand - bad which ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

expected '-' in -a spec ***(Linker)***
There should be a minus sign (-) between the high and low addresses in a -A spec, e.g.

```
-AROM=1000h-1FFFh
```

exponent expected ***(Parser)***
A floating point constant must have at least one digit after the "e" or "E".

expression error ***(Code Generator, Assembler, Optimiser)***
There is a syntax error in this expression, OR there is an error in the intermediate code file. This could be caused by running out of disk space.

expression generates no code ***(Code Generator)***
This expression generates no code. Check for things like leaving off the parentheses in a function call.

expression stack overflow at op * ***(Preprocessor)***
Expressions in #if lines are evaluated using a stack with a size of 128. It is possible for very complex expressions to overflow this. Simplify the expression.

expression syntax ***(Parser)***
This expression is badly formed and cannot be parsed by the compiler.

expression too complex ***(Parser)***
This expression has caused overflow of the compiler's internal stack and should be re-arranged or split into two expressions.

external declaration inside function ***(Parser)***
A function contains an "extern" declaration. This is legal but is invariably A Bad Thing as it restricts the scope of the function declaration to the function body. This means that if the compiler encounters another declaration, use or definition of the extern object later in the same file, it will no longer have the earlier declaration and thus will be unable to check that the declarations are consistent. This can lead to strange behaviour of your program or signature errors at link time. It will also hide any previous declarations of the same thing, again subverting the compiler's type checking. As a general rule, always declare "extern" variables and functions outside any other functions.

fast interrupt can't be used in large model ***(Code Generator)***
The large model (bank switched) does not support fast interrupts, as the alternate register set is used for bank switching.

field width not valid at this point ***(Parser)***
A field width may not appear at this point in a printf() type format specifier.

file name index out of range in line no. record ***(Cromwell)***
The .COD file has an invalid format in the specified record.

filename work buffer overflow ***(Preprocessor)***
A filename constructed while looking for an include file has exceeded the length of an internal buffer. Since this buffer is 4096 bytes long, this is unlikely to happen.

fixup overflow in expression * ***(Linker)***
The linker was asked to relocate (fixup) an item that would not fit back into the space after relocation. For example this will occur if a byte size object is initialized with an address that is bigger than 255. This error occurred in a complex expression.

fixup overflow referencing * ***(Linker)***
The linker was asked to relocate (fixup) an item that would not fit back into the space after relocation. For example this will occur if a byte size object is initialized with an address that is bigger than 255.

flag * unknown ***(Assembler)***
This option used on a "PSECT" directive is unknown to the assembler.

float param coerced to double ***(Parser)***
Where a non-prototyped function has a parameter declared as "float", the compiler converts this into a "double float". This is because the default C type conversion conventions provide that when a floating point number is passed to a non-prototyped function, it will be converted to double. It is important that the function declaration be consistent with this convention.

floating exponent too large ***(Assembler)***
The exponent of the floating point number is too large. For the Z80 the largest floating point exponent is decimal 19.

floating number expected ***(Assembler)***
The arguments to the "DEFF" pseudo-op must be valid floating point numbers.

formal parameter expected after # ***(Preprocessor)***
The stringization operator # (not to be confused with the leading # used for preprocessor control lines) must be followed by a formal macro parameter. If you need to stringize a token, you will need to define a special macro to do it, e.g.

#define __mkstr__(x) #x

then use __mkstr__(token) wherever you need to convert a token into a string.

function * appears in multiple call graphs: rooted at * ***(Linker)***
This function can be called from both main line code and interrupt code. Use the reentrant keyword, if this compiler supports it, or recode to avoid using local variables or parameters, or duplicate the function.

function * argument evaluation overlapped ***(Linker)***
A function call involves arguments which overlap between two functions. This could occur with a call like:

void fn1(void) { fn3(7, fn2(3), fn2(9)); /* Offending call */ } char fn2(char fred) { return fred + fn3(5,1,0); } char fn3(char one, char two, char three) { return one+two+three; }

where fn1 is calling fn3, and two arguments are evaluated by calling fn2, which in turn calls fn3. The structure should be modified to prevent this.

function * is never called ***(Linker)***
This function is never called. This may not represent a problem, but space could be saved by removing it. If you believe this function should be called, check your source code.

function body expected ***(Parser)***
Where a function declaration is encountered with K&R style arguments (i.e. argument names but no types inside the parentheses) a function body is expected to follow.

function declared implicit int ***(Parser)***
Where the compiler encounters a function call of a function whose name is presently undefined, the compiler will automatically declare the function to be of type "int", with unspecified (K&R style) parameters. If a definition of the function is subsequently encountered, it is possible that its type and arguments will be different from the earlier implicit declaration, causing a compiler error. The solution is to ensure that all functions are defined or at least declared before use, preferably with prototyped parameters. If it is necessary to make a forward declaration of a function, it should be preceded with the keywords "extern" or "static" as appropriate.

function does not take arguments ***(Parser, Code Generator)***
This function has no parameters, but it is called here with one or more arguments.

function is already 'extern'; can't be 'static' ***(Parser)***
This function was already declared extern, possibly through an implicit declaration. It has now been redeclared static, but this redeclaration is invalid. If the problem has arisen because of use before definition, either move the definition earlier in the file, or place a static forward definition earlier in the file, e.g. static int fred(void);

function or function pointer required ***(Parser)***
Only a function or function pointer can be the subject of a function call. This error can be produced when an expression has a syntax error resulting in a variable or expression being followed by an opening parenthesis ("(") which denotes a function call.

functions can't return arrays ***(Parser)***
A function can return only a scalar (simple) type or a structure. It cannot return an array.

functions can't return functions ***(Parser)***
A function cannot return a function. It can return a function pointer. A function returning a pointer to a function could be declared like this: int (* (name()))(). Note the many parentheses that are necessary to make the parts of the declaration bind correctly.

functions nested too deep ***(Code Generator)***
This error is unlikely to happen with C code, since C cannot have nested functions!

garbage after operands ***(Assembler)***
There is something on this line after the operands other than a comment. This could indicate an operand error.

garbage on end of line ***(Assembler)***
There were non-blank and non-comment characters after the end of the operands for this instruction. Note that a comment must be started with a semicolon.

hex digit expected ***(Parser)***
After "0x" should follow at least one of the hex digits 0-9 and A-F or a-f.

I/O error reading symbol table ***(Cromwell)***
Cromwell could not read the symbol table. This could be because the file was truncated or there was some other problem reading the file.

ident records do not match ***(Linker)***
The object files passed to the linker do not have matching ident records. This means they are for different processor types.

identifier expected ***(Parser)***
Inside the braces of an "enum" declaration should be a comma-separated list of identifiers.

identifier redefined: * ***(Parser)***
This identifier has already been defined. It cannot be defined again.

identifier redefined: * (from line *) ***(Parser)***
This identifier has been defined twice. The ’from line’ value is the line number of the first declaration.

illegal # command * ***(Preprocessor)***
The preprocessor has encountered a line starting with #, but which is not followed by a recognized control keyword. This probably means the keyword has been misspelt. Legal control keywords are: assert, asm, define, elif, else, endasm, endif, error, if, ifdef, ifndef, include, line, pragma, undef.

illegal #if line ***(Preprocessor)***
There is a syntax error in the expression following #if. Check the expression to ensure it is properly constructed.

illegal #undef argument ***(Preprocessor)***
The argument to #undef must be a valid name. It must start with a letter.

illegal ’#’ directive ***(Preprocessor, Parser)***
The compiler does not understand the "#" directive. It is probably a misspelling of a pre-processor "#" directive.

illegal character (* decimal) in #if ***(Preprocessor)***
The #if expression had an illegal character. Check the line for correct syntax.

illegal character * ***(Parser)***
This character is illegal.

illegal character * in #if ***(Preprocessor)***
There is a character in a #if expression that has no business being there. Valid characters are the letters, digits and those comprising the acceptable operators.

illegal conversion ***(Parser)***
This expression implies a conversion between incompatible types, e.g. a conversion of a structure type into an integer.

illegal conversion between pointer types ***(Parser)***
A pointer of one type (i.e. pointing to a particular kind of object) has been converted into a pointer of a different type. This will usually mean you have used the wrong variable, but if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed.

illegal conversion of integer to pointer ***(Parser)***
An integer has been assigned to or otherwise converted to a pointer type. This will usually mean you have used the wrong variable, but if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed.

illegal conversion of pointer to integer ***(Parser)***
A pointer has been assigned to or otherwise converted to a integral type. This will usually mean you have used the wrong variable, but if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed.

illegal flag * ***(Linker)***
This flag is unrecognized.

illegal function qualifier(s) ***(Parser)***
A qualifier such as "const" or "volatile" has been applied to a function. These qualifiers only make sense when used with an lvalue (i.e. an expression denoting memory storage). Perhaps you left out a star ("*") indicating that the function should return a pointer to a qualified object.

illegal initialisation ***(Parser)***
You can’t initialise a "typedef" declaration, because it does not reserve any storage that could be initialised.

illegal operation on a bit variable ***(Parser)***
Not all operations on bit variables are supported. This operation is one of those.

illegal operator in #if ***(Preprocessor)***
A #if expression has an illegal operator. Check for correct syntax.

illegal or too many -g flags ***(Linker)***
There has been more than one -g option, or the -g option did not have any arguments following. The arguments specify how the segment addresses are calculated.

illegal or too many -o flags ***(Linker)***
This -o flag is illegal, or another -o option has been encountered. A -o option to the linker must have a filename. There should be no space between the filename and the -o, e.g. -ofile.obj

illegal or too many -p flags ***(Linker)***
There have been too many -p options passed to the linker, or a -p option was not followed by any arguments. The arguments of separate -p options may be combined and separated by commas.

illegal record type ***(Linker)***
There is an error in an object file. This is either an invalid object file, or an internal error in the linker. Try recreating the object file.

illegal relocation size: * ***(Linker)***
There is an error in the object code format read by the linker. This either means you are using a linker that is out of date, or that there is an internal error in the assembler or linker.

illegal relocation type: * ***(Linker)***
An object file contained a relocation record with an illegal relocation type. This probably means the file is corrupted or not an object file.

illegal switch * ***(Code Generator, Assembler, Optimiser)***
This command line option was not understood.

illegal type for array dimension ***(Parser)***
An array dimension must be either an integral type or an enumerated value.

illegal type for index expression ***(Parser)***
An index expression must be either integral or an enumerated value.

illegal type for switch expression ***(Parser)***
A "switch" operation must have an expression that is either an integral type or an enumerated value.

illegal use of void expression ***(Parser)***
A void expression has no value and therefore you can't use it anywhere an expression with a value is required, e.g. as an operand to an arithmetic operator.

image too big ***(Objtohex)***
The program image being constructed by objtohex is too big for its virtual memory system.

implicit conversion of float to integer ***(Parser)***
A floating point value has been assigned or otherwise converted to an integral type. This could result in truncation of the floating point value. A typecast will make this warning go away.

implicit return at end of non-void function *(Parser)*
A function which has been declared to return a value has an execution path that will allow it to reach the end of the function body, thus returning without a value. Either insert a return statement with a value, or if the function is not to return a value, declare it "void".

implict signed to unsigned conversion *(Parser)*
A signed number is being assigned or otherwise converted to a larger unsigned type. Under the ANSI "value preserving" rules, this will result in the signed value being first sign-extended to a signed number the size of the target type, then converted to unsigned (which involves no change in bit pattern). Thus an unexpected sign extension can occur. To ensure this does not happen, first convert the signed value to an unsigned equivalent, e.g. if you want to assign a signed char to an unsigned int, first typecast the char value to "unsigned char".

inappropriate ’else’ *(Parser)*
An "else" keyword has been encountered that cannot be associated with an "if" statement. This may mean there is a missing brace or other syntactic error.

inappropriate break/continue *(Parser)*
A "break" or "continue" statement has been found that is not enclosed in an appropriate control structure. "continue" can only be used inside a "while", "for" or "do while" loop, while "break" can only be used inside those loops or a "switch" statement.

include files nested too deep *(Assembler)*
Macro expansions and include file handling have filled up the assembler’s internal stack. The maximum number of open macros and include files is 30.

included file * was converted to lower case *(Preprocessor)*
The file specified to be included was not found, but a file with a lowercase version of the name of the file specified was found and used instead.

incompatible intermediate code version; should be * *(Code Generator)*
The intermediate code file produced by P1 is not the correct version for use with this code generator. This is either that incompatible versions of one or more compilers have been installed in the same directory, or a temporary file error has occurred leading to corruption of a temporary file. Check the setting of the TEMP environment variable. If it refers to a long path name, change it to something shorter.

incomplete * record body: length = * *(Linker)*
An object file contained a record with an illegal size. This probably means the file is truncated or not an object file.

incomplete ident record *(Libr)*
The IDENT record in the object file was incomplete.

incomplete record ***(Objtohex, Libr)***
The object file passed to objtohex or the librarian is corrupted.

incomplete record: * ***(Linker)***
An object code record is incomplete. This is probably due to a corrupted or invalid object module. Recompile the source file, watching for out of disk space errors etc.

incomplete record: type = * length = *
This message is produced by the DUMP or XSTRIP utilities and indicates that the object file is not a valid HI-TECH object file, or that it has been truncated, possibly due to running out of disk or RAMdisk space.

incomplete symbol record ***(Libr)***
The SYM record in the object file was incomplete.

inconsistent lineno tables ***(Cromwell)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

inconsistent storage class ***(Parser)***
A declaration has conflicting storage classes. Only one storage class should appear in a declaration.

inconsistent symbol tables ***(Cromwell)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

inconsistent type ***(Parser)***
Only one basic type may appear in a declaration, thus combinations like "int float" are illegal.

index offset too large ***(Assembler)***
An offset on a Z80 indexed addressing form must lie in the range -128 to 127.

initialisation syntax ***(Parser)***
The initialisation of this object is syntactically incorrect. Check for the correct placement and number of braces and commas.

initializer in ’extern’ declaration ***(Parser)***
A declaration containing the keyword "extern" has an initialiser. This overrides the "extern" storage class, since to initialise an object it is necessary to define (i.e. allocate storage for) it.

insufficient memory for macro def’n ***(Assembler)***
There is not sufficient memory to store a macro definition.

integer constant expected ***(Parser)***
A colon appearing after a member name in a structure declaration indicates that the member is a bitfield. An integral constant must appear after the colon to define the number of bits in the bitfield.

integer expression required ***(Parser)***
In an "enum" declaration, values may be assigned to the members, but the expression must evaluate to a constant of type "int".

integral argument required ***(Parser)***
An integral argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments.

integral type required ***(Parser)***
This operator requires operands that are of integral type only.

interrupt_level should be 0 to 7 ***(Parser)***
The pragma ’interrupt_level’ must have an argument from 0 to 7.

invalid disable: * ***(Preprocessor)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

invalid format specifier or type modifier ***(Parser)***
The format specifier or modifier in the printf() style string is illegal for this particular format.

invalid hex file: *, line * ***(Cromwell)***
The specified Hex file contains an invalid line.

invalid number syntax ***(Assembler, Optimiser)***
The syntax of a number is invalid. This can be, e.g. use of 8 or 9 in an octal number, or other malformed numbers.

jump out of range ***(Assembler)***
A short jump ("JR") instruction has been given an address that is more than 128 bytes away from the present location. Use the "JP" opcode instead.

label identifier expected ***(Parser)***
An identifier denoting a label must appear after "goto".

lexical error ***(Assembler, Optimiser)***
An unrecognized character or token has been seen in the input.

library * is badly ordered ***(Linker)***
This library is badly ordered. It will still link correctly, but it will link faster if better ordered.

library file names should have .lib extension: * ***(Libr)***
Use the .lib extension when specifying a library.

line does not have a newline on the end ***(Parser)***
The last line in the file is missing the newline (linefeed, hex 0A) from the end. Some editors will create such files, which can cause problems for include files. The ANSI C standard requires all source files to consist of complete lines only.

line too long ***(Optimiser)***
This line is too long. It will not fit into the compiler’s internal buffers. It would require a line over 1000 characters long to do this, so it would normally only occur as a result of macro expansion.

local illegal outside macros ***(Assembler)***
The "LOCAL" directive is only legal inside macros. It defines local labels that will be unique for each invocation of the macro.

local psect ’*’ conflicts with global psect of same name ***(Linker)***
A local psect may not have the same name as a global psect.

logical type required ***(Parser)***
The expression used as an operand to "if", "while" statements or to boolean operators like ! and && must be a scalar integral type.

long argument required ***(Parser)***
A long argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments.

macro * wasn’t defined ***(Preprocessor)***
A macro name specified in a -U option to the preprocessor was not initially defined, and thus cannot be undefined.

macro argument after * must be absolute ***(Assembler)***
The argument after * in a macro call must be absolute, as it must be evaluated at macro call time.

macro argument may not appear after local ***(Assembler)***
The list of labels after the directive "LOCAL" may not include any of the formal parameters to the macro.

macro expansions nested too deep ***(Assembler)***
Macro expansions in the assembler are nested too deep. The limit is 30 macros and include files nested at one time.

macro work area overflow ***(Preprocessor)***
The total length of a macro expansion has exceeded the size of an internal table. This table is normally 8192 bytes long. Thus any macro expansion must not expand into a total of more than 8K bytes.

member * redefined ***(Parser)***
This name of this member of the struct or union has already been used in this struct or union.

members cannot be functions ***(Parser)***
A member of a structure or a union may not be a function. It may be a pointer to a function. The correct syntax for a function pointer requires the use of parentheses to bind the star ("*") to the pointer name, e.g. "int (*name)();".

metaregister * can’t be used directly ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

mismatched comparision ***(Code Generator)***
A comparison is being made between a variable or expression and a constant value which is not in the range of possible values for that expression, e.g. if you compare an unsigned character to the constant value 300, the result will always be false (not equal) since an unsigned character can NEVER equal 300. As an 8 bit value it can represent only 0-255.

misplaced ’?’ or ’:’, previous operator is * ***(Preprocessor)***
A colon operator has been encountered in a #if expression that does not match up with a corresponding ? operator. Check parentheses etc.

misplaced constant in #if ***(Preprocessor)***
A constant in a #if expression should only occur in syntactically correct places. This error is most probably caused by omission of an operator.

missing ’)’ ***(Parser)***
A closing parenthesis was missing from this expression.

missing ’=’ in class spec ***(Linker)***
A class spec needs an = sign, e.g. -Ctext=ROM

missing ’]’ ***(Parser)***
A closing square bracket was missing from this expression.

missing arg to -a ***(Parser)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

missing arg to -e ***(Linker)***
The error file name must be specified following the -e linker option.

missing arg to -i ***(Parser)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

missing arg to -j ***(Linker)***
The maximum number of errors before aborting must be specified following the -j linker option.

missing arg to -q ***(Linker)***
The -Q linker option requires the machine type for an argument.

missing arg to -u ***(Linker)***
The -U (undefine) option needs an argument, e.g. -U_symbol

missing arg to -w ***(Linker)***
The -W option (listing width) needs a numeric argument.

missing argument to ’pragma psect’ ***(Parser)***
The pragma ’psect’ requires an argument of the form oldname=newname where oldname is an existing psect name known to the compiler, and newname is the desired new name. Example: #pragma psect bss=battery

missing argument to ’pragma switch’ ***(Parser)***
The pragma ’switch’ requires an argument of auto, direct or simple.

missing basic type: int assumed ***(Parser)***
This declaration does not include a basic type, so int has been assumed. This declaration is not illegal, but it is preferable to include a basic type to make it clear what is intended.

missing key in avmap file ***(Linker)***
A file required for producing Avocet format symbol files is corrupted. Try re-installing the compiler.

missing memory key in avmap file ***(Linker)***
A file required for producing Avocet format symbol files is corrupted. Try re-installing the compiler.

missing name after pragma ’inline’ ***(Parser)***
The ’inline’ pragma has the syntax:

#pragma inline func_name

where func_name is the name of a function which is to be expanded to inline code. This pragma has no effect except on functions specially recognized by the code generator.

missing name after pragma ’printf_check’ ***(Parser)***
The pragma ’printf_check’, which enable printf style format string checking for a function, requires a function name, e.g.

#pragma printf_check sprintf

missing newline ***(Preprocessor)***
A new line is missing at the end of the line. Each line, including the last line, must have a new line at the end. This problem is normally introduced by editors.

missing number after % in -p option ***(Linker)***
The % operator in a -p option (for rounding boundaries) must have a number after it.

missing number after pragma 'pack' *(Parser)*
The pragma 'pack' requires a decimal number as argument. For example

#pragma pack(1)

will prevent the compiler aligning structure members onto anything other than one byte boundaries. Use this with caution as some processors enforce alignment and will not operate correctly if word fetches are made on odd boundaries (e.g. 68000, 8096).

missing number after pragma interrupt_level *(Parser)*
Pragma 'interrupt_level' requires an argument from 0 to 7.

missing processor name after -p *(Cromwell)*
The -p option to cromwell must specify a processor.

mod by zero in #if, zero result assumed *(Preprocessor)*
A modulus operation in a #if expression has a zero divisor. The result has been assumed to be zero.

module * defines no symbols *(Libr)*
No symbols were found in the module's object file.

module has code below file base of * *(Linker)*
This module has code below the address given, but the -C option has been used to specify that a binary output file is to be created that is mapped to this address. This would mean code from this module would have to be placed before the beginning of the file! Check for missing psect directives in assembler files.

multi-byte constant * isn't portable *(Preprocessor)*
Multi-byte constants are not portable, and in fact will be rejected by later passes of the compiler.

multiple free: * *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

multiply defined symbol * *(Assembler, Linker)*
This symbol has been defined in more than one place in this module.

near function should be static *(Code Generator)*
A near function in the bank switched model should be static, as it cannot be called from another module.

nested #asm directive *(Preprocessor)*
It is not legal to nest #asm directives. Check for a missing or misspelt #endasm directive.

nested comments *(Preprocessor)*
This warning is issued when nested comments are found. A nested comment may indicate that a previous closing comment marker is missing or malformed.

no #asm before #endasm ***(Preprocessor)***
A #endasm operator has been encountered, but there was no previous matching #asm.

no arg to -o ***(Assembler)***
The assembler requires that an output file name argument be supplied after the "-O" option. No space should be left between the -O and the filename.

no case labels ***(Code Generator)***
There are no case labels in this switch statement.

no end record ***(Linker)***
This object file has no end record. This probably means it is not an object file.

no end record found ***(Linker)***
An object file did not contain an end record. This probably means the file is corrupted or not an object file.

no file arguments ***(Assembler)***
The assembler has been invoked without any file arguments. It cannot assemble anything.

no identifier in declaration ***(Parser)***
The identifier is missing in this declaration. This error can also occur where the compiler has been confused by such things as missing closing braces.

no input files specified ***(Cromwell)***
Cromwell must have an input file to convert.

no memory for string buffer ***(Parser)***
P1 was unable to allocate memory for the longest string encountered, as it attempts to sort and merge strings. Try reducing the number or length of strings in this module.

no output file format specified ***(Cromwell)***
The output format must be specified to Cromwell.

no psect specified for function variable/argument allocation ***(Linker)***
This is probably caused by omission of correct run-time startoff module. Check the linker arguments, or "Object Files..." in HPD.

no room for arguments ***(Preprocessor, Parser, Code Generator, Linker, Objtohex)***
The code generator could not allocate any more memory. Try increasing the size of available memory.

no space for macro def’n ***(Assembler)***
The assembler has run out of memory.

no start record: entry point defaults to zero *(Linker)*
None of the object files passed to the linker contained a start record. The start address of the program has been set to zero. This may be harmless, but it is recommended that you define a start address in your startup module by using the "END" directive.

no. of arguments redeclared *(Parser)*
The number of arguments in this function declaration does not agree with a previous declaration of the same function.

nodecount = * *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

non-constant case label *(Code Generator)*
A case label in this switch statement has a value which is not a constant.

non-prototyped function declaration: * *(Parser)*
A function has been declared using old-style (K&R) arguments. It is preferable to use prototype declarations for all functions. If the function has no arguments, declare it as e.g. "int func(void)".

non-scalar types can’t be converted *(Parser)*
You can’t convert a structure, union or array to anything else. You can convert a pointer to one of those things, so perhaps you left out an ampersand ("&").

non-void function returns no value *(Parser)*
A function that is declared as returning a value has a "return" statement that does not specify a return value.

not a member of the struct/union * *(Parser)*
This identifier is not a member of the structure or union type with which it used here.

not a variable identifier: * *(Parser)*
This identifier is not a variable; it may be some other kind of object, e.g. a label.

not an argument: * *(Parser)*
This identifier that has appeared in a K&R stype argument declarator is not listed inside the parentheses after the function name. Check spelling.

null format name *(Cromwell)*
The -I or -O option to Cromwell must specify a file format.

object code version is greater than * *(Linker)*
The object code version of an object module is higher than the highest version the linker is known to work with. Check that you are using the correct linker.

object file is not absolute ***(Objtohex)***
The object file passed to objtohex has relocation items in it. This may indicate it is the wrong object file, or that the linker or objtohex have been given invalid options.

only functions may be qualified interrupt ***(Parser)***
The qualifier "interrupt" may not be applied to anything except a function.

only functions may be void ***(Parser)***
A variable may not be "void". Only a function can be "void".

only lvalues may be assigned to or modified ***(Parser)***
Only an lvalue (i.e. an identifier or expression directly denoting addressable storage) can be assigned to or otherwise modified. A typecast does not yield an lvalue. To store a value of different type into a variable, take the address of the variable, convert it to a pointer to the desired type, then dereference that pointer, e.g. "*(int *)&x = 1" is legal whereas "(int)x = 1" is not.

only modifier l valid with this format ***(Parser)***
The only modifier that is legal with this format is l (for long).

only modifiers h and l valid with this format ***(Parser)***
Only modifiers h (short) and l (long) are legal with this printf() format specifier.

only register storage class allowed ***(Parser)***
The only storage class allowed for a function parameter is "register".

oops! -ve number of nops required! ***(Assembler)***
An internal error has occurred. Contact HI-TECH.

operand error ***(Assembler, Optimiser)***
The operand to this opcode is invalid. Check you assembler reference manual for the proper form of operands for this instruction.

operands of * not same pointer type ***(Parser)***
The operands of this operator are of different pointer types. This probably means you have used the wrong pointer, but if the code is actually what you intended, use a typecast to suppress the error message.

operands of * not same type ***(Parser)***
The operands of this operator are of different pointer. This probably means you have used the wrong variable, but if the code is actually what you intended, use a typecast to suppress the error message.

operator * in incorrect context ***(Preprocessor)***
An operator has been encountered in a #if expression that is incorrectly placed, e.g. two binary operators are not separated by a value.

out of far memory *(Code Generator)*
The compiler has run out of far memory. Try removing TSR's etc. If your system supports EMS memory, the compiler will be able to use up to 64K of this, so if it is not enable, try enabling EMS.

out of memory *(Code Generator, Assembler, Optimiser)*
The compiler has run out of memory. If you have unnecessary TSRs loaded, remove them. If you are running the compiler from inside another program, try running it directly from the command prompt. Similarly, if you are using HPD, try using the command line compiler driver instead.

out of memory allocating * blocks of * *(Linker)*
Memory was required to extend an array but was unavailable.

out of memory for assembler lines *(Optimiser)*
The optimizer has run out of memory to store assembler lines, e.g. #asm lines from the C source, or C source comment lines. Reduce the size of the function.

out of near memory *(Code Generator)*
The compiler has run out of near memory. This is probably due to too many symbol names. Try splitting the program up, or reducing the number of unused symbols in header files etc.

out of space in macro * arg expansion *(Preprocessor)*
A macro argument has exceeded the length of an internal buffer. This buffer is normally 4096 bytes long.

output file cannot be also an input file *(Linker)*
The linker has detected an attempt to write its output file over one of its input files. This cannot be done, because it needs to simultaneously read and write input and output files.

page width must be >= * *(Assembler)*
The listing page width must be at least * characters. Any less will not allow a properly formatted listing to be produced.

phase error *(Assembler)*
The assembler has calculated a different value for a symbol on two different passes. This is probably due to bizarre use of macros or conditional assembly.

phase error in macro args *(Assembler)*
The assembler has detected a difference in the definition of a symbol on the first and a subsequent pass.

phase error on temporary label *(Assembler)*
The assembler has detected a difference in the definition of a symbol on the first and a subsequent pass.

pointer required *(Parser)*
A pointer is required here. This often means you have used "->" with a structure rather than a structure pointer.

pointer to * argument required ***(Parser)***
A pointer argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments.

pointer to non-static object returned ***(Parser)***
This function returns a pointer to a non-static (e.g. automatic) variable. This is likely to be an error, since the storage associated with automatic variables becomes invalid when the function returns.

portion of expression has no effect ***(Code Generator)***
Part of this expression has no side effects, and no effect on the value of the expression.

possible pointer truncation ***(Parser)***
A pointer qualified "far" has been assigned to a default pointer or a pointer qualified "near", or a default pointer has been assigned to a pointer qualified "near". This may result in truncation of the pointer and loss of information, depending on the memory model in use.

preprocessor assertion failure ***(Preprocessor)***
The argument to a preprocessor #assert directive has evaluated to zero. This is a programmer induced error.

probable missing '}' in previous block ***(Parser)***
The compiler has encountered what looks like a function or other declaration, but the preceding function has not been ended with a closing brace. This probably means that a closing brace has been omitted from somewhere in the previous function, although it may well not be the last one.

psect * cannot be in classes * ***(Linker)***
A psect cannot be in more than one class. This is either due to assembler modules with conflicting class= options, or use of the -C option to the linker.

psect * memory delta redefined: */* ***(Linker)***
A global psect has been defined with two different deltas.

psect * memory space redefined: */* ***(Linker)***
A global psect has been defined in two different memory spaces. Either rename one of the psects or, if they are the same psect, place them in the same memory space using the SPACE psect flag.

psect * not loaded on * boundary ***(Linker)***
This psect has a relocatability requirement that is not met by the load address given in a -P option. For example if a psect must be on a 4K byte boundary, you could not start it at 100H.

psect * not relocated on * boundary ***(Linker)***
This psect is not relocated on the required boundary. Check the relocatability of the psect and correct the -p option. if necessary.

psect * not specified in -p option ***(Linker)***
This psect was not specified in a -P or -A option to the linker. It has been linked at the end of the program, which is probably not where you wanted it.

psect * re-orged ***(Linker)***
This psect has had its start address specified more than once.

psect * selector value redefined ***(Linker)***
The selector value for this psect has been defined more than once.

psect * type redefined: * ***(Linker)***
This psect has had its type defined differently by different modules. This probably means you are trying to link incompatible object modules, e.g. linking 386 flat model code with 8086 real mode code.

psect exceeds address limit: * ***(Linker)***
The maximum address of the psect exceeds the limit placed on it using the LIMIT psect flag.

psect exceeds max size: * ***(Linker)***
The psect has more bytes in it than the maximum allowed as specified using the SIZE psect flag.

psect is absolute: * ***(Linker)***
This psect is absolute and should not have an address specified in a -P option.

psect may not be local and global ***(Assembler)***
A psect may not be declared to be local if it has already been declared to be (default) global.

psect origin multiply defined: * ***(Linker)***
The origin of this psect is defined more than once.

psect property redefined ***(Assembler)***
A property of a psect has been defined in more than place to be different.

psect reloc redefined ***(Assembler)***
The relocatability of this psect has been defined differently in two or more places.

psect selector redefined ***(Linker)***
The selector associated with this psect has been defined differently in two or more places.

psect size redefined ***(Assembler)***
The maximum size of this psect has been defined differently in two or more places.

qualifiers redeclared ***(Parser)***
This function has different qualifiers in different declarations.

read error on * ***(Linker)***
The linker encountered an error trying to read this file.

record too long ***(Objtohex)***
This indicates that the object file is not a valid HI-TECH object file.

record too long: * ***(Linker)***
An object file contained a record with an illegal size. This probably means the file is corrupted or not an object file.

recursive function calls: ***(Linker)***
These functions (or function) call each other recursively. One or more of these functions has statically allocated local variables (compiled stack). Either use the reentrant keyword (if supported with this compiler) or recode to avoid recursion.

recursive macro definition of * ***(Preprocessor)***
The named macro has been defined in such a manner that expanding it causes a recursive expansion of itself!

redefining macro * ***(Preprocessor)***
The macro specified is being redefined, to something different to the original definition. If you want to deliberately redefine a macro, use #undef first to remove the original definition.

redundant & applied to array ***(Parser)***
The address operator "&" has been applied to an array. Since using the name of an array gives its address anyway, this is unnecessary and has been ignored.

refc == 0 ***(Assembler, Optimiser)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

regused - bad arg to g ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

relocation error ***(Assembler, Optimiser)***
It is not possible to add together two relocatable quantities. A constant may be added to a relocatable value, and two relocatable addresses in the same psect may be subtracted. An absolute value must be used in various places where the assembler must know a value at assembly time.

relocation offset * out of range * ***(Linker)***
An object file contained a relocation record with a relocation offset outside the range of the preceding text record. This means the object file is probably corrupted.

relocation too complex ***(Assembler)***
The complex relocation in this expression is too big to be inserted into the object file.

remsym error ***(Assembler)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

replace what ? ***(Libr)***
The librarian requires one or more modules to be listed for replacement when using the ’r’ key.

rept argument must be >= 0 ***(Assembler)***
The argument to a "REPT" directive must be greater than zero.

seek error: * ***(Linker)***
The linker could not seek when writing an output file.

segment * overlaps segment * ***(Linker)***
The named segments have overlapping code or data. Check the addresses being assigned by the "-P" option.

signatures do not match: * ***(Linker)***
The specified function has different signatures in different modules. This means it has been declared differently, e.g. it may have been prototyped in one module and not another. Check what declarations for the function are visible in the two modules specified and make sure they are compatible.

signed bitfields not supported ***(Parser)***
Only unsigned bitfields are supported. If a bitfield is declared to be type "int", the compiler still treats it as unsigned.

simple integer expression required ***(Parser)***
A simple integral expression is required after the operator "@", used to associate an absolute address with a variable.

simple type required for * ***(Parser)***
A simple type (i.e. not an array or structure)is required as an operand to this operator.

sizeof external array * is zero ***(Parser)***
The sizeof an external array evaluates to zero. This is probably due to the array not having an explicit dimension in the extern declaration.

sizeof yields 0 ***(Code Generator)***
The code generator has taken the size of an object and found it to be zero. This almost certainly indicates an error in your declaration of a pointer, e.g. you may have declared a pointer to a zero length array. In general, pointers to arrays are of little use. If you require a pointer to an array of objects of unknown length, you only need a pointer to a single object that can then be indexed or incremented.

static object has zero size: * ***(Code Generator)***
A static object has been declared, but has a size of zero.

storage class illegal ***(Parser)***
A structure or union member may not be given a storage class. Its storage class is determined by the storage class of the structure.

storage class redeclared ***(Parser)***
A variable or function has been re-declared with a different storage class. This can occur where there are two conflicting declarations, or where an implicit declaration is followed by an actual declaration.

strange character * after ## ***(Preprocessor)***
A character has been seen after the token catenation operator ## that is neither a letter nor a digit. Since the result of this operator must be a legal token, the operands must be tokens containing only letters and digits.

strange character after # * ***(Preprocessor)***
There is an unexpected character after #.

string concatenation across lines ***(Parser)***
Strings on two lines will be concatenated. Check that this is the desired result.

string expected ***(Parser)***
The operand to an "asm" statement must be a string enclosed in parentheses.

string lookup failed in coff:get_string() ***(Cromwell)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

string too long ***(Assembler)***
This string is too long. Shorten it.

struct/union member expected ***(Parser)***
A structure or union member name must follow a dot (".") or arrow ("->").

struct/union redefined: * ***(Parser)***
A structure or union has been defined more than once.

struct/union required ***(Parser)***
A structure or union identifier is required before a dot (".").

struct/union tag or '{' expected ***(Parser)***
An identifier denoting a structure or union or an opening brace must follow a "struct" or "union" keyword.

symbol * cannot be global ***(Linker)***
There is an error in an object file, where a local symbol has been declared global. This is either an invalid object file, or an internal error in the linker. Try recreating the object file.

symbol * has erroneous psect: * ***(Linker)***
There is an error in an object file, where a symbol has an invalid psect. This is either an invalid object file, or an internal error in the linker. Try recreating the object file.

symbol * not defined in #undef ***(Preprocessor)***
The symbol supplied as argument to #undef was not already defined. This is a warning only, but could be avoided by including the #undef in a #ifdef ... #endif block.

syntax error ***(Assembler, Optimiser)***
A syntax error has been detected. This could be caused a number of things.

syntax error in -a spec ***(Linker)***
The -A spec is invalid. A valid -A spec should be something like:

-AROM=1000h-1FFFh

syntax error in checksum list ***(Linker)***
There is a syntax error in a checksum list read by the linker. The checksum list is read from standard input by the linker, in response to an option. Re-read the manual on checksum list.

syntax error in local argument ***(Assembler)***
There is a syntax error in a local argument.

text does not start at 0 ***(Linker)***
Code in some things must start at zero. Here it doesn't.

text offset too low ***(Linker)***
You aren't likely to see this error. Rhubarb!

text record has bad length: * ***(Linker)***
There is an error in an object file. This is either an invalid object file, or an internal error in the linker. Try recreating the object file.

text record has length too small: * ***(Linker)***
This indicates that the object file is not a valid HI-TECH object file.

this function too large - try reducing level of optimization ***(Code Generator)***
A large function has been encountered when using a -Og (global optimization) switch. Try re-compiling without the global optimization, or reduce the size of the function.

this is a struct ***(Parser)***
This identifier following a "union" or "enum" keyword is already the tag for a structure, and thus should only follow the keyword "struct".

this is a union ***(Parser)***
This identifier following a "struct" or "enum" keyword is already the tag for a union, and thus should only follow the keyword "union".

this is an enum ***(Parser)***
This identifier following a "struct" or "union" keyword is already the tag for an enumerated type, and thus should only follow the keyword "enum".

too few arguments ***(Parser)***
This function requires more arguments than are provided in this call.

too few arguments for format string ***(Parser)***
There are too few arguments for this format string. This would result in a garbage value being printed or converted at run time.

too many (*) enumeration constants ***(Parser)***
There are too many enumeration constants in an enumerated type. The maximum number of enumerated constants allowed in an enumerated type is 512.

too many (*) structure members ***(Parser)***
There are too many members in a structure or union. The maximum number of members allowed in one structure or union is 512.

too many address spaces - space * ignored ***(Linker)***
The limit to the number of address spaces is currently 16.

too many arguments ***(Parser)***
This function does not accept as many arguments as there are here.

too many arguments for format string ***(Parser)***
There are too many arguments for this format string. This is harmless, but may represent an incorrect format string.

too many arguments for macro ***(Preprocessor)***
A macro may only have up to 31 parameters, as per the C Standard.

too many arguments in macro expansion ***(Preprocessor)***
There were too many arguments supplied in a macro invocation. The maximum number allowed is 31.

too many cases in switch ***(Code Generator)***
There are too many case labels in this switch statement. The maximum allowable number of case labels in any one switch statement is 511.

too many comment lines - discarding ***(Assembler)***
The compiler is generating assembler code with embedded comments, but this function is so large that an excessive number of source line comments are being generated. This has been suppressed so that the optimizer will not run out of memory loading comment lines.

too many errors ***(Preprocessor, Parser, Code Generator, Assembler, Linker)***
There were so many errors that the compiler has given up. Correct the first few errors and many of the later ones will probably go away.

too many file arguments. usage: cpp [input [output]] ***(Preprocessor)***
CPP should be invoked with at most two file arguments.

too many files in coff file ***(Cromwell)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

too many include directories ***(Preprocessor)***
A maximum of 7 directories may be specified for the preprocessor to search for include files.

too many initializers ***(Parser)***
There are too many initializers for this object. Check the number of initializers against the object definition (array or structure).

too many input files ***(Cromwell)***
To many input files have been specified to be converted by Cromwell.

too many macro parameters ***(Assembler)***
There are too many macro parameters on this macro definition.

too many nested #* statements ***(Preprocessor)***
#if, #ifdef etc. blocks may only be nested to a maximum of 32.

too many nested #if statements ***(Preprocessor)***
#if, #ifdef etc. blocks may only be nested to a maximum of 32.

too many output files ***(Cromwell)***
To many output file formats have been specified to Cromwell.

too many psect class specifications ***(Linker)***
There are too many psect class specifications (-C options)

too many psect pragmas ***(Code Generator)***
Too many "pragma psect" directives have been used.

too many psects ***(Assembler)***
There are too many psects! Boy, what a program!

too many qualifier names ***(Parser)***
There are too many qualifier names specified.

too many relocation items ***(Objtohex)***
Objtohex filled up a table. This program is just way too complex!

too many segment fixups ***(Objtohex)***
There are too many segment fixups in the object file given to objtohex.

too many segments ***(Objtohex)***
There are too many segments in the object file given to objtohex.

too many symbols ***(Assembler)***
There are too many symbols for the assemblers symbol table. Reduce the number of symbols in your program. If it is the linker producing this error, suggest changing some global to local symbols.

too many symbols (*) ***(Linker)***
There are too many symbols in the symbol table, which has a limit of * symbols. Change some global symbols to local symbols to reduce the number of symbols.

too many symbols in * ***(Optimiser)***
There are too many symbols in the specified function. Reduce the size of the function.

too many temporary labels ***(Assembler)***
There are too many temporary labels in this assembler file. The assembler allows a maximum of 2000 temporary labels.

too much indirection ***(Parser)***
A pointer declaration may only have 16 levels of indirection.

too much pushback ***(Preprocessor)***
This error should not occur, and represents an internal error in the preprocessor.

type conflict ***(Parser)***
The operands of this operator are of incompatible types.

type modifier already specified ***(Parser)***
This type modifier has already be specified in this type.

type modifiers not valid with this format ***(Parser)***
Type modifiers may not be used with this format.

type redeclared ***(Parser)***
The type of this function or object has been redeclared. This can occur because of two incompatible declarations, or because an implicit declaration is followed by an incompatible declaration.

type specifier reqd. for proto arg ***(Parser)***
A type specifier is required for a prototyped argument. It is not acceptable to just have an identifier.

unable to open list file * ***(Linker)***
The named list file could not be opened.

unbalanced paren's, op is * *(Preprocessor)*
The evaluation of a #if expression found mismatched parentheses. Check the expression for correct parenthesisation.

undefined *: * *(Parser)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

undefined enum tag: * *(Parser)*
This enum tag has not been defined.

undefined identifier: * *(Parser)*
This symbol has been used in the program, but has not been defined or declared. Check for spelling errors.

undefined shift (* bits) *(Code Generator)*
An attempt has been made to shift a value by a number of bits equal to or greater than the number of bits in the data type, e.g. shifting a long by 32 bits. This will produce an undefined result on many processors. This is non-portable code and is flagged as having undefined results by the C Standard.

undefined struct/union *(Parser)*
This structure or union tag is undefined. Check spelling etc.

undefined struct/union: * *(Parser)*
The specified structure or union tag is undefined. Check spelling etc.

undefined symbol * *(Assembler)*
The named symbol is not defined, and has not been specified "GLOBAL".

undefined symbol * in #if, 0 used *(Preprocessor)*
A symbol on a #if expression was not a defined preprocessor macro. For the purposes of this expression, its value has been taken as zero.

undefined symbol in fnaddr record: * *(Linker)*
The linker has found an undefined symbol in the fnaddr record for a non-reentrant function.

undefined symbol in fnbreak record: * *(Linker)*
The linker has found an undefined symbol in the fnbreak record for a non-reentrant function.

undefined symbol in fncall record: * *(Linker)*
The linker has found an undefined symbol in the fncall record for a non-reentrant function.

undefined symbol in fnindir record: * *(Linker)*
The linker has found an undefined symbol in the fnindir record for a non-reentrant function.

undefined symbol in fnroot record: * ***(Linker)***
The linker has found an undefined symbol in the fnroot record for a non-reentrant function.

undefined symbol in fnsize record: * ***(Linker)***
The linker has found an undefined symbol in the fnsize record for a non-reentrant function.

undefined symbol: ***(Assembler, Linker)***
The symbol following is undefined at link time. This could be due to spelling error, or failure to link an appropriate module.

undefined symbols: ***(Linker)***
A list of symbols follows that were undefined at link time.

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

undefined variable: * ***(Parser)***
This variable has been used but not defined at this point.

unexpected end of file ***(Linker)***
This probably means an object file has been truncated because of a lack of disk space.

unexpected eof ***(Parser)***
An end-of-file was encountered unexpectedly. Check syntax.

unexpected text in #control line ignored ***(Preprocessor)***
This warning occurs when extra characters appear on the end of a control line, e.g.

```
#endif something
```

The "something" will be ignored, but a warning is issued. It is preferable (and in accordance with Standard C) to enclose the "something" as a comment, e.g.

```
#endif /* something */
```

unexpected \ in #if ***(Preprocessor)***
The backslash is incorrect in the #if statement.

unknown 'with' psect referenced by psect * ***(Linker)***
The specified psect has been placed with a psect using the psect 'with' flag. The psect it has been placed with does not exist.

unknown complex operator * ***(Linker)***
There is an error in an object file. This is either an invalid object file, or an internal error in the linker. Try recreating the object file.

unknown fnrec type * ***(Linker)***
This indicates that the object file is not a valid HI-TECH object file.

unknown format name '*' ***(Cromwell)***
The output format specified to Cromwell is unknown.

unknown option * ***(Preprocessor)***
This option to the preprocessor is not recognized.

unknown pragma * ***(Parser)***
An unknown pragma directive was encountered.

unknown predicate * ***(Code Generator)***
Internal error - Contact HI-TECH.

unknown psect ***(Optimiser)***
The assembler file read by the optimizer has an unknown psect.

unknown psect: * ***(Linker, Optimiser)***
This psect has been listed in a -P option, but is not defined in any module within the program.

unknown qualifier '*' given to -a ***(Parser)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

unknown qualifier '*' given to -i ***(Parser)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

unknown record type: * ***(Linker)***
An invalid object module has been read by the linker. It is either corrupted or not an object file.

unknown register name * ***(Linker)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

unknown symbol type * ***(Linker)***
The symbol type encountered is unknown to this linker. Check that the correct linker is being used.

unreachable code ***(Parser)***
This section of code will never be executed, because there is no execution path by which it could be reached. Look for missing "break" statements inside a control structure like "while" or "for".

unreasonable matching depth ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

unrecognized option to -z: * ***(Code Generator)***
The code generator has been passed a -Z option it does not understand. This should not happen if it is invoked with the standard driver.

unrecognized qualifer name after ’strings’ ***(Parser)***
The pragma ’strings’ requires a list of valid qualifier names. For example

#pragma strings const code

would add const and code to the current string qualifiers. If no qualifiers are specified, all qualification will be removed from subsequent strings. The qualifier names must be recognized by the compiler.

unterminated #if[n][def] block from line * ***(Preprocessor)***
A #if or similar block was not terminated with a matching #endif. The line number is the line on which the #if block began.

unterminated comment in included file ***(Preprocessor)***
Comments begun inside an included file must end inside the included file.

unterminated macro arg ***(Assembler)***
An argument to a macro is not terminated. Note that angle brackets ("< >") are used to quote macro arguments.

unterminated string ***(Assembler, Optimiser)***
A string constant appears not to have a closing quote missing.

unterminated string in macro body ***(Preprocessor, Assembler)***
A macro definition contains a string that lacks a closing quote.

unused constant: * ***(Parser)***
This enumerated constant is never used. Maybe it isn’t needed at all.

unused enum: * ***(Parser)***
This enumerated type is never used. Maybe it isn’t needed at all.

unused label: * ***(Parser)***
This label is never used. Maybe it isn’t needed at all.

unused member: * ***(Parser)***
This structure member is never used. Maybe it isn’t needed at all.

unused structure: * ***(Parser)***
This structure tag is never used. Maybe it isn’t needed at all.

unused typedef: * ***(Parser)***
This typedef is never used. Maybe it isn’t needed at all.

unused union: * ***(Parser)***
This union type is never used. Maybe it isn’t needed at all.

unused variable declaration: * ***(Parser)***
This variable is never used. Maybe it isn’t needed at all.

unused variable definition: * ***(Parser)***
This variable is never used. Maybe it isn’t needed at all.

upper case #include files are non-portable ***(Preprocessor)***
When using DOS, the case of an #include file does not matter. In other operating systems the case is significant.

variable may be used before set: * ***(Code Generator)***
This variable may be used before it has been assigned a value. Since it is an auto variable, this will result in it having a random value.

void function cannot return value ***(Parser)***
A void function cannot return a value. Any "return" statement should not be followed by an expression.

while expected ***(Parser)***
The keyword "while" is expected at the end of a "do" statement.

work buffer overflow doing * ## ***(Preprocessor)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

work buffer overflow: * ***(Preprocessor)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

write error (out of disk space?) * ***(Linker)***
Probably means that the hard disk is full.

write error on * ***(Assembler, Linker, Cromwell)***
A write error occurred on the named file. This probably means you have run out of disk space.

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

wrong number of macro arguments for * - * instead of * ***(Preprocessor)***
A macro has been invoked with the wrong number of arguments.

Library Functions

The functions within the HI-TECH C Z80/Z180 compiler library are listed in this chapter. Each entry begins with the name of the function. This is followed by information analysed into the following headings.

Synopsis

This is the C definition of the function, and the header file in which it is declared.

Description

This is a narrative description of the function and its purpose.

Example

This is an example of the use of the function. It is usually a complete small program that illustrates the function.

Data types

If any special data types (structures etc.) are defined for use with the function, they are listed here with their C definition. These data types will be defined in the header file given under heading - Synopsis.

See also

This refers you to any allied functions.

Return value

The type and nature of the return value of the function, if any, is given. Information on error returns is also included

Only those headings which are relevant to each function are used.

ABORT

Synopsis

```
#include <stdlib.h>

void abort (void)
```

Description

The function **`abort()`** is called to terminate the program abnormally. It will print an appropriate message and exit with a status of negative one (-1), CPM only. Under DOS, the user will need to provide a printf statement prior to calling abort. This function is not available for an embedded program.

Example

```
#include <stdio.h>
#include <stdlib.h>

void
main (void)
{
    char * ptr, c;

    ptr = &c;
    if(ptr == NULL)
        abort();
}
```

Return Value

Never returns.

Note

This routine is not usable in a ROM based system.

ABS

Synopsis

```
#include <stdlib.h>

int abs (int j)
```

Description

The **`abs()`** function returns the absolute value of **`j`**.

Example

```
#include <stdio.h>
#include <stdlib.h>

void
main (void)
{
    int a = -5;

    printf("The absolute value of %d is %d\n", a, abs(a));
}
```

Return Value

The absolute value of **`j`**.

ACOS

Synopsis

```
#include <math.h>

double acos (double f)
```

Description

The **`acos()`** function implements the converse of `cos()`, i.e. it is passed a value in the range -1 to +1, and returns an angle in radians whose cosine is equal to that value.

Example

```
#include <math.h>
#include <stdio.h>

    /* Print acos() values for -1 to 1 in degrees. */

void
main (void)
{
    float i, a;

    for(i = -1.0; i < 1.0 ; i += 0.1) {
        a = acos(i)*180.0/3.141592;
        printf("acos(%f) = %f degrees\n", i, a);
    }
}
```

See Also

`sin(), cos(), tan(), asin(), atan(), atan2()`

Return Value

An angle in radians, in the range 0 to π. Where the argument value is outside the domain -1 to 1, the return value will be zero.

ASCTIME

Synopsis

```
#include <time.h>

char * asctime (struct tm * t)
```

Description

The **`asctime()`** function takes the time broken down into the **`struct tm`** structure, pointed to by its argument, and returns a 26 character string describing the current date and time in the format:

Sun Sep 16 01:03:52 1973\n\0

Note the *newline* at the end of the string. The width of each field in the string is fixed. The example gets the current time, converts it to a **`struct tm`** pointer with `localtime()`, it then converts this to ASCII and prints it. The `time()` function will need to be provided by the user (see `time()` for details).

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;
    struct tm * tp;

    time(&clock);
    tp = localtime(&clock);
    printf("%s", asctime(tp));
}
```

See Also

`ctime(), gmtime(), localtime(), time()`

Return Value

A pointer to the string.

Note

The example will require the user to provide the `time()` routine as it cannot be supplied with the compiler. See `time()` for more details.

Data Types

```
struct tm {
     int tm_sec;
     int tm_min;
     int tm_hour;
     int tm_mday;
     int tm_mon;
     int tm_year;
     int tm_wday;
     int tm_yday;
     int tm_isdst;
};
```

ASIN

Synopsis

```
#include <math.h>

double asin (double f)
```

Description

The **`asin()`** function implements the converse of `sin()`, i.e. it is passed a value in the range -1 to +1, and returns an angle in radians whose sine is equal to that value.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    float i, a;

    for(i = -1.0; i < 1.0 ; i += 0.1) {
        a = asin(i)*180.0/3.141592;
        printf("asin(%f) = %f degrees\n", i, a);
    }
}
```

See Also

`sin(), cos(), tan(), acos(), atan(), atan2()`

Return Value

An angle in radians, in the range $-\pi/2$ to $+\pi/2$. Where the argument value is outside the domain -1 to 1, the return value will be zero.

ASSERT

Synopsis

```
#include <assert.h>

void assert (int e)
```

Description

This macro is used for debugging purposes; the basic method of usage is to place assertions liberally throughout your code at points where correct operation of the code depends upon certain conditions being true initially. An **`assert()`** routine may be used to ensure at run time that an assumption holds true. For example, the following statement asserts that the pointer `tp` is not equal to `NULL`:

assert(tp);

If at run time the expression evaluates to false, the program will abort with a message identifying the source file and line number of the assertion, and the expression used as an argument to it. A fuller discussion of the uses of **`assert()`** is impossible in limited space, but it is closely linked to methods of proving program correctness.

Example

```
void
ptrfunc (struct xyz * tp)
{
    assert(tp != 0);
}
```

Note

When required for ROM based systems, the underlying routine `_fassert(...)` will need to be implemented by the user.

ATAN

Synopsis

```
#include <math.h>

double atan (double x)
```

Description

This function returns the arc tangent of its argument, i.e. it returns an angle `e` in the range -π/2 to π/2 such that `tan(e) == x`.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f\n", atan(1.5));
}
```

See Also

`sin(), cos(), tan(), asin(), acos(), atan2()`

Return Value

The arc tangent of its argument.

ATAN2

Synopsis

```
#include <math.h>

double atan2 (double y, double x)
```

Description

This function returns the arc tangent of **y/x**, using the sign of both arguments to determine the quadrant of the return value.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f\n", atan2(1.5, 1));
}
```

See Also

`sin(), cos(), tan(), asin(), acos(), atan()`

Return Value

The arc tangent of **y/x** in the range -π to +π radians. If both **y** and **x** are zero, a domain error occurs and zero is returned.

ATEXIT

Synopsis

```
#include <stdlib.h>

int atexit (void (*func)(void))
```

Description

The **`atexit()`** function, registers the function pointed to by **`func`**, to be called without arguments at normal program termination. On program termination, all functions registered by **`atexit()`** are called, in the reverse order of their registration.

Example

```
#include <stdio.h>
#include <stdlib.h>

char * fname;
FILE * fp;

void
rmfile (void)
{
    if(fp)
        close(fp);
    if(fname)
        remove(fname);
}
        /* create a file; on exit, close and remove it */
void
main (void)
{
    if(!(fp = fopen((fname = "test.fil"), "w")))
        exit(1);
    atexit(rmfile);
}
```

See Also

`exit()`

Return Value

The **`atexit()`** function returns zero if the registration succeeds, nonzero if it fails.

Note

This routine is not usable in a ROM based system.

ATOF

Synopsis

```
#include <stdlib.h>

double atof (const char * s)
```

Description

The **`atof()`** function scans the character string passed to it, skipping leading blanks. It then converts an ASCII representation of a number to a double. The number may be in decimal, normal floating point or scientific notation.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    double i;

    gets(buf);
    i = atof(buf);
    printf("Read %s: converted to %f\n", buf, i);
}
```

See Also

`atoi(), atol()`

Return Value

A double precision floating point number. If no number is found in the string, 0.0 will be returned.

ATOI

Synopsis

```
#include <stdlib.h>

int atoi (const char * s)
```

Description

The **`atoi()`** function scans the character string passed to it, skipping leading blanks and reading an optional sign. It then converts an ASCII representation of a decimal number to an integer.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    int i;

    gets(buf);
    i = atoi(buf);
    printf("Read %s: converted to %d\n", buf, i);
}
```

See Also

`xtoi(), atof(), atol()`

Return Value

A signed integer. If no number is found in the string, 0 will be returned.

ATOL

Synopsis

```
#include <stdlib.h>

long atol (const char * s)
```

Description

The **`atol()`** function scans the character string passed to it, skipping leading blanks. It then converts an ASCII representation of a decimal number to a long integer.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    long i;

    gets(buf);
    i = atol(buf);
    printf("Read %s: converted to %ld\n", buf, i);
}
```

See Also

`atoi(), atof()`

Return Value

A long integer. If no number is found in the string, 0 will be returned.

BDOS

Synopsis

```
#include <cpm.h>

char bdos (int func, int arg)
```

Description

The **`bdos()`** function calls the CP/M BDOS with **`func`** in register C (CL for CP/M-86) and **`arg`** in register DE (DX), if required. The return value is the byte returned by the BDOS in register A (AX). Constant values for the various BDOS function values are defined in **`cpm.h`**.

Example

```
#include <cpm.h>

    /* test to see if a key has been pressed */

char
kbhit (void)
{
    return(bdos(0x0B) & 0xFF) != 0;
}
```

See Also

`bdoshl(), bios(), msdos()`

Note

This routine is not usable in a ROM based system.

BDOSHL

Synopsis

```
#include <cpm.h>

short bdoshl (int, ...)
```

Description

This function will make a CP/M BDOS call, returning the value contained in HL after the call. It is used for some BDOS calls where the return value is in HL. The example shows the use of **`bdoshl()`** to determine the CP/M version number.

Example

```
#include <cpm.h>
#include <stdio.h>

void
main (void)
{
    short verno;

    verno = bdoshl(12);
    printf("CP/M version number %d.%d\n", (verno >> 4) & 0xF, verno & 0xF);
}
```

See Also

`bdos(), bios(), msdos()`

Note

This routine is not usable in a ROM based system.

BIOS

Synopsis

```
#include <cpm.h>

char bios (int n, int a1, int a2)
```

Description

This function will call the **n**'th bios entry point (cold boot = 0, warm boot = 1, etc.) with register BC (CX) set to the argument **a1** and DE (DX) set to the argument **a2**. The return value is the contents of register A (AX) after the bios call. On CP/M-86, BDOS function 50 is used to perform the bios call. This function should not be used unless unavoidable, since it is highly non-portable. There is even no guarantee of portability of bios calls between differing CP/M systems.

See Also

bdos()

Note

This routine is not usable in a ROM based system.

BSEARCH

Synopsis

```
#include <stdlib.h>

void * bsearch (const void * key, void * base, size_t n_memb,
          size_t size, int (*compar)(const void *, const void *))
```

Description

The **`bsearch()`** function searches a sorted array for an element matching a particular key. It uses a binary search algorithm, calling the function pointed to by **`compar`** to compare elements in the array.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

struct value {
    char name[40];
    int value;
} values[100];

int
val_cmp (const void * p1, const void * p2)
{
    return strcmp(((const struct value *)p1)->name,
            ((const struct value *)p2)->name);
}

void
main (void)
{
    char inbuf[80];
    int i;
    struct value * vp;

    i = 0;
    while(gets(inbuf)) {
        sscanf(inbuf,"%s %d", values[i].name, &values[i].value);
        i++;
    }
    qsort(values, i, sizeof values[0], val_cmp);
    vp = bsearch("fred", values, i, sizeof values[0], val_cmp);
    if(!vp)
        printf("Item ’fred’ was not found\n");
    else
        printf("Item ’fred’ has value %d\n", vp->value);
}
```

See Also

`qsort()`

Return Value

A pointer to the matched array element (if there is more than one matching element, any of these may be returned). If no match is found, a null pointer is returned.

Note

The comparison function must have the correct prototype.

CALLOC

Synopsis

```
#include <stdlib.h>

void * calloc (size_t cnt, size_t size)
```

Description

The **`calloc()`** function attempts to obtain a contiguous block of dynamic memory which will hold **`cnt`** objects, each of length **`size`**. The block is filled with zeros.

Example

```
#include <stdlib.h>
#include <stdio.h>

struct test {
    int a[20];
} * ptr;

    /* Allocate space for 20 structures. */

void
main (void)
{

    ptr = calloc(20, sizeof(struct test));
    if(!ptr)
        printf("Failed\n");
    else
        free(ptr);
}
```

See Also

`brk(), sbrk(), malloc(), free()`

Return Value

A pointer to the block is returned, or zero if the memory could not be allocated.

CEIL

Synopsis

```
#include <math.h>

double ceil (double f)
```

Description

This routine returns the smallest whole number not less than **f**.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    double j;

    scanf("%lf", &j);
    printf("The ceiling of %lf is %lf\n", j, ceil(j));
}
```

CGETS

Synopsis

```
#include <conio.h>

char * cgets (char * s)
```

Description

The **`cgets()`** function will read one line of input from the console into the buffer passed as an argument. It does so by repeated calls to `getche()`. As characters are read, they are buffered, with *backspace* deleting the previously typed character, and *ctrl-U* deleting the entire line typed so far. Other characters are placed in the buffer, with a *carriage return* or *line feed (newline)* terminating the function. The collected string is null terminated.

Example

```
#include <conio.h>
#include <string.h>

char buffer[80];

void
main (void)
{
    for(;;) {
        cgets(buffer);
        if(strcmp(buffer, "exit") == 0)
            break;
        cputs("Type ’exit’ to finish\n");
    }
}
```

See Also

`getch(), getche(), putch(), cputs()`

Return Value

The return value is the character pointer passed as the sole argument.

10

CHMOD

Synopsis

```
#include <stat.h>
#include <unixio.h>

int chmod (const char * name, unsigned mode)
```

Description

This function changes the file attributes (or modes) of the named file. The argument **`name`** may be any valid file name. The **`mode`** argument may include all bits defined in **`stat.h`** except those relating to the type of the file, e.g. `S_IFDIR`.

Example

```
#include <stat.h>
#include <stdio.h>
#include <unixio.h>

    /* make a file read-only */

void
main (int argc, char ** argv)
{
    if(argc > 1)
        if(chmod(argv[1], S_IREAD) < 0)
            perror(argv[1]);
}
```

See Also

`stat(), creat()`

Return Value

Zero is returned on success, negative one (-1) on failure.

Note

*Not all bits may be changed under all operating systems, e.g. neither DOS nor CP/M permit a file to be made unreadable, thus even if **`mode`** does not include `S_IREAD` the file will still be readable (and `stat()` will still return `S_IREAD` in flags).*
This routine is not usable in a ROM based system.

CLOSE

Synopsis

```
#include <unixio.h>

int close (int fd)
```

Description

This routine closes the file associated with the file descriptor **fd**, which will have been previously obtained from a call to either functions, `open()` or `creat()`.

Example

```
#include <unixio.h>
#include <stdio.h>

    /* create an empty file */

void
main (int argc, char ** argv)
{
    int fd;

    if(argc > 1) {
        if((fd = creat(argv[1], 0600)) < 0)
            perror(argv[1]);
        else
            close(fd);
    }
}
```

See Also

`open(), read(), write(), seek()`

Return Value

Returns zero for a successful close, or negative one (-1) otherwise.

Note

This routine is not usable in a ROM based system.

CLRERR, CLREOF

Synopsis

```
#include <stdio.h>

void clrerr (FILE * stream)
void clreof (FILE * stream)
```

Description

These are macros, defined in **`stdio.h`**, which reset the error and end of file flags respectively for the specified stream. They should be used with care; the major valid use is for clearing an end-of-file status on input from a terminal like device, where it may be valid to continue to read after having seen an end-of-file indication. If a **`clreof()`** is not done, then repeated reads will continue to return `EOF`.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char buf[80];

    for(;;) {
        if(!gets(buf)) {
            printf("EOF seen\n");
            clreof(stdin);
        } else if(strcmp(buf, "quit") == 0)
        break;
    }
}
```

See Also

`fopen(), fclose()`

Note

This routine is not usable in a ROM based system.

COS

Synopsis

```
#include <math.h>

double cos (double f)
```

Description

This function yields the cosine of its argument, which is an angle in radians. The cosine is calculated by expansion of a polynomial series approximation.

Example

```
#include <math.h>
#include <stdio.h>

#define C 3.141592/180.0

void
main (void)
{
    double i;

    for(i = 0 ; i <= 180.0 ; i += 10)
        printf("sin(%3.0f) = %f, cos = %f\n", i, sin(i*C), cos(i*C));
}
```

See Also

`sin(), tan(), asin(), acos(), atan(), atan2()`

Return Value

A double in the range -1 to +1.

COSH, SINH, TANH

Synopsis

```
#include <math.h>

double cosh (double f)
double sinh (double f)
double tanh (double f)
```

Description

These functions are the hyperbolic implementations of the trigonometric functions; `cos()`, `sin()` and `tan()`.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f\n", cosh(1.5));
    printf("%f\n", sinh(1.5));
    printf("%f\n", tanh(1.5));
}
```

Return Value

The function **`cosh()`** returns the hyperbolic cosine value.
The function **`sinh()`** returns the hyperbolic sine value.
The function **`tanh()`** returns the hyperbolic tangent value.

CPUTS

Synopsis

```
#include <conio.h>

void cputs (const char * s)
```

Description

The **`cputs()`** function writes its argument string to the console, outputting *carriage returns* before each *newline* in the string. It calls `putch()` repeatedly. On a hosted system **`cputs()`** differs from `puts()` in that it reads the console directly, rather than using file I/O. In an embedded system **`cputs()`** and `puts()` are equivalent.

Example

```
#include <conio.h>
#include <string.h>

char buffer[80];

void
main (void)
{
    for(;;) {
        cgets(buffer);
        if(strcmp(buffer, "exit") == 0)
            break;
        cputs("Type ’exit’ to finish\n");
    }
}
```

See Also

`cputs(), puts(), putch()`

10

CREAT

Synopsis

```
#include <stat.h>
#include <unixio.h>

int creat (const char * name, int mode)
```

Description

This routine attempts to create the file named by **name**. If the file exists and is writeable, it will be removed and re-created. The return value is negative one (-1) if the create failed, or a small non-negative number if it succeeded. This number is a valuable token which must be used to write to or close the file subsequently. The argument **mode** is used to initialize the attributes of the created file. The allowable bits are the same as for `chmod()`, but for Unix compatibility it is recommended that a mode of `0666` or `0600` be used. Under CP/M the mode is ignored - the only way to set a file's attributes is via the `chmod()` function.

Example

```
#include <unixio.h>
#include <stat.h>
#include <stdio.h>

void
main (int argc, char ** argv)
{
    int fd;

    if(argc > 1) {
        if((fd = creat(argv[1], 0600)) < 0)
            perror(argv[1]);
        else
            close(fd);
        }
}
```

See Also

`open(), close(), read(), write(), seek(), stat(), chmod()`

Return Value

If successful it will return a small non negative number. If failed will return negative one (-1).

Note

This routine is not usable in a ROM based system.

CTIME

Synopsis

```
#include <time.h>

char * ctime (time_t * t)
```

Description

The **`ctime()`** function converts the time in seconds pointed to by its argument to a string of the same form as described for `asctime()`. Thus the example program prints the current time and date.

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;

    time(&clock);
    printf("%s", ctime(&clock));
}
```

See Also

`gmtime(), localtime(), asctime(), time()`

Return Value

A pointer to the string.

Note

The example will require the user to provide the `time()` routine as one cannot be supplied with the compiler. See `time()` for more detail.

Data Types

`typedef long time_t;`

DI, EI

Synopsis

```
#include <intrpt.h>

void ei (void)
void di (void)
```

Description

The **`ei()`** and **`di()`** routines enable and disable interrupts respectively. These are implemented as macros defined in **`intrpt.h`**. On most processors they will expand to an in-line assembler instruction that sets or clears the interrupt enable or mask bit.

The example shows the use of **`ei()`** and **`di()`** around access to a long variable that is modified during an interrupt. If this was not done, it would be possible to return an incorrect value, if the interrupt occurred between accesses to successive words of the count value.

Example

```
#include <intrpt.h>

long count;

void
interrupt tick (void)
{
    count++;
}

long
getticks (void)
{
    long val;    /* Disable interrupts around access
        to count, to ensure consistency.*/
    di();
    val = count;
    ei();
    return val;
}
```

10

DIV

Synopsis

```
#include <stdlib.h>

div_t div (int numer, int demon)
```

Description

The **div()** function computes the quotient and remainder of the numerator divided by the denominator.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    div_t x;

    x = div(12345, 66);
    printf("quotient = %d, remainder = %d\n", x.quot, x.rem);
}
```

Return Value

Returns the quotient and remainder into the **div_t** structure.

Data Types

```
typedef struct
{
     int quot;
     int rem;
}  div_t;
```

DUP

Synopsis

```
#include <unixio.h>

int dup (int fd)
```

Description

Given a file descriptor, such as returned by `open()`, this routine will return another file descriptor which will refer to the same open file.

Example

```
#include <stdio.h>
#include <unixio.h>
#include <stdlib.h>
#include <sys.h>

void
main (int argc, char ** argv)
{
    FILE * fp;

    if(argc < 3) {
        fprintf(stderr, "Usage: fd2 stderr_file command ...\n");
        exit(1);
    }
    if(!(fp = fopen(argv[1], "w"))) {
        perror(argv[1]);
        exit(1);
    }
    close(2);
    dup(fileno(fp));        /* make stderr reference file */
    fclose(fp);
    spawnvp(argv[2], argv+2);
    close(2);
}
```

See Also

`open(), close(), creat(), read(), write()`

Return Value

Negative one (-1) is returned if the **`fd`** argument is a bad descriptor or does not refer to an open file.

Note

This routine is not usable in a ROM based system.

EVAL_POLY

Synopsis

```
#include <math.h>

double eval_poly (double x, const double * d, int n)
```

Description

The **eval_poly()** function evaluates a polynomial, whose coefficients are contained in the array **d**, at **x**, for example:

y = x*x*d2 + x*d1 + d0.

The order of the polynomial is passed in **n**.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    double x, y;
    double d[3] = {1.1, 3.5, 2.7};

    x = 2.2;
    y = eval_poly(x, d, 2);
    printf("The polynomial evaluated at %f is %f\n", x, y);
}
```

Return Value

A double value, being the polynomial evaluated at **x**.

EXECL, EXECV

Synopsis

```
#include <sys.h>

int execl (char * name, pname, ...)
int execv (char * name, ppname)
```

Description

The functions load and execute the program specified by the string **`name`**. The **`execl()`** routine takes the arguments for the program from the zero-terminated list of string arguments. The **`execv()`** function is passed a pointer to an array of strings. The array must be zero-terminated. If the named program is found and can be read, the call does not return. Thus any return from these routines may be treated as an error.

Example

```
#include <cpm.h>
#include <stdio.h>
#include <sys.h>

void
main (void)
{
    execl("tst.com", "tst", "argument 1", 0);
    perror("tst.com");
}
```

See Also

`spawnl(), spawnv(), system()`

Note

*The second argument to **`execl()`** or the first string in the array of strings passed to **`execv()`** is nominally argv[0] in the executed program. However, since CP/M has no way of passing the program name, this string will be unused. It must however be present as a placeholder.*
This routine is not usable in a ROM based system.

EXIT

Synopsis

```
#include <stdlib.h>

void exit (int status)
```

Description

This call will close all open files and exit from the program. On CP/M, this means a return to CCP level, under MS-DOS a return to the DOS prompt or the program which spawned the current program. The value **`status`** is used as the exit value of the program. This is recovered under DOS with the wait for status DOS call. The status value will be stored on CP/M at 80H. In an embedded system **`exit()`** normally restarts the program as though a hardware reset had occurred. This call will never return.

Example

```
#include <stdlib.h>

void
main (void)
{
    exit(0);
}
```

See Also

`atexit()`

Return Value

Never returns.

Note

This routine is not usable in a ROM based system.

EXP

Synopsis

```
#include <math.h>

double exp (double f)
```

Description

The **`exp()`** routine returns the exponential function of its argument, i.e. `e` to the power of **`f`**.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    for(f = 0.0 ; f <= 5 ; f += 1.0)
        printf("e to %1.0f = %f\n", f, exp(f));
}
```

See Also

`log(), log10(), pow()`

FABS

Synopsis

```
#include <math.h>

double fabs (double f)
```

Description

This routine returns the absolute value of its double argument.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f %f\n", fabs(1.5), fabs(-1.5));
}
```

See Also

abs()

FCLOSE

Synopsis

```
#include <stdio.h>

int fclose (FILE * stream)
```

Description

This routine closes the specified I/O stream. The value **`stream`** should be a token returned by a previous call to `fopen()`.

Example

```
#include <stdio.h>

void
main (int argc, char ** argv)
{
    FILE * fp;

    if(argc > 1) {
        if(!(fp = fopen(argv[1], "r")))
            perror(argv[1]);
        else {
            fprintf(stderr, "Opened %s\n", argv[1]);
            fclose(fp);
        }
    }
}
```

See Also

`fopen(), fread(), fwrite()`

Return Value

Zero is returned on a successful close, `EOF` otherwise.

Note

This routine is not usable in a ROM based system.

FDOPEN

Synopsis

```
#include <stdio.h>

FILE * fdopen (int fd, const char * mode)
```

Description

Where it is desired to associate a *stdio stream* with a low-level file descriptor that already refers to an open file, this function may be used. It will return a pointer to a **FILE** structure which references the specified low-level file descriptor, as though the function `fopen()` had been called. The **mode** argument is the same as for `fopen()`.

Example

```
#include <stdio.h>

void
main (void)
{
    FILE * fp;

    /* 3 is the device AUX */

    fp = fdopen(3, "w");
    fprintf(fp, "AUX test line\n");
}
```

See Also

`fopen(), freopen(), close()`

Return Value

`NULL` is returned if a **FILE** structure could not be allocated

Note

This routine is not usable in a ROM based system.

FEOF, FERROR

Synopsis

```
#include <stdio.h>

int feof (FILE * stream)
int ferror (FILE * stream)
```

Description

These macros test the status of the *end-of-file* and *error* bits respectively for the specified stream. Each will be true if the corresponding flag is set. The macros are defined in **`stdio.h`**. The value **`stream`** must be a token returned by a previous `fopen()` call.

Example

```
#include <stdio.h>

void
main (void)
{
    while(!feof(stdin))
        getchar();
}
```

See Also

`fopen(), fclose()`

Return Value

The **`feof()`** function returns non-zero if the end-of-file indicator is set for **`stream`**.
The function **`ferror()`** returns non zero if the error indicator is set for **`stream`**.

Note

This routine is not usable in a ROM based system.

FFLUSH

Synopsis

```
#include <stdio.h>

int fflush (FILE * stream)
```

Description

The **fflush()** function will output to the disk file or other device currently open on the specified **stream** the contents of the associated buffer. This is typically used for flushing buffered standard output in interactive applications. Normally stdout is opened in line buffered mode, and is flushed before any input is done on a *stdio stream*, but if input is to be done via console I/O routines, it may be necessary to call **fflush()** first.

Example

```
#include <stdio.h>
#include <conio.h>

void
main (void)
{
    printf("press a key: ");
    fflush(stdout);
    getch();
}
```

See Also

fopen(), fclose()

Note

This routine is not usable in a ROM based system.

FGETC

Synopsis

```
#include <stdio.h>

int fgetc (FILE * stream)
```

Description

The **`fgetc()`** function returns the next character from the input stream. If end-of-file is encountered `EOF` will be returned. It is for this reason that the function is declared as an integer. The integer `EOF` is not a valid byte, thus end-of-file is distinguishable from reading a byte, of all 1 bits, from the file. The routine **`fgetc()`** is the non-macro version of `getc()`.

Example

```
#include <stdio.h>

void
main (void)
{
    FILE * fp;
    int c;

    fp = fopen( "file.dat", "r" );
    if( fp != NULL ) {
        while( (c = fgetc( fp )) != EOF )
            fputc( c, stdout );
        fclose( fp );
    }
}
```

See Also

`fopen(), fclose(), fputc(), getc(), putc()`

Return Value

A character from the input **`stream`**, or `EOF` on end-of-file.

Note

This routine is not usable in a ROM based system.

FGETS

Synopsis

```
#include <stdio.h>

char * fgets (char * s, int n, FILE * stream)
```

Description

The **`fgets()`** function places in the buffer **`s`** up to **`n-1`** characters from the input **`stream`**. If a *newline* is seen in the input before the correct number of characters is read, then **`fgets()`** will return immediately. The *newline* will be left in the buffer. The buffer will be null terminated in any case.

Example

```
#include <stdio.h>

void
main (void)
{
    char buffer[128];

    while(fgets(buffer, sizeof buffer, stdin))
        fputs(buffer, stdout);
}
```

Return Value

A successful **`fgets()`** will return its first argument; `NULL` is returned on end-of-file or error.

Note

This routine is not usable in a ROM based system.

FILENO

Synopsis

```
#include <stdio.h>

int fileno (FILE * stream)
```

Description

The **`fileno()`** routine is a macro from **`stdio.h`** which yields the file descriptor associated with **`stream`**. It is mainly used when it is desired to perform some low-level operation on a file, opened as a *stdio stream*.

Example

```
#include <stdio.h>
#include <unixio.h>
#include <stdlib.h>
#include <sys.h>

void
main (int argc, char ** argv)
{
    FILE * fp;

    if(argc < 3) {
        fprintf(stderr, "Usage: fd2 stderr_file command ...\n");
        exit(1);
    }
    if(!(fp = fopen(argv[1], "w"))) {
        perror(argv[1]);
        exit(1);
    }
    close(2);
    dup(fileno(fp));        /* make stderr reference file */
    fclose(fp);
    spawnvp(argv[2], argv+2);
    close(2);
}
```

See Also

`fopen(), fclose(), open(), close(), dup()`

Note

This routine is not usable in a ROM based system.

FLOOR

Synopsis

```
#include <math.h>

double floor (double f)
```

Description

This routine returns the largest whole number not greater than **f**.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f\n", floor( 1.5 ));
    printf("%f\n", floor( -1.5));
}
```

FOPEN

Synopsis

```
#include <stdio.h>

FILE * fopen (const char * name, const char * mode)
```

Description

The **`fopen()`** function attempts to open a file for reading or writing (or both) according to the **`mode`** string supplied. The mode string is interpreted as follows:

r The file is opened for reading if it exists. If the file does not exist the call fails.

r+ If the file exists it is opened for reading and writing. If the file does not already exist the call fails.

w The file is created if it does not exist, or truncated if it does. It is then opened for writing.

w+ The file is created if it does not already exist, or truncated if it does. The file is opened for reading and writing.

a The file is created if it does not already exist, and opened for writing. All writes will be dynamically forced to the end of the file, thus this mode is known as *append* mode.

a+ The file is created if it does not already exist, and opened for reading and writing. All writes to the file will be dynamically forced to the end of the file, i.e. while any portion of the file may be read, all writes will take place at the end of the file and will not overwrite any existing data. Calling `fseek()` in an attempt to write at any other place in the file will not be effective.

The "**b**" modifier may be appended to any of the above modes, e.g. "**r+b**" or "**rb+**" are equivalent. Adding the "**b**" modifier will cause the file to be opened in binary rather than ASCII mode. Opening in ASCII mode ensures that text files are read in a manner compatible with the Unix-derived conventions for C programs, i.e. that text files contain lines delimited by *newline* characters. The special treatment of read or written characters varies with the operating system, but includes some or all of the following:

NEWLINE (LINE FEED) - Converted to *carriage return*, *line feed* on output.

RETURN - Ignored on input, inserted before *newline* on output.

CTRL-Z - Signals end-of-file on input, appended on `fclose()` on output if necessary on CP/M.

Opening a file in binary mode will allow each character to be read just as written, but because the exact size of a file is not known to CP/M, the file may contain more bytes than were written to it. See `open()` for a description of what constitutes a file name.

When using one of the read/write modes (with a ’`+`’ character in the string), although they permit reading and writing on the same stream, it is not possible to arbitrarily mix input and output calls to the same stream. At any given time a stream opened with a "`+`" mode will be in either an input or output state. The state may only be changed when the associated buffer is empty, which is only guaranteed immediately after a call to `fflush()` or one of the file positioning functions `fseek()` or `rewind()`. The buffer will also be empty after encountering `EOF` while reading a binary stream. It is recommended that an explicit call to `fflush()` be used to ensure this situation. Thus after reading from a stream you should call `fflush()` or `fseek()` before attempting to write on that stream, and vice versa.

Example

```
#include <stdio.h>

void
main (int argc, char ** argv)
{
    FILE * fp;

    if(argc > 1) {
        if(!(fp = fopen(argv[1], "r")))
            perror(argv[1]);
        else {
            fprintf(stderr, "Opened %s\n", argv[1]);
            fclose(fp);
        }
    }
}
```

See Also

`fclose(), fgetc(), fputc(), freopen(), fread(), fflush(), fwrite(), fseek()`

Return Value

The **`fopen()`** routine returns a file pointer to the object, if it fails it will return a null pointer.

Note

This routine is not usable in a ROM based system.

FPRINTF, VFPRINTF

Synopsis

```
#include <stdio.h>

int fprintf (FILE * stream, const char * fmt, ...)

#include <stdarg.h>
#include <stdio.h>

int vfprintf (FILE * stream, const char * fmt, va_list va_arg)
```

Description

The **`fprintf()`** function performs formatted printing on the specified **`stream`**. Refer to `printf()` for the details of the available formats. The **`vfprintf()`** function is similar to **`fprintf()`** but takes a variable argument list pointer rather than a list of arguments. See the description of the **`va_start()`** function for more information on variable argument lists.

Example

```
#include <stdio.h>

void
main (int argc, char ** argv)
{
    FILE * fp;

    if(argc > 1) {
        if(!(fp = fopen(argv[1], "r")))
            perror(argv[1]);
        else {
            fprintf(stderr, "Opened %s\n", argv[1]);
            fclose(fp);
        }
    }
}
```

See Also

`printf(), fscanf(), sscanf(), sprintf()`

Return Value

It returns the number of characters transmitted or a negative number if it fails.

Note

This routine is not usable in a ROM based system.

FPUTC

Synopsis

```
#include <stdio.h>

int fputc (int c, FILE * stream)
```

Description

The character **c** is written to the supplied **stream**. This is the non-macro version of `putc()`.

Example

```
#include <stdio.h>

void
main (void)
{
    FILE * fp;
    int c;

    fp = fopen( "file.dat", "r" );
    if( fp != NULL ) {
        while( (c = fgetc( fp )) != EOF )
            fputc( c, stdout );
        fclose( fp );
    }
}
```

See Also

`putc(), fgetc(), fopen(), fflush()`

Return Value

The character is returned if it was successfully written, `EOF` is returned otherwise.

Note

Note that "written to the stream" may mean only placing the character in the buffer associated with the stream.
This routine is not usable in a ROM based system.

FPUTS

Synopsis

```
#include <stdio.h>

int fputs (const char * s, FILE * stream)
```

Description

The null terminated string **`s`** is written to the **`stream`**. No *newline* is appended (cf. `puts()`).

Example

```
#include <stdio.h>

void
main (void)
{
    fputs("This is a line\n", stdout);
}
```

See Also

`puts(), fgets(), fopen(), fclose()`

Return Value

The return value is zero for success, `EOF` for error.

Note

This routine is not usable in a ROM based system.

FREAD

Synopsis

```
#include <stdio.h>

int fread (void * buf, size_t size, size_t cnt, FILE * stream)
```

Description

The **fread()** function will read up to **cnt** objects, each of length **size**, into memory at **buf** from the **stream**. No word alignment in the **stream** is assumed or necessary. The read is done via successive getc()'s.

Example

```
#include <stdio.h>

void
main (void)
{
    char buf[80];
    int i;

    i = fread(buf, 1, sizeof(buf), stdin);
    printf("Read %d bytes\n", i);
}
```

See Also

fwrite(), fopen(), fclose(), getc()

Return Value

The return value is the number of objects read. If none is read, zero will be returned. Note that a return value less than **cnt**, but greater than zero, may not represent an error (cf. fwrite()).

Note

This routine is not usable in a ROM based system.

FREE

Synopsis

```
#include <stdlib.h>
void free (void * ptr)
```

Description

The **`free()`** function deallocates the block of memory at **`ptr`**, which must have been obtained from a call to `malloc()` or `calloc()`.

Example

```
#include <stdlib.h>
#include <stdio.h>

struct test {
    int a[20];
} * ptr;

        /* Allocate space for 20 structures. */
void
main (void)
{
    ptr = calloc(20, sizeof(struct test));
    if(!ptr)
        printf("Failed\n");
    else
        free(ptr);
}
```

See Also

`malloc(), calloc()`

FREOPEN

Synopsis

```
#include <stdio.h>

FILE * freopen (const char * name, const char * mode, FILE * stream)
```

Description

The **`freopen()`** function closes the given **stream** (if open) then re-opens the **`stream`** attached to the file, described by **`name`**. The mode of opening is given by **`mode`**. This function is commonly used to attach `stdin` or `stdout` to a file, as in the following example.

Example

```
#include <stdio.h>

void
main (void)
{
    char buf[80];

    if(!freopen("test.fil", "r", stdin))
        perror("test.fil");
    if(gets(buf))
        puts(buf);
}
```

See Also

`fopen(), fclose()`

Return Value

It either returns the **`stream`** argument, if successful, or `NULL` if not.
See `fopen()` for more information.

Note

This routine is not usable in a ROM based system.

FREXP

Synopsis

```
#include <math.h>

double frexp (double f, int * p)
```

Description

The **`frexp()`** function breaks a floating point number into a normalized fraction and an integral power of 2. The integer is stored into the **`int`** object pointed to by **`p`**. Its return value `x` is in the interval [0.5, 1.0) or zero, and **`f`** equals `x` times 2 raised to the power stored in **`*p`**. If **`f`** is zero, both parts of the result are zero.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;
    int i;

    f = frexp(23456.34, &i);
    printf("23456.34 = %f * 2^%d\n", f, i);
}
```

See Also

`ldexp()`

FSCANF, VFSCANF

Synopsis

```
#include <stdio.h>

int fscanf (FILE * stream, const char * fmt, ...)

#include <stdarg.h>
#include <stdio.h>

int vfscanf (FILE * stream, const char * fmt, va_list va_arg)
```

Description

This routine performs formatted input from the specified **`stream`**. See `scanf()` for a full description of the behaviour of the routine.
The **`vfscanf()`** function is similar to **`fscanf()`** but takes a variable list pointer rather than a list of arguments. See the description of `va_start()` for more information on variable argument lists.

Example

```
#include <stdio.h>

void
main (void)
{
    int i;

    printf("Enter a number: ");
    fscanf(stdin, "%d", &i);
    printf("Read %d\n", i);
}
```

See Also

`scanf(), sscanf(), fopen(), fclose()`

Return Value

The number of values assigned, or `EOF` if an error occurred and no items were converted.

Note

This routine is not usable in a ROM based system.

FSEEK

Synopsis

```
#include <stdio.h>

int fseek (FILE * stream, long offs, int wh)
```

Description

The **fseek()** function positions the "file pointer" (i.e. a pointer to the next character to be read or written) of the specified **stream** as follows:

wh	resultant location
0	offs
1	offs+previous location
2	offs+length of file

It should be noted that **offs** is a signed value. Thus the 3 allowed modes give positioning relative to the beginning of the file, the current file pointer and the end of the file respectively. Note however that positioning beyond the end of the file is legal, but will result in an end-of-file indication if an attempt is made to read data there. It is quite in order to write data beyond the previous end-of-file. The **fseek()** function correctly accounts for any buffered data. The current file position can be determined with the function `ftell()`.

Example

```
#include <stdio.h>
#include <stdlib.h>

void
main (void)
{
    FILE * fp;

            /* open file for read/write */
    fp = fopen("test.fil", "r+");
    if(!fp)
        exit(1);
    fseek(fp,0L, 2);        /* seek to end */
    fputs("Another line!\n", fp);
    fclose(fp);
}
```

See Also

`lseek(), fopen(), fclose(), ftell()`

Return Value

`EOF` is returned if the positioning request could not be satisfied, otherwise zero.

Note

This routine is not usable in a ROM based system.

FTELL

Synopsis

```
#include <stdio.h>
long ftell (FILE * stream)
```

Description

This function returns the current position of the conceptual read/write pointer associated with **`stream`**. This is the position relative to the beginning of the file of the next byte to be read from or written to the file.

Example

```
#include <stdio.h>
#include <stdlib.h>

void
main (void)
{
    FILE * fp;

    fp = fopen("test.fil", "r");
    if(!fp)
        exit(1);
    fseek(fp, 0L, 2);       /* seek to end */
    printf("size = %ld\n", ftell(fp));
}
```

See Also

`fseek()`

Return Value

A pointer to the current byte read/write position in the given stream.

Note

This routine is not usable in a ROM based system.

FWRITE

Synopsis

`#include <stdio.h>`

`int fwrite (const void * buf, size_t size, size_t cnt, FILE * stream)`

Description

The **`fwrite()`** function accepts **`cnt`** objects of length **`size`** bytes to be written from memory at **`buf`**, to the specified **`stream`**.

Example

```
#include <stdio.h>

void
main (void)
{
    if(fwrite("a test string\n", 1, 14, stdout) != 14)
        fprintf(stderr, "Fwrite failed\n");
}
```

See Also

`fread(), fopen(), fclose()`

Return Value

The number of whole objects written will be returned, or zero if none could be written. Any return value not equal to **`cnt`** should be treated as an error (cf. `fread()`).

Note

This routine is not usable in a ROM based system.

GETC

Synopsis

```
#include <stdio.h>

int getc (FILE * stream)
```

Description

One character is read from the specified stream and returned. This is the macro version of `fgetc()`, and is defined in **`stdio.h`**.

Example

```
#include <stdio.h>

void
main (void)
{
    int i;

    while((i = getc(stdin)) != EOF)
        putchar(i);
}
```

Return Value

`EOF` will be returned on end-of-file or error.

Note

This routine is not usable in a ROM based system.

GETCH, GETCHE

Synopsis

```
#include <conio.h>

char getch (void)
char getche (void)
```

Description

The **`getch()`** function reads a single character from the console keyboard and returns it without echoing. The **`getche()`** function is similar but does echo the character typed.

In an embedded system, the source of characters is defined by the particular routines supplied. By default, the library contains a version of **`getch()`** that will interface to the Lucifer Debugger. The user should supply an appropriate routine if another source is desired, e.g. a serial port.

The module *getch.c* in the SOURCES directory contains model versions of all the console I/O routines. Other modules may also be supplied, e.g. *ser180.c* has routines for the serial port in a Z180.

Example

```
#include <conio.h>

void
main (void)
{
    char c;

    while((c = getche()) != '\n')
        continue;
}
```

See Also

`cgets(), cputs(), ungetch()`

10

GETCHAR

Synopsis

```
#include <stdio.h>

int getchar (void)
```

Description

The **`getchar()`** routine is a `getc(stdin)` operation. It is a macro defined in **`stdio.h`**. Note that under normal circumstances **`getchar()`** will NOT return unless a *carriage return* has been typed on the console. To get a single character immediately from the console, use the function `getch()`.

Example

```
#include <stdio.h>

void
main (void)
{
    int c;

    while((c = getchar()) != EOF)
        putchar(c);
}
```

See Also

`getc(), fgetc(), freopen(), fclose()`

Note

This routine is not usable in a ROM based system.

GETENV

Synopsis

```
#include <stdlib.h>

char * getenv (const char * s)
extern char ** environ
```

Description

The **`getenv()`** function will search the vector of environment strings for one matching the argument supplied, and return the value part of that environment string. For example, if the environment contains the string:

COMSPEC=C:\COMMAND.COM

Thus executing the routine **`getenv("COMSPEC")`** will return *C:\COMMAND.COM*. The global variable **`environ`** is a pointer to an array of pointers to environment strings, terminated by a null pointer. This array is initialized at startup time under MS-DOS from the environment pointer supplied when the program was executed. Under CP/M no such environment is supplied, so the first call to **`getenv()`** will attempt to open a file in the current user number on the current drive called *ENVIRON*. This file should contain definitions for any environment variables desired to be accessible to the program, e.g.

HITECH=0:C:

Each variable definition should be on a separate line, consisting of the variable name (conventionally all in upper case) followed without intervening white space by an equal sign ('=') then the value to be assigned to that variable.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    printf("comspec = %s\n", getenv("COMSPEC"));
}
```

Return Value

`NULL` if the specified variable could not be found.

10

Note

This routine is not usable in a ROM based system.

GETS

Synopsis

```
#include <stdio.h>

char * gets (char * s)
```

Description

The **`gets()`** function reads a line from standard input into the buffer at **`s`**, deleting the *newline* (cf. `fgets()`). The buffer is null terminated. In an embedded system, **`gets()`** is equivalent to `cgets()`, and results in `getche()` being called repeatedly to get characters. Editing (with *backspace*) is available.

Example

```
#include <stdio.h>

void
main (void)
{
    char buf[80];

    printf("Type a line: ");
    if(gets(buf))
        puts(buf);
}
```

See Also

`fgets(), freopen(), puts()`

Return Value

It returns its argument, or `NULL` on end-of-file.

10

GETUID

Synopsis

```
#include <sys.h>

int getuid (void)
```

Description

The **getuid()** function returns the current user number. Under CP/M, the current user number determines the user number associated with an opened or created file, unless overridden by an explicit user number prefix in the file name.

Example

```
#include <stdio.h>
#include <sys.h>

void
main (void)
{
    printf("Current user number is: %d\n", getuid());
}
```

See Also

setuid(), open()

Note

This routine is not usable in a ROM based system.

GETW

Synopsis

```
#include <stdio.h>

int getw (FILE * stream)
```

Description

The **`getw()`** function returns one word (16 bits for the Z80 and 8086) from the nominated **`stream`**. `EOF` is returned on end-of-file, but since this is a perfectly good word, the `feof()` macro should be used for testing for `EOF`. When reading the word, no special alignment in the file is necessary, as the read is done by two consecutive `getc()`'s. The byte ordering is however undefined. The word read should in general have been written by `putw()`. Do not rely on this function to read binary data written by another program.

Example

```
#include <stdio.h>

void
main (void)
{
    FILE * fp;
    int c;

    fp = fopen("file.dat", "r");
    if(!feof(fp)) {
        while((c = getw(fp)) != EOF)
            putw(c, stdout);
        fclose(fp);
    }
}
```

See Also

`putw(), getc(), fopen(), fclose()`

Note

This routine is not usable in a ROM based system.

10

GMTIME

Synopsis

```
#include <time.h>

struct tm * gmtime (time_t * t)
```

Description

This function converts the time pointed to by **`t`** which is in seconds since 00:00:00 on Jan 1, 1970, into a broken down time stored in a structure as defined in **`time.h`**. The structure is defined in the 'Data Types' section.

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;
    struct tm * tp;

    time(&clock);
    tp = gmtime(&clock);
    printf("It's %d in London\n", tp->tm_year+1900);
}
```

See Also

`ctime(), asctime(), time(), localtime()`

Return Value

Returns a structure of type **`tm`**.

Note

The example will require the user to provide the `time()` routine as one cannot be supplied with the compiler. See `time()` for more detail.

Data Types

```
typedef long time_t;
struct tm {
     int    tm_sec;
     int    tm_min;
     int    tm_hour;
     int    tm_mday;
     int    tm_mon;
     int    tm_year;
     int    tm_wday;
     int    tm_yday;
     int    tm_isdst;
};
```

IM

Synopsis

```
#include <intrpt.h>

void im (unsigned char mode)
```

Description

This function allows setting of the interrupt mode on a Z80 processor. The argument to **`im()`** is the mode number, 0, 1 or 2. If the mode is set to 2, then as a side effect, this routine will set the I register to point to the base of the interrupt vector table.

Example

```
#include <intrpt.h>

void
main (void)
{
    im(2);

     .
     .
     .
     .
}
```

See Also

`ROM_VECTOR(), RAM_VECTOR(), CHANGE_VECTOR(), set_vector()`

ISALNUM, ISALPHA, ISDIGIT, ISLOWER et. al.

Synopsis

```
#include <ctype.h>

int isalnum (char c)
int isalpha (char c)
int isascii (char c)
int iscntrl (char c)
int isdigit (char c)
int islower (char c)
int isprint (char c)
int isgraph (char c)
int ispunct (char c)
int isspace (char c)
int isupper (char c)
int isxdigit(char c)
```

Description

These macros, defined in **`ctype.h`**, test the supplied character for membership in one of several overlapping groups of characters. Note that all except **`isascii()`** are defined for **`c`**, if **`isascii(c)`** is true or if c = EOF.

isalnum	**(c)**	c is in 0-9 or a-z or A-Z
isalpha	**(c)**	c is in A-Z or a-z
isascii	**(c)**	c is a 7 bit ascii character
iscntrl	**(c)**	c is a control character
isdigit	**(c)**	c is a decimal digit
islower	**(c)**	c is in a-z
isprint	**(c)**	c is a printing char
isgraph	**(c)**	c is a non-space printable character
ispunct	**(c)**	c is not alphanumeric
isspace	**(c)**	c is a space, tab or newline
isupper	**(c)**	c is in A-Z
isxdigit	**(c)**	c is in 0-9 or a-f or A-F

Example

```
#include <ctype.h>
#include <stdio.h>

void
```

```
main (void)
{
    char buf[80];
    int i;

    gets(buf);
    i = 0;
    while(isalnum(buf[i]))
        i++;
    buf[i] = 0;
    printf("'%s' is the word\n", buf);
}
```

See Also

toupper(), tolower(), toascii()

ISATTY

Synopsis

```
#include <unixio.h>

int isatty (int fd)
```

Description

This tests the type of the file associated with **`fd`**. It returns true if the file is attached to a tty-like device. This would normally be used for testing, if standard input is coming from a file or the console. For testing *stdio streams*, use **`isatty(fileno(stream))`**.

Example

```
#include <unixio.h>
#include <stdio.h>

void
main (void)
{
    if(isatty(fileno(stdin)))
        printf("input not redirected\n");
    else
        printf("Input is redirected\n");
}
```

Return Value

Zero if the stream is associated with a file; one if it is associated with the console or other keyboard type device.

Note

This routine is not usable in a ROM based system.

KBHIT

Synopsis

```
#include <conio.h>

int kbhit (void)
```

Description

This function returns 1 if a character has been pressed on the console keyboard, 0 otherwise. Normally the character would then be read via `getch()`.

Example

```
#include <conio.h>

void
main (void)
{
    int i;

    while(!kbhit()) {
        cputs("I’m waiting..");
        for(i = 0 ; i != 1000 ; i++)
            continue;
    }
}
```

See Also

`getch()`, `getche()`

Return Value

Returns one if a character has been pressed on the console keyboard, zero otherwise.

LDEXP

Synopsis

```
#include <math.h>

double ldexp (double f, int i)
```

Description

The **`ldexp()`** function performs the inverse of `frexp()` operation; the integer **`i`** is added to the exponent of the floating point **`f`** and the resultant returned.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    f = ldexp(1.0, 10);
    printf("1.0 * 2^10 = %f\n", f);
}
```

See Also

`frexp()`

Return Value

The return value is the integer **`i`** added to the exponent of the floating point value **`f`**.

LDIV

Synopsis

```
#include <stdlib.h>

ldiv_t ldiv (long number, long denom)
```

Description

The **ldiv()** routine divides the numerator by the denominator, computing the quotient and the remainder. The sign of the quotient is the same as that of the mathematical quotient. Its absolute value is the largest integer which is less than the absolute value of the mathematical quotient.

The **ldiv()** function is similar to the `div()` function, the difference being that the arguments and the members of the returned structure are all of type **long int**.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    ldiv_t lt;

    lt = ldiv(1234567, 12345);
    printf("Quotient = %ld, remainder = %ld\n", lt.quot, lt.rem);
}
```

See Also

`div()`

Return Value

Returns a structure of type **ldiv_t**

Data Types

```
typedef struct {
     long    quot;  /* quotient */
     long    rem;   /* remainder */
} ldiv_t;
```

LOCALTIME

Synopsis

```
#include <time.h>

struct tm * localtime (time_t * t)
```

Description

The **`localtime()`** function converts the time pointed to by **t** which is in seconds since 00:00:00 on Jan 1, 1970, into a broken down time stored in a structure as defined in **`time.h`**. The routine **`localtime()`** takes into account the contents of the global integer `time_zone`. This should contain the number of minutes that the local time zone is *westward* of Greenwich. Since there is no way under MS-DOS of actually predetermining this value, by default **`localtime()`** will return the same result as **`gmtime()`**.

Example

```
#include <stdio.h>
#include <time.h>

char * wday[] = {
    "Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday"
};

void
main (void)
{
    time_t clock;
    struct tm * tp;

    time(&clock);
    tp = localtime(&clock);
    printf("Today is %s\n", wday[tp->tm_wday]);
}
```

See Also

`ctime(), asctime(), time()`

Return Value

Returns a structure of type **`tm`**.

Note

The example will require the user to provide the `time()` routine as one cannot be supplied with the compiler. See `time()` for more detail.

Data Types

```
typedef long time_t;
struct tm {
     int    tm_sec;
     int    tm_min;
     int    tm_hour;
     int    tm_mday;
     int    tm_mon;
     int    tm_year;
     int    tm_wday;
     int    tm_yday;
     int    tm_isdst;
};
```

LOG, LOG10

Synopsis

```
#include <math.h>

double log (double f)
double log10 (double f)
```

Description

The **`log()`** function returns the natural logarithm of **`f`**. The function **`log10()`** returns the logarithm to base 10 of **`f`**.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    for(f = 1.0 ; f <= 10.0 ; f += 1.0)
        printf("log(%1.0f) = %f\n", f, log(f));
}
```

See Also

`exp(), pow()`

Return Value

Zero if the argument is negative.

LONGJMP

Synopsis

```
#include <setjmp.h>

void longjmp (jmp_buf buf, int val)
```

Description

The **`longjmp()`** function, in conjunction with `setjmp()`, provides a mechanism for non-local goto's. To use this facility, `setjmp()` should be called with a **`jmp_buf`** argument in some outer level function. The call from `setjmp()` will return 0.

To return to this level of execution, **`lonjmp()`** may be called with the same **`jmp_buf`** argument from an inner level of execution. ***Note*** however that the function which called `setjmp()` must still be active when **`longjmp()`** is called. Breach of this rule will cause disaster, due to the use of a stack containing invalid data. The **`val`** argument to **`longjmp()`** will be the value apparently returned from the `setjmp()`. This should normally be non-zero, to distinguish it from the genuine `setjmp()` call.

Example

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf jb;

void
inner (void)
{
    longjmp(jb, 5);
}

void
main (void)
{
    int i;

    if(i = setjmp(jb)) {
        printf("setjmp returned %d\n", i);
        exit(0);
    }
    printf("setjmp returned 0 - good\n");
    printf("calling inner...\n");
    inner();
    printf("inner returned - bad!\n");
}
```

See Also

`setjmp()`

Return Value

The **`longjmp()`** routine never returns.

Note

*The function which called `setjmp()` must still be active when **`longjmp()`** is called. Breach of this rule will cause disaster, due to the use of a stack containing invalid data.*

LSEEK

Synopsis

```
#include <unixio.h>

long lseek (int fd, long offs, int wh)
```

Description

This function operates in an analogous manner to `fseek()`, however it does so on unbuffered low-level I/O file descriptors, rather than on *stdio streams*. It also returns the resulting pointer location. Thus **`lseek(fd, 0L, 1)`** returns the current pointer location without moving it.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <unixio.h>

void
main (void)
{
    int fd;

    fd = open("test.fil", 1); /* open for write */
    if(fd < 0)
        exit(1);
    lseek(fd,0L, 2);        /* seek to end */
    write(fd, "more stuff\r\n", 12);
    close(fd);
}
```

See Also

`open(), close(), read(), write()`

Return Value

Negative one (-1) is returned on error, the resulting location otherwise.

Note

This routine is not usable in a ROM based system.

MALLOC

Synopsis

```
#include <stdlib.h>

void * malloc (size_t cnt)
```

Description

The **`malloc()`** function attempts to allocate **`cnt`** bytes of memory from the "heap", the dynamic memory allocation area. If successful, it returns a pointer to the block, otherwise zero is returned. The memory so allocated may be freed with `free()`, or changed in size via `realloc()`. The **`malloc()`** routine calls `sbrk()` to obtain memory, and is in turn called by `calloc()`. The **`malloc()`** function does not clear the memory it obtains, unlike `calloc()`.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>


void
main (void)
{
    char * cp;

    cp = malloc(80);
    if(!cp)
        printf("Malloc failed\n");
    else {
        strcpy(cp, "a string");
        printf("block = ’%s’\n", cp);
        free(cp);
    }
}
```

See Also

`calloc(), free(), realloc()`

Return Value

A pointer to the memory if it succeeded; `NULL` otherwise.

MEMCHR

Synopsis

```
#include <string.h>

void * memchr (const void * block, int val, size_t length)
```

Description

The **memchr()** function is similar to strchr() except that instead of searching null terminated strings, it searches a block of memory specified by length for a particular byte. Its arguments are a pointer to the memory to be searched, the value of the byte to be searched for, and the length of the block. A pointer to the first occurrence of that byte in the block is returned.

Example

```
#include <string.h>
#include <stdio.h>

unsigned int ary[] = {1, 5, 0x6789, 0x23};

void
main (void)
{
    char * cp;

    cp = memchr(ary, 0x89, sizeof ary);
    if(!cp)
        printf("not found\n");
    else
        printf("Found at offset %u\n", cp - (char *)ary);
}
```

See Also

strchr()

Return Value

A pointer to the first byte matching the argument if one exists; NULL otherwise.

MEMCMP

Synopsis

`#include <string.h>`

`int memcmp (const void * s1, const void * s2, size_t n)`

Description

The **`memcmp()`** function compares two blocks of memory, of length **`n`**, and returns a signed value similar to `strncmp()`. Unlike `strncmp()` the comparison does not stop on a null character. The ASCII collating sequence is used for the comparison, but the effect of including non-ASCII characters in the memory blocks on the sense of the return value is indeterminate. Testing for equality is always reliable.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    int buf[10], cow[10], i;

    buf[0] = 1;
    buf[2] = 4;
    cow[0] = 1;
    cow[2] = 5;
    buf[1] = 3;
    cow[1] = 3;
    i = memcmp(buf, cow, 3*sizeof(int));
    if(i < 0)
        printf("less than\n");
    else if(i > 0)
        printf("Greater than\n");
    else
        printf("Equal\n");
}
```

See Also

`strncpy(), strncmp(), strchr(), memset(), memchr()`

Return Value

Returns negative one, zero or one, depending on whether **`s1`** points to string which is less than, equal to or greater than the string pointed to by **`s2`** in the collating sequence.

MEMCPY

Synopsis

```
#include <string.h>

void * memcpy (void * d, const void * s, size_t n)
```

Description

The **`memcpy()`** function copies **`n`** bytes of memory starting from the location pointed to by **`s`** to the block of memory pointed to by **`d`**. The result of copying overlapping blocks is undefined. The **`memcpy()`** function differs from `strcpy()` in that it copies a specified number of bytes, rather than all bytes up to a null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];

    memset(buf, 0, sizeof buf);
    memcpy(buf, "a partial string", 10);
    printf("buf = ’%s’\n", buf);
}
```

See Also

`strncpy(), strncmp(), strchr(), memset()`

Return Value

The **`memcpy()`** routine returns its first argument.

MEMMOVE

Synopsis

```
#include <string.h>

void * memmove (void * s1, const void * s2, size_t n)
```

Description

The **`memmove()`** function is similar to the function `memcpy()` except copying of overlapping blocks is handled correctly. That is, it will copy forwards or backwards as appropriate to correctly copy one block to another that overlaps it.

See Also

`strncpy(), strncmp(), strchr(), memcpy()`

Return Value

The function **`memmove()`** returns its first argument.

MEMSET

Synopsis

```
#include <string.h>

void * memset (void * s, int c, size_t n)
```

Description

The **`memset()`** function fills **`n`** bytes of memory starting at the location pointed to by **`s`** with the byte **`c`**.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char abuf[20];

    strcpy(abuf, "This is a string");
    memset(abuf, ’x’, 5);
    printf("buf = ’%s’\n", abuf);
}
```

See Also

`strncpy(), strncmp(), strchr(), memcpy(), memchr()`

MODF

Synopsis

```
#include <math.h>

double modf (double value, double * iptr)
```

Description

The **modf()** function splits the argument **value** into integral and fractional parts, each having the same sign as **value**. For example, -3.17 would be split into the intergral part (-3) and the fractional part (-0.17).

The integral part is stored as a double in the object pointed to by **iptr**.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double i_val, f_val;

    f_val = modf( -3.17, &i_val);
}
```

Return Value

The signed fractional part of **value**.

OPEN

Synopsis

```
#include <unixio.h>

int open (const char * name, int mode)
```

Description

The **open()** function is the fundamental means of opening files for reading and writing. The file specified by **name** is sought, and if found is opened for reading, writing or both. The variable **mode** is encoded as follows:

Mode	Meaning
0	Open for reading only
1	Open for writing only
2	Open for both reading and writing

The file must already exist - if it does not, `creat()` should be used to create it. On a successful open, a file descriptor is returned. This is a non-negative integer which may be used to refer to the open file subsequently. If the open fails, negative one (-1) is returned. Under MS-DOS the syntax of filenames are standard MS-DOS. The syntax of a CP/M filename is:

```
[uid:][drive:]name.type
```

where **uid** is a decimal number 0 to 15, **drive** is a letter *A* to *P* or *a* to *p*, **name** is 1 to 8 characters and type is 0 to 3 characters. Though there are few inherent restrictions on the characters in the **name** and **type**, it is recommended that they be restricted to the alphanumerics and standard printing characters. Use of strange characters may cause problems in accessing and/or deleting the file.

One or both of **uid:** and **drive:** may be omitted; if both are supplied, the **uid:** must come first. Note that the '**[**' and '**]**' are meta-symbols only. Some examples are:

```
fred.dat
file.c
0:xyz.com
0:a:file1.p
a:file2.
```

If the **uid:** is omitted, the file will be sought with **uid** equal to the current user number, as returned by `getuid()`. If **drive:** is omitted, the file will be sought on the currently selected drive. The following special file names are recognized:

lst:	**Accesses the list device - write only**
pun:	**Accesses the punch device - write only**
rdr:	**Accesses the reader device - read only**
con:	**Accesses the system console - read/write**

File names may be in any case - they are converted to upper case during processing of the name. MS-DOS filenames may be any valid MS-DOS version 2.xx or higher filename, e.g.

```
fred.nrk
A:\HITECH\STDIO.H
```

The special device names (e.g. CON, LST) are also recognized. These do not require (and should not have) a trailing colon.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <unixio.h>

void
main (void)
{
    int fd;

    fd = open("test.fil", 1); /* open for write */
    if(fd < 0)
        exit(1);
    lseek(fd,0L, 2);        /* seek to end */
    write(fd, "more stuff\r\n", 12);
    close(fd);
}
```

See Also

`close(), fopen(), fclose(), read(), write(), creat()`

Return Value

If successful a non negative number (file descriptor), or negative one (-1) if it fails.

Note

This routine is not usable in a ROM based system.

PERROR

Synopsis

```
#include <stdio.h>

void perror (const char * s)
```

Description

This routine will print on the *stderr stream* the argument **`s`**, followed by a descriptive message detailing the last error returned from a DOS system call. The error number is retrieved from the global variable **`errno`**. The **`perror()`** is of limited usefulness under CP/M as it does not give as much error information as under MS-DOS.

Example

```
#include <stdio.h>
#include <stdlib.h>

void
main (int argc, char ** argv)
{
    if(argc < 2)
        exit(0);
    if(!freopen(argv[1], "r", stdin))
        perror(argv[1]);
}
```

See Also

`strerror()`

Note

This routine is not usable in a ROM based system.

PERSIST_CHECK, PERSIST_VALIDATE

Synopsis

```
#include <sys.h>

int persist_check (int flag)
void persist_validate (void)
```

Description

The **`persist_check()`** function is used with non-volatile RAM variables, declared with the `persistent` qualifier. It tests the nvram area, using a magic number stored in a hidden variable by a previous call to **`persist_validate()`** and a checksum also calculated by **`persist_validate()`**. If the magic number and checksum are correct, it returns true (non-zero). If either are incorrect, it returns zero. In this case it will optionally zero out and re-validate the non-volatile RAM area (by calling **`persist_validate()`**). This is done if the flag argument is true.

The **`persist_validate()`** routine should be called after each change to a persistent variable. It will set up the magic number and recalculate the checksum.

Example

```
#include <sys.h>
#include <stdio.h>

persistent long reset_count;

void
main (void)
{
    if(!persist_check(1))
        printf("Reset count invalid - zeroed\n");
    else
        printf("Reset number %ld\n", reset_count);
    reset_count++;          /* update count */
    persist_validate();     /* and checksum */
    for(;;)
        continue;       /* sleep until next reset */
}
```

Return Value

FALSE (zero) if the NV-RAM area is invalid; TRUE (non-zero) if the NVRAM area is valid.

POW

Synopsis

```
#include <math.h>

double pow (double f, double p)
```

Description

The **`pow()`** function raises its first argument, **`f`**, to the power **`p`**.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    for(f = 1.0 ; f <= 10.0 ; f += 1.0)
        printf("pow(2, %1.0f) = %f\n", f, pow(2, f));
}
```

See Also

`log(), log10(), exp()`

Return Value

`f` to the power of **`p`**.

PRINTF, VPRINTF

Synopsis

```
#include <stdio.h>

int printf (const char * fmt, ...)

#include <stdio.h>
#include <stdarg.h>

int vprintf (const char * fmt, va_list va_arg)
```

Description

The **`printf()`** function is a formatted output routine, operating on `stdout`. There are corresponding routines operating on a given stream (`fprintf()`) or into a string buffer (`sprintf()`). The **`printf()`** routine is passed a format string, followed by a list of zero or more arguments. In the format string are conversion specifications, each of which is used to print out one of the argument list values.

Each conversion specification is of the form **`%m.nc`** where the percent symbol **`%`** introduces a conversion, followed by an optional width specification **`m`**. The **`n`** specification is an optional precision specification (introduced by the dot) and **`c`** is a letter specifying the type of the conversion.

A minus sign ('-') preceding **`m`** indicates left rather than right adjustment of the converted value in the field. Where the field width is larger than required for the conversion, blank padding is performed at the left or right as specified. Where right adjustment of a numeric conversion is specified, and the first digit of **`m`** is 0, then padding will be performed with zeroes rather than blanks. For integer formats, the precision indicates a minimum number of digits to be output, with leading zeros inserted to make up this number if required.

A hash character (#) preceding the width indicates that an alternate format is to be used. The nature of the alternate format is discussed below. Not all formats have alternates. In those cases, the presence of the hash character has no effect.

The floating point formats require that the appropriate floating point library is linked. From within HPD this can be forced by selecting the "Float formats in printf" selection in the options menu. From the command line driver, use the option **`-LF`**.

If the character **`*`** is used in place of a decimal constant, e.g. in the format **`%*d`**, then one integer argument will be taken from the list to provide that value. The types of conversion are:

`f`
Floating point - **`m`** is the total width and **`n`** is the number of digits after the decimal point. If **`n`** is omitted it defaults to 6. If the precision is zero, the decimal point will be omitted unless the alternate format is specified.

e
Print the corresponding argument in scientific notation. Otherwise similar to **f**.

g
Use **e** or **f** format, whichever gives maximum precision in minimum width. Any trailing zeros after the decimal point will be removed, and if no digits remain after the decimal point, it will also be removed.

o x X u d
Integer conversion - in radices 8, 16, 16, 10 and 10 respectively. The conversion is signed in the case of **d**, unsigned otherwise. The precision value is the total number of digits to print, and may be used to force leading zeroes. E.g. **%8.4x** will print at least 4 hex digits in an 8 wide field. Preceding the key letter with an **l** indicates that the value argument is a long integer. The letter **X** prints out hexadecimal numbers using the upper case letters *A-F* rather than *a-f* as would be printed when using **x**. When the alternate format is specified, a leading zero will be supplied for the octal format, and a leading `0x` or `0X` for the hex format.

s
Print a string - the value argument is assumed to be a character pointer. At most **n** characters from the string will be printed, in a field **m** characters wide.

c
The argument is assumed to be a single character and is printed literally.

Any other characters used as conversion specifications will be printed. Thus **%%** will produce a single percent sign.

The **vprintf()** function is similar to **printf()** but takes a variable argument list pointer rather than a list of arguments. See the description of `va_start()` for more information on variable argument lists. An example of using **vprintf()** is given below.

Example

```
printf("Total = %4d%%", 23)
       yields 'Total =   23%'

printf("Size is %lx" , size)
       where size is a long, prints size
       as hexadecimal.

printf("Name = %.8s", "a1234567890")
       yields 'Name = a1234567'

printf("xx%*d", 3, 4)
       yields 'xx  4'

/* vprintf example */
```

10

```
#include        <stdio.h>

int
error (char * s, ...)
{
    va_list ap;

    va_start(ap, s);
    printf("Error: ");
    vprintf(s, ap);
    putchar('\n');
    va_end(ap);
}

void
main (void)
{
    int i;

    i = 3;
    error("testing 1 2 %d", i);
}
```

See Also

`fprintf()`, `sprintf()`

Return Value

The **`printf()`** and **`vprintf()`** functions return the number of characters written to `stdout`.

PUTC

Synopsis

```
#include <stdio.h>

int putc (int c, FILE * stream)
```

Description

The **putc()** function is the macro version of fputc() and is defined in **stdio.h**. It places the supplied character onto the specified I/O stream.

Example

```
#include <stdio.h>

char * x = "this is a string";

void
main (void)
{
    char * cp;

    cp = x;
    while(*x)
        putc(*x++, stdout);
    putc('\n', stdout);
}
```

See Also

fputc(), getc(), fopen(), fclose(), putch()

Return Value

The character passed as argument, or on error EOF.

Note

This routine is not usable in a ROM based system.

10

PUTCH

Synopsis

```
#include <conio.h>

void putch (char c)
```

Description

The **putch()** function outputs the character **c** to the console screen, prepending a *carriage return* if the character is a *newline*. In a CP/M or MS-DOS system this will use one of the system I/O calls. In an embedded system this routine, and associated others, will be defined in a hardware dependent way. The standard **putch()** routines in the embedded library interface either to a serial port or to the Lucifer Debugger.

Example

```
#include <conio.h>

char * x = "This is a string";

void
main (void)
{
    char * cp;

    cp = x;
    while(*x)
        putch(*x++);
    putch('\n');
}
```

See Also

cgets(), cputs(), getch(), getche()

PUTCHAR

Synopsis

```
#include <stdio.h>

int putchar (int c)
```

Description

The **`putchar()`** function is a `putc()` operation on `stdout`, defined in **`stdio.h`**.

Example

```
#include <stdio.h>

char * x = "This is a string";

void
main (void)
{
    char * cp;

    cp = x;
    while(*x)
        putchar(*x++);
    putchar('\n');
}
```

See Also

`putc(), getc(), freopen(), fclose()`

Return Value

The character passed as argument, or `EOF` if an error occurred.

Note

This routine is not usable in a ROM based system.

PUTS

Synopsis

```
#include <stdio.h>

int puts (const char * s)
```

Description

The **`puts()`** function writes the string **`s`** to the *stdout stream*, appending a *newline*. The null character terminating the string is not copied.

Example

```
#include <stdio.h>

void
main (void)
{
    puts("Hello, world!");
}
```

See Also

`fputs(), gets(), freopen(), fclose()`

Return Value

`EOF` is returned on error; zero otherwise.

PUTW

Synopsis

```
#include <stdio.h>

int putw (int w, FILE * stream)
```

Description

The **`putw()`** function copies the word **`w`** to the given **`stream`**. It returns **`w`**, except on error, in which case `EOF` is returned. Since this is a good integer, `ferror()` should be used to check for errors. The routine `getw()` may be used to read in integer written by `putw()`.

Example

```
#include <stdio.h>

void
main (void)
{
    FILE * fp;
    int c;

    fp = fopen("file.dat", "r");
    if(!feof(fp)) {
        while((c = getw(fp)) != EOF)
        putw(c, stdout);
        fclose(fp);
    }
}
```

See Also

`getw(), fopen(), fclose()`

Return Value

The return value is **`w`**, except on error, then the return value is `EOF`.

Note

This routine is not usable in a ROM based system.

QSORT

Synopsis

```
#include <stdlib.h>

void qsort (void * base, size_t nel, size_t width,
         int (*func)(const void *, const void *))
```

Description

The **qsort()** function is an implementation of the quicksort algorithm. It sorts an array of **nel** items, each of length **width** bytes, located contiguously in memory at **base**. The argument **func** is a pointer to a function used by **qsort()** to compare items. It calls **func** with pointers to two items to be compared. If the first item is considered to be greater than, equal to or less than the second then **func** should return a value greater than zero, equal to zero or less than zero respectively.

Example

```
#include <stdio.h>
#include <stdlib.h>

int aray[] = {
    567, 23, 456, 1024, 17, 567, 66
};

int
sortem (const void * p1, const void * p2)
{
    return *(int *)p1 - *(int *)p2;
}

void
main (void)
{
    register int i;

    qsort(aray, sizeof aray/sizeof aray[0], sizeof aray[0], sortem);
    for(i = 0 ; i != sizeof aray/sizeof aray[0] ; i++)
        printf("%d\t", aray[i]);
    putchar('\n');
}
```

Note

The function parameter must be a pointer to a function of type similar to:

```
int func (const void *, const void *)
```

*i.e. it must accept two const void * parameters, and must be prototyped.*

RAM_VECTOR, CHANGE_VECTOR, READ_RAM_VECTOR

Synopsis

```
#include <intrpt.h>

void RAM_VECTOR (unsigned vector, isr func)
void CHANGE_VECTOR (unsigned vector, isr func)
void (* READ_RAM_VECTOR (unsigned vector)(void))
```

Description

The **`RAM_VECTOR()`**, **`CHANGE_VECTOR()`** and **`READ_RAM_VECTOR()`** macros are used to initialize, modify and read interrupt vectors which are directed through internal RAM based interrupt vectors. These macros should only be used for vectors which need to be modifiable, so as to point at different interrupt functions at different points in the program. The **`CHANGE_VECTOR()`** and **`READ_RAM_VECTOR()`** macros should only be used with interrupt vectors which have been initialized using **`RAM_VECTOR()`**, otherwise garbage will be returned.

Please refer to the section "*Interrupt Handling in C*" in this manual for further details.

Example

```
volatile unsigned char wait_flag;

interrupt void wait_handler(void)
{
    ++wait_flag;
}

void wait_for_serial_intr(void)
{
    interrupt void (*old_handler)(void);

    di();
    old_handler = READ_RAM_VECTOR(RXI);
    wait_flag = 0;
    CHANGE_VECTOR(RXI, wait_handler);
```

See Also

`di(), ei(), ROM_VECTOR()`

Note

These macros, for the Z80/Z180, may only be used with mode 2 interrupts.

RAND

Synopsis

```
#include <stdlib.h>

int rand (void)
```

Description

The **rand()** function is a pseudo-random number generator. It returns an integer in the range 0 to 32767, which changes in a pseudo-random fashion on each call. The algorithm will produce a deterministic sequence if started from the same point. The starting point is set using the **srand()** call. The example shows use of the **time()** function to generate a different starting point for the sequence each time.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t toc;
    int i;

    time(&toc);
    srand((int)toc);
    for(i = 0 ; i != 10 ; i++)
        printf("%d\t", rand());
    putchar('\n');
}
```

See Also

srand()

Note

The example will require the user to provide the `time()` routine as one cannot be supplied with the compiler. See `time()` for more detail.

READ

Synopsis

```
#include <unixio.h>

size_t read (int fd, void * buf, size_t cnt)
```

Description

The **`read()`** function will read from the file associated with **`fd`** up to **`cnt`** bytes into a buffer located at **`buf`**. It returns the number of bytes actually read. A zero return indicates end-of-file. A negative return indicates error. The argument **`fd`** should have been obtained from a previous call to `open()`. It is possible for **`read()`** to return less bytes than requested, e.g. when reading from the console, in which case **`read()`** will read one line of input.

Example

```
#include <stdio.h>
#include <unixio.h>

void
main (void)
{
    char buf[80];
    int i;

        /* read from stdin */

    i = read(0, buf, sizeof(buf));
    printf("Read %d bytes\n", i);
}
```

See Also

`open(), close(), write()`

Return Value

The number of bytes read; zero on end-of-file, negative one (-1) on error. Be careful not to misinterpret a read of > 32767 as a negative return value.

Note

This routine is not usable in a ROM based system.

REALLOC

Synopsis

```
#include <stdlib.h>

void * realloc (void * ptr, size_t cnt)
```

Description

The **`realloc()`** function frees the block of memory at **`ptr`**, which should have been obtained by a previous call to `malloc()`, `calloc()` or **`realloc()`**, then attempts to allocate **`cnt`** bytes of dynamic memory, and if successful copies the contents of the block of memory located at **`ptr`** into the new block.

At most, **`realloc()`** will copy the number of bytes which were in the old block, but if the new block is smaller, will only copy **`cnt`** bytes.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * cp;

    cp = malloc(255);
    if(gets(cp))
        cp = realloc(cp, strlen(cp)+1);
    printf("buffer now %d bytes long\n", strlen(cp)+1);
}
```

See Also

`malloc(), calloc()`

Return Value

A pointer to the new (or resized) block. `NULL` if the block could not be expanded. A request to shrink a block will never fail.

REMOVE

Synopsis

```
#include <stdio.h>

int remove (const char * s)
```

Description

The **`remove()`** function will attempt to remove the file named by the argument **`s`** from the directory.

Example

```
#include <stdio.h>

void
main (void)
{
    if(remove("test.fil") < 0)
        perror("test.fil");
}
```

See Also

`unlink()`

Return Value

Zero on success, negative one (-1) on error.

Note

This routine is not usable in a ROM based system.

RENAME

Synopsis

```
#include <stdio.h>

int rename (const char * name1, const char * name2)
```

Description

The file named by **`name1`** will be renamed to **`name2`**.

Example

```
#include <stdio.h>

void
main (void)
{
    if(rename("test.fil", "test1.fil"))
        perror("Rename");
}
```

See Also

`open(), close(), unlink()`

Return Value

Negative one (-1) will be returned if the rename was not successful, zero if the rename was performed.

Note

Rename is not permitted across drives or directories.
This routine is not usable in a ROM based system.

REWIND

Synopsis

```
#include <stdio.h>

int rewind (FILE * stream)
```

Description

This function will attempt to re-position the read/write pointer of the nominated **stream** to the beginning of the file. This call is equivalent to `fseek(stream, 0L, 0)`.

Example

```
#include <stdio.h>
#include <stdlib.h>

void
main (void)
{
    char buf[80];

    if(!freopen("test.fil", "r", stdin))
        exit(1);
    gets(buf);
    printf("got ’%s’\n", buf);
    rewind(stdin);
    gets(buf);
    printf("Got ’%s’ again\n", buf);
}
```

See Also

`fseek(), ftell()`

Return Value

A return value of negative one (-1) indicates that the attempt was not successful, perhaps because the stream is associated with a non-random access file such as a character device.

Note

This routine is not usable in a ROM based system.

ROM_VECTOR

Synopsis

```
#include <intrpt.h>

void ROM_VECTOR (unsigned vector, isr func)
```

Description

The **`ROM_VECTOR()`** macro is used to set up a "*hard coded*" ROM vector, which points to an interrupt handler. This macro does not generate any code which is executed at run-time, so it can be placed anywhere in your code. **`ROM_VECTOR()`** generates in-line assembler code, so the vector address passed to it may be in any format acceptable to the assembler.

Please refer to the section "*Interrupt Handling in C*", in this manual for further details.

See Also

`di(), ei(), RAM_VECTOR()`

Note

These macros, for the Z80/Z180, may only be used with mode 2 interrupts.

SBRK

Synopsis

```
#include <sys.h>

void * sbrk (int incr)
```

Description

The **`sbrk()`** function increments the current highest memory location allocated to the program by **`incr`** bytes. It returns a pointer to the previous highest location. Thus **`sbrk(0)`** returns a pointer to the current highest location, without altering its value. This is a low-level routine not intended to be called by user code. Use `malloc()` instead.

See Also

`brk(), malloc(), calloc(), realloc(), free()`

Return Value

If there is insufficient memory to satisfy the request, (void *) -1 (negative one) is returned.

Note

This routine is not usable in a ROM based system.

SCANF, VSCANF

Synopsis

```
#include <stdio.h>

int scanf (const char * fmt, ...)

#include <stdio.h>
#include <stdarg.h>

int vscanf (const char *, va_list ap)
```

Description

The **`scanf()`** function performs formatted input ("de-editing") from the *stdin stream*. Similar functions are available for streams in general, and for strings. The function **`vscanf()`** is similar, but takes a pointer to an argument list rather than a series of additional arguments. This pointer should have been initialised with `va_start()`.

The input conversions are performed according to the **`fmt`** string; in general a character in the format string must match a character in the input; however a space character in the format string will match zero or more "white space" characters in the input, i.e. *spaces, tabs or newlines*.

A conversion specification takes the form of the character **`%`**, optionally followed by an assignment suppression character ('**`*`**'), optionally followed by a numerical maximum field width, followed by a conversion specification character. Each conversion specification, unless it incorporates the assignment suppression character, will assign a value to the variable pointed at by the next argument. Thus if there are two conversion specifications in the **`fmt`** string, there should be two additional pointer arguments.

The conversion characters are as follows:

`o x d`
Skip white space, then convert a number in base 8, 16 or 10 radix respectively. If a field width was supplied, take at most that many characters from the input. A leading minus sign will be recognized.

`f`
Skip white space, then convert a floating number in either conventional or scientific notation. The field width applies as above.

`s`
Skip white space, then copy a maximal length sequence of non-white-space characters. The pointer argument must be a pointer to char. The field width will limit the number of characters copied. The resultant string will be null terminated.

`c`
Copy the next character from the input. The pointer argument is assumed to be a pointer to char. If a field

width is specified, then copy that many characters. This differs from the **s** format in that white space does not terminate the character sequence.

The conversion characters **o**, **x**, **u**, **d** and **f** may be preceded by an **l** to indicate that the corresponding pointer argument is a pointer to long or double as appropriate. A preceding **h** will indicate that the pointer argument is a pointer to short rather than int.

Example

```
scanf("%d %s", &a, &c)
     with input " 12s"
     will assign 12 to a, and "s" to s.

scanf("%3cd %lf", &c, &f)
     with input " abcd -3.5"
     will assign " abc" to c, and -3.5 to f.
```

See Also

`fscanf(), sscanf(), printf(), va_arg()`

Return Value

The **scanf()** function returns the number of successful conversions; EOF is returned if end-of-file was seen before any conversions were performed.

SETJMP

Synopsis

```
#include <setjmp.h>

int setjmp (jmp_buf buf)
```

Description

The **`setjmp()`** function is used with `longjmp()` for non-local goto's. See `longjmp()` for further information.

Example

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf jb;

void
inner (void)
{
    longjmp(jb, 5);
}

void
main (void)
{
    int i;

    if(i = setjmp(jb)) {
        printf("setjmp returned %d\n", i);
        exit(0);
    }
    printf("setjmp returned 0 - good\n");
    printf("calling inner...\n");
    inner();
    printf("inner returned - bad!\n");
}
```

See Also

`longjmp()`

Return Value

The **`setjmp()`** function returns zero after the real call, and non-zero if it apparently returns after a call to `longjmp()`.

SETUID

Synopsis

```
#include <cpm.h>

int setuid (unsigned char uid)
```

Description

The **setuid()** function will set the current user number to **uid**. The **uid** argument should be a number in the range 0-15.

Example

```
#include<stdio.h>
#include<cpm.h>

void
main (void)
{
    setuid(2);
    printf("Current user number is: %d\n", getuid());
}
```

See Also

getuid()

Note

This routine is not usable in a ROM based system.

SETVBUF, SETBUF

Synopsis

```
#include <stdio.h>

int setvbuf (FILE * stream, char * buf, int mode, size_t size)
void setbuf (FILE * stream, char * buf)
```

Description

The **`setvbuf()`** function allows the buffering behaviour of a *stdio stream* to be altered. It supersedes the function **`setbuf()`** which is retained for backwards compatibility. The arguments to **`setvbuf()`** are as follows: **`stream`** designates the *stdio stream* to be affected; **`buf`** is a pointer to a buffer which will be used for all subsequent I/O operations on this **`stream`**. If **`buf`** is `NULL`, then the routine will allocate a buffer from the heap if necessary, of size `BUFSIZ` as defined in **`stdio.h`**. The argument **`mode`** may take the values `_IONBF`, to turn buffering off completely, `_IOFBF`, for full buffering, or `_IOLBF` for line buffering. Full buffering means that the associated buffer will only be flushed when full, while line buffering means that the buffer will be flushed at the end of each line or when input is requested from another *stdio stream*. The argument **`size`** is the size of the buffer supplied. By default, **`stdout`** and **`stdin`** are line buffered when associated with a terminal-like device, and full buffered when associated with a file.

If a buffer is supplied by the caller, that buffer will remain associated with that stream even over `fclose()`, `fopen()` calls until another **`setvbuf()`** changes it.

Example

```
#include <stdio.h>

char buffer[8192];

void
main (void)
{
    int i, j;

        /* set a large buffer for stdout */

    setvbuf(stdout, buffer, _IOFBF, sizeof buffer);
    for(i = 0 ; i != 2000 ; i++)
        if((i % 100) == 0)
            printf("i = %4d\n", i);
        else
```

```
            for(j = 0 ; j != 1000 ; j++)
                continue;
}
```

See Also

fopen(), freopen(), fclose()

Note

*If the **buf** argument is NULL, then the size is ignored.*
This routine is not usable in a ROM based system.

SET_VECTOR

Synopsis

```
#include <intrpt.h>

isr set_vector (isr * vector, isr func)
```

Description

This routine allows an interrupt vector to be initialized. The first argument should be the address of the interrupt vector (not the vector number but the actual address) cast to a pointer to **`isr`**, which is a typedef'd pointer to an interrupt function. The second argument should be the function which you want the interrupt vector to point to. This must be declared using the **`interrupt`** type qualifier.

Not all compilers support this routine; the macros `ROM_VECTOR()`, `RAM_VECTOR()` and `CHANGE_VECTOR()` are used with some processors. These routines are to be preferred even where **`set_vector()`** is supported. See **`intrpt.h`** or the processor specific manual section to determine what is supported for a particular compiler.

The example shown sets up a vector for the DOS ctrl-BREAK interrupt.

Example

```
#include <signal.h>
#include <stdlib.h>
#include <intrpt.h>

static far interrupt void
brkintr (void)
{
    exit(-1);
}

#define BRKINT  0x23
#define BRKINTV ((far isr *)(BRKINT * 4))

void
set_trap (void)
{
    set_vector(BRKINTV, brkintr);
}
```

See Also

`di(), ei(), ROM_VECTOR(), RAM_VECTOR(), CHANGE_VECTOR()`

Return Value

The return value of **`set_vector()`** is the previous contents of the vector, if **`set_vector()`** is implemented as a function. If it is implemented as a macro, it has no return value.

Note

*The **`set_vector()`** routine is equivalent to `ROM_VECTOR()` and is present only for compatibility with version 5 and 6 HI-TECH compilers. It is suggested that `ROM_VECTOR()` be used in place of **`set_vector()`** for maximum compatibility with future versions of HI-TECH C.*

Data Types

```
typedef interrupt void (*isr)(void)
```

SIGNAL

Synopsis

```
#include <signal.h>

void (* signal (int sig, void (*func)(int)))(int)
```

Description

The **signal()** function provides a mechanism for catching ctrl-C's (ctrl-BREAK for MS-DOS) typed on the console during I/O. Under CP/M the console is polled whenever an I/O call is performed. Under MS-DOS the polling depends on the setting of the BREAK command, if a ctrl-C is detected a certain action will be performed. The default action is to exit summarily; this may be modified with **signal()**. The **sig** argument to signal may at the present time be only **SIGINT**, signifying an interrupt condition. The **func** argument may be one of **SIG_DFL**, representing the default action i.e. to exit immediately, **SIG_IGN**, to ignore ctrl-C's completely, or the address of a function which will be called with one argument, the number of the signal caught, when a ctrl-C is seen. As the only signal supported is **SIGINT**, this will always be the value of the argument to the called function.

Example

```
#include <signal.h>
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf jb;

void
catch (int c)
{
    longjmp(jb, 1);
}

void
main (void)
{
    int i;

    if(setjmp(jb)) {
        printf("\n\nCaught signal\n");
        exit(0);
    }
    signal(SIGINT, catch);
    for(i = 0 ;; i++ ) {
        printf("%6d\r", i);
    }
}
```

See Also

exit()

Note

This routine is not usable in a ROM based system.

SIN

Synopsis

```
#include <math.h>

double sin (double f)
```

Description

This function returns the sine function of its argument.

Example

```
#include <math.h>
#include <stdio.h>

#define C 3.141592/180.0

void
main (void)
{
    double i;

    for(i = 0 ; i <= 180.0 ; i += 10)
        printf("sin(%3.0f) = %f, cos = %f\n", i, sin(i*C), cos(i*C));
}
```

See Also

`cos(), tan(), asin(), acos(), atan(), atan2()`

Return Value

Sine vale of **`f`**.

SPRINTF, VSPRINTF

Synopsis

```
#include <stdio.h>

int sprintf (char * buf, const char * fmt, ...)

#include <stdio.h>
#include <stdarg.h>

int vsprintf (char * buf, const char * fmt, va_list ap)
```

Description

The **`sprintf()`** function operates in a similar fashion to `printf()`, except that instead of placing the converted output on the *stdout stream*, the characters are placed in the buffer at **`buf`**. The resultant string will be null terminated, and the number of characters in the buffer will be returned.

The **`vsprintf()`** function is similar to **`sprintf()`** but takes a variable argument list pointer rather than a list of arguments. See the description of `va_start()` for more information on variable argument lists.

See Also

`printf(), fprintf(), sscanf()`

Return Value

Both these routines return the number of characters placed into the buffer.

SQRT

Synopsis

```
#include <math.h>

double sqrt (double f)
```

Description

The function **`sqrt()`**, implements a square root routine using Newton’s approximation.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double i;

    for(i = 0 ; i <= 20.0 ; i += 1.0)
        printf("square root of %.1f = %f\n", i, sqrt(i));
}
```

See Also

`exp()`

Return Value

Returns the value of the square root.

Note

A domain error occurs if the argument is negative.

SRAND

Synopsis

```
#include <stdlib.h>

void srand (unsigned int seed)
```

Description

The **srand()** function initializes the random number generator accessed by rand() with the given **seed**. This provides a mechanism for varying the starting point of the pseudo-random sequence yielded by rand(). On the z80, a good place to get a truly random seed is from the refresh register. Otherwise timing a response from the console will do, or just using the system time.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t toc;
    int i;

    time(&toc);
    srand((int)toc);
    for(i = 0 ; i != 10 ; i++)
        printf("%d\t", rand());
    putchar('\n');
}
```

See Also

rand()

10

SSCANF, VSSCANF

Synopsis

```
#include <stdio.h>

int sscanf (const char * buf, const char * fmt, ...)

#include <stdio.h>
#include <stdarg.h>

int vsscanf (const char * buf, const char * fmt, va_list ap)
```

Description

The **`sscanf()`** function operates in a similar manner to `scanf()`, except that instead of the conversions being taken from `stdin`, they are taken from the string at **`buf`**.

The **`vsscanf()`** function takes an argument pointer rather than a list of arguments. See the description of `va_start()` for more information on variable argument lists.

See Also

`scanf(), fscanf(), sprintf()`

Return Value

Returns the value of `EOF` if an input failure occurs, else returns the number of input items.

STAT

Synopsis

```
#include <stat.h>

int stat (char * name, struct stat * statbuf)
```

Description

This routine returns information about the file by **name**. The information returned is operating system dependent, but may include file attributes (e.g. read only), file size in bytes, and file modification and/or access times. The argument **name** should be the name of the file, and may include path names under DOS, user numbers under CP/M, etc. The argument **statbuf** should be the address of a structure as defined in **stat.h** which will be filled in with the information about the file. The structure of **struct stat** is as follows:

```
{
 short   st_mode; /* flags */
 long    st_atime;/* access time */
 long    st_mtime;/* modification time */
 long    st_size; /* file size */
};
```

The access and modification times (under DOS these are both set to the modification time) are in seconds since 00:00:00 Jan 1 1970. The function `ctime()` may be used to convert this to a readable value. The file size is self explanatory. The flag bits are as follows:

Flag	Meaning
S_IFMT	mask for file type
S_IFDIR	file is a directory
S_IFREG	file is a regular file
S_IREAD	file is readable
S_IWRITE	file is writeable
S_IEXEC	file is executable
S_HIDDEN	file is hidden
S_SYSTEM	file is marked system
S_ARCHIVE	file has been written to

Example

```
#include <stdio.h>
#include <stat.h>
#include <time.h>
#include <stdlib.h>

int
main (int argc, char ** argv)
{
    struct stat sb;

    if(argc > 1) {
        if(stat(argv[1], &sb)) {
            perror(argv[1]);
            exit(1);
        }
        printf("%s: %ld bytes, modified %s", argv[1],
                       sb.st_size, ctime(&sb.st_mtime));
    }
    exit(0);
}
```

See Also

`ctime(), creat(), chmod()`

Return Value

The **`stat()`** function returns zero on success, negative one (-1) on failure, e.g. if the file could not be found.

Note

This routine is not usable in a ROM based system.

Data Types

```
struct stat
{
     unsigned long  t_ino;        /* unused */
     unsigned short st_dev;       /* unused */
     unsigned short st_mode;      /* flags */
     long           st_atime;     /* access time */
     long           st_mtime;     /* modification time */
     long           st_size;      /* file size in bytes */
};
```

STRCAT

Synopsis

```
#include <string.h>

char * strcat (char * s1, const char * s2)
```

Description

This function appends (catenates) string **s2** to the end of string **s1**. The result will be null terminated. The argument **s1** must point to a character array big enough to hold the resultant string.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = " ... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

See Also

strcpy(), strcmp(), strncat(), strlen()

Return Value

The value of **s1** is returned.

STRCHR, STRICHR

Synopsis

```
#include <string.h>

char * strchr (const char * s, int c)
char * strichr (const char * s, int c)
```

Description

The **`strchr()`** function searches the string **`s`** for an occurrence of the character **`c`**. If one is found, a pointer to that character is returned, otherwise `NULL` is returned.

The **`strichr()`** function is the case-insensitive version of this function.

Example

```
#include <strings.h>
#include <stdio.h>

void
main (void)
{
    static char temp[] = "Here it is...";
    char c = ’s’;

    if(strchr(temp, c))
        printf("Character %c was found in string\n", c);
    else
        printf("No character was found in string");
}
```

See Also

`strrchr(), strlen(), strcmp()`

Return Value

A pointer to the first match found, or `NULL` if the character does not exist in the string.

Note

Although the function takes an integer argument for the character, only the lower 8 bits of the value are used.

STRCMP, STRICMP

Synopsis

```
#include <string.h>

int strcmp (const char * s1, const char * s2)
int stricmp (const char * s1, const char * s2)
```

Description

The **`strcmp()`** function compares its two, null terminated, string arguments and returns a signed integer to indicate whether **`s1`** is less than, equal to or greater than **`s2`**. The comparison is done with the standard collating sequence, which is that of the ASCII character set.

The **`stricmp()`** function is the case-insensitive version of this function.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    int i;

    if((i = strcmp("ABC", "ABc")) < 0)
        printf("ABC is less than ABc\n");
    else if(i > 0)
        printf("ABC is greater than ABc\n");
    else
        printf("ABC is equal to ABc\n");
}
```

See Also

`strlen(), strncmp(), strcpy(), strcat()`

Return Value

A signed integer less than, equal to or greater than zero.

Note

Other C implementations may use a different collating sequence; the return value is negative, zero or positive, i.e. do not test explicitly for negative one (-1) or one (1).

STRCPY

Synopsis

```
#include <string.h>

char * strcpy (char * s1, const char * s2)
```

Description

This function copies a null terminated string **s2** to a character array pointed to by **s1**. The destination array must be large enough to hold the entire string, including the null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = " ... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

See Also

`strncpy(), strlen(), strcat(), strlen()`

Return Value

The destination buffer pointer **s1** is returned.

STRCSPN

Synopsis

```
#include <string.h>

size_t strcspn (const char * s1, const char * s2)
```

Description

The **`strcspn()`** function returns the length of the initial segment of the string pointed to by **`s1`** which consists of characters NOT from the string pointed to by **`s2`**.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    static char set[] = "xyz";

    printf("%d\n", strcspn( "abcdevwxyz", set));
    printf("%d\n", strcspn( "xxxbcadefs", set));
    printf("%d\n", strcspn( "1234567890", set));
}
```

See Also

`strspn()`

Return Value

Returns the length of the segment.

10

STRDUP

Synopsis

```
#include <string.h>

char * strdup (const char * s1)
```

Description

The **`strdup()`** function returns a pointer to a new string which is a duplicate of the string pointed to by **`s1`**. The space for the new string is obtained using `malloc()`. If the new string cannot be created, a null pointer is returned.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * ptr;

    ptr = strdup("This is a copy");
    printf("%s\n", ptr);
}
```

Return Value

Pointer to the new string, or `NULL` if the new string cannot be created.

STRLEN

Synopsis

```
#include <string.h>

size_t strlen (const char * s)
```

Description

The **strlen()** function returns the number of characters in the string **s**, not including the null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = " ... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

Return Value

The number of characters preceding the null terminator.

10

STRNCAT

Synopsis

```
#include <string.h>

char * strncat (char * s1, const char * s2, size_t n)
```

Description

This function appends (catenates) string **s2** to the end of string **s1**. At most **n** characters will be copied, and the result will be null terminated. **s1** must point to a character array big enough to hold the resultant string.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = " ... end of line";
    strncat(s1, s2, 5);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

See Also

`strcpy(), strcmp(), strcat(), strlen()`

Return Value

The value of **s1** is returned.

STRNCMP, STRNICMP

Synopsis

```
  #include <string.h>

  int strncmp (const char * s1, const char * s2, size_t n)
  int strnicmp (const char * s1, const char * s2, size_t n)
```

Description

The **`strcmp()`** function compares its two, null terminated, string arguments, up to a maximum of **`n`** characters, and returns a signed integer to indicate whether **`s1`** is less than, equal to or greater than **`s2`**. The comparison is done with the standard collating sequence, which is that of the ASCII character set.

The **`stricmp()`** function is the case-insensitive version of this function.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    int i;

    i = strcmp("abcxyz", "abcxyz");
    if(i == 0)
        printf("Both strings are equal\n");
    else if(i > 0)
        printf("String 2 less than string 1\n");
    else
        printf("String 2 is greater than string 1\n");
}
```

See Also

`strlen(), strcmp(), strcpy(), strcat()`

Return Value

A signed integer less than, equal to or greater than zero.

Note

Other C implementations may use a different collating sequence; the return value is negative, zero or positive, i.e. do not test explicitly for negative one (-1) or one (1).

STRNCPY

Synopsis

```
#include <string.h>

char * strncpy (char * s1, const char * s2, size_t n)
```

Description

This function copies a null terminated string **s2** to a character array pointed to by **s1**. At most **n** characters are copied. If string **s2** is longer than **n** then the destination string will not be null terminated. The destination array must be large enough to hold the entire string, including the null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strncpy(buffer, "Start of line", 6);
    s1 = buffer;
    s2 = " ... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

See Also

`strcpy(), strcat(), strlen(), strcmp()`

Return Value

The destination buffer pointer **s1** is returned.

STRPBRK

Synopsis

```
#include <string.h>

char * strpbrk (const char * s1, const char * s2)
```

Description

The **strpbrk()** function returns a pointer to the first occurrence in string **s1** of any character from string **s2**, or a null pointer if no character from **s2** exists in **s1**.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * str = "This is a string.";

    while(str != NULL) {
        printf( "%s\n", str );
        str = strpbrk( str+1, "aeiou" );
    }
}
```

Return Value

Pointer to the first matching character, or `NULL` if no character found.

STRRCHR, STRRICHR

Synopsis

```
#include <string.h>

char * strrchr (const char * s, int c)
char * strrichr (const char * s, int c)
```

Description

The **`strrchr()`** function is similar to the **`strchr()`** function, but searches from the end of the string rather than the beginning, i.e. it locates the *last* occurrence of the character **`c`** in the null terminated string **`s`**. If successful it returns a pointer to that occurrence, otherwise it returns `NULL`.

The **`strrichr()`** function is the case-insensitive version of this function.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * str = "This is a string.";

    while(str != NULL) {
      printf( "%s\n", str );
      str = strrchr( str+1, ’s’);
    }
}
```

See Also

`strchr(), strlen(), strcmp(), strcpy(), strcat()`

Return Value

A pointer to the character, or `NULL` if none is found.

STRSPN

Synopsis

```
#include <string.h>

size_t strspn (const char * s1, const char * s2)
```

Description

The **strspn()** function returns the length of the initial segment of the string pointed to by **s1** which consists entirely of characters from the string pointed to by **s2**.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    printf("%d\n", strspn("This is a string", "This"));
    printf("%d\n", strspn("This is a string", "this"));
}
```

See Also

strcspn()

Return Value

The length of the segment.

STRSTR, STRISTR

Synopsis

```
#include <string.h>

char * strstr (const char * s1, const char * s2)
char * stristr (const char * s1, const char * s2)
```

Description

The **`strstr()`** function locates the first occurrence of the sequence of characters in the string pointed to by **`s2`** in the string pointed to by **`s1`**.

The **`stristr()`** routine is the case-insensitive version of this function.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    printf("%d\n", strstr("This is a string", "str"));
}
```

Return Value

Pointer to the located string or a null pointer if the string was not found.

STRTOK

Synopsis

```
#include <string.h>

char * strtok (char * s1, const char * s2)
```

Description

A number of calls to **`strtok()`** breaks the string **`s1`** (which consists of a sequence of zero or more text tokens separated by one or more characters from the separator string **`s2`**) into its separate tokens.

The first call must have the string **`s1`**. This call returns a pointer to the first character of the first token, or `NULL` if no tokens were found. The inter-token separator character is overwritten by a null character, which terminates the current token.

For subsequent calls to **`strtok()`**, **`s1`** should be set to a null pointer. These calls start searching from the end of the last token found, and again return a pointer to the first character of the next token, or `NULL` if no further tokens were found.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * ptr;
    char * buf = "This is a string of words.";
    char * sep_tok = ".,?! ";

    ptr = strtok(buf, sep_tok);
    while(ptr != NULL) {
        printf("%s\n", ptr);
        ptr = strtok(NULL, sep_tok);
    }
}
```

Return Value

Returns a pointer to the first character of a token, or a null pointer if no token was found.

Note

*The separator string **`s2`** may be different from call to call.*

TAN

Synopsis

```
#include <math.h>

double tan (double f)
```

Description

The **`tan()`** function calculates the tangent of **`f`**.

Example

```
#include <math.h>
#include <stdio.h>

#define C 3.141592/180.0

void
main (void)
{
    double i;

    for(i = 0 ; i <= 180.0 ; i += 10)
        printf("tan(%3.0f) = %f\n", i, tan(i*C));
}
```

See Also

`sin(), cos(), asin(), acos(), atan(), atan2()`

Return Value

The tangent of **`f`**.

TIME

Synopsis

```
#include <time.h>

time_t time (time_t * t)
```

Description

This function is not provided as it is dependant on the target system supplying the current time. This function will be user implemented. When implemented, this function should return the current time in seconds since 00:00:00 on Jan 1, 1970. If the argument **t** is not equal to NULL, the same value is stored into the object pointed to by **t**.

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;

    time(&clock);
    printf("%s", ctime(&clock));
}
```

See Also

ctime(), gmtime(), localtime(), asctime()

Return Value

This routine when implemented will return the current time in seconds since 00:00:00 on Jan 1, 1970.

Note

*The **time()** routine is not supplied, if required the user will have to implement this routine to the specifications outlined above.*

TMPFILE

Synopsis

```
#include <stdio.h>

FILE * tmpfile (void)
```

Description

This function creates a temporary binary file which is automatically closed and deleted on program termination.

See Also

`fopen()`

Return Value

Pointer to the file stream created. A null pointer is returned if the temporary file could not be opened.

Note

The file is opened using the "`wb+`" mode.
This routine is not usable in a ROM based system.

TMPNAM

Synopsis

```
#inlcude <stdio.h>

char * tmpnam (char * s)
```

Description

The **tmpnam()** function generates a valid file name which is different from the names of existing files. The **tmpnam()** function may be called up to TMP_MAX times, with a different string generated each time. The behaviour of **tmpnam()** is undefined for any further calls to it.

Example

```
#include <stdio.h>
#include <stdlib.h>

void
main (void)
{
    char * filename;
    FILE * fp;

    filename = tmpnam(NULL);
    if(filename != NULL)  {
        fp = fopen(filename, "wt");
            /* Can be used to create an error report */
        fclose(fp);
        free(filename);
    }
    else
        printf("Cannot create temp file name\n");
}
```

Return Value

If the argument is a null pointer, **tmpnam()** leaves the result in an internal static object and returns a pointer to that object. *Note* that subsequent calls to **tmpnam()** may modify that object. If the argument is not a null pointer, it is assumed to point to a character of at least **L_tmpnam** characters. In this case, the result is written in to the array, and the argument is returned.

Note

This routine is not usable in a ROM based system.

TOLOWER, TOUPPER, TOASCII

Synopsis

```
#include <ctype.h>

char toupper (int c)
char tolower (int c)
char toascii (int c)
```

Description

The **`toupper()`** function converts its lower case alphabetic argument to upper case, the **`tolower()`** routine performs the reverse conversion and the **`toascii()`** macro returns a result that is guaranteed in the range 0-0177. The functions **`toupper()`** and **`tolower()`** return their arguments if it is not an alphabetic character.

Example

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

void
main (void)
{
    char * array1 = "aBcDE";
    int i;

    for(i=0;i < strlen(array1); ++i)  {
        printf("%c", tolower(array1[i]));
    }
    printf("\n");
}
```

See Also

`islower(), isupper(), isascii(), et. al.`

UNGETC

Synopsis

```
#include <stdio.h>

int ungetc (int c, FILE * stream)
```

Description

The **ungetc()** function will attempt to push back the character **c** onto the named **stream**, such that a subsequent call to the getc() operation will return the character. If the **stream** is not buffered, at most one level of push back will be allowed, even this may not be possible. EOF is returned if the **ungetc()** function could not be performed.

Example

```
#include <stdio.h>
#include <ctype.h>

void
main (void)
{
    FILE * stream;
    int c;
    long number = 0;

    if(stream = fopen("temp.dat", "r"))  {
        c = fgetc(stream);
        while(isdigit(c)) {
            number = number*10 + (c - ’0’);
            c = fgetc(stream);
        }
        ungetc(c, stream);
        printf("Read number is = %ld\n", number);
        fclose(stream);
    }
    else
        printf("Could not open file.\n");
}
```

See Also

getc()

Return Value

Returns the character pushed back, or `EOF` if the **`ungetc()`** could not be performed.

Note

This routine is not usable in a ROM based system.

UNGETCH

Synopsis

```
#include <conio.h>

void ungetch (char c)
```

Description

The **`ungetch()`** function will push back the character **`c`** onto the console stream, such that a subsequent `getch()` operation will return the character. At most one level of push back will be allowed.

See Also

`getch(), getche()`

UNLINK

Synopsis

```
#include <unixio.h>

int unlink (const char * name)
```

Description

The **`unlink()`** function will remove (delete) the named file, that is, erase the file from its directory. See `open()` for a description of the file name construction. Zero will be returned if successful, negative one if the file did not exist or it could not be removed. The ANSI function `remove()` is preferred to **`unlink()`**.

Example

```
#include <unixio.h>

void
main (void)
{
    if(unlink("test.fil") < 0)
        perror("test.fil");
}
```

See Also

`open(), close(), rename(), remove()`

Return Value

Zero will be returned if successful, negative one (-1) if the file did not exist or it could not be removed.

Note

This routine is not usable in a ROM based system.

VA_START, VA_ARG, VA_END

Synopsis

```
#include <stdarg.h>

void va_start (va_list ap, parmN)
type va_arg  (ap, type)
void va_end  (va_list ap)
```

Description

These macros are provided to give access in a portable way to parameters to a function represented in a prototype by the ellipsis symbol (...), where type and number of arguments supplied to the function are not known at compile time.

The rightmost parameter to the function (shown as **parmN**) plays an important role in these macros, as it is the starting point for access to further parameters. In a function taking variable numbers of arguments, a variable of type **va_list** should be declared, then the macro **va_start()** invoked with that variable and the name of **parmN**. This will initialize the variable to allow subsequent calls of the macro **va_arg()** to access successive parameters.

Each call to **va_arg()** requires two arguments; the variable previously defined and a type name which is the type that the next parameter is expected to be. Note that any arguments thus accessed will have been widened by the default conventions to *int*, *unsigned int* or *double*. For example if a character argument has been passed, it should be accessed by **va_arg(ap, int)** since the *char* will have been widened to *int*.

An example is given below of a function taking one integer parameter, followed by a number of other parameters. In this example the function expects the subsequent parameters to be pointers to char, but note that the compiler is not aware of this, and it is the programmers responsibility to ensure that correct arguments are supplied.

Example

```
#include <stdio.h>
#include <stdarg.h>

void
pf (int a, ...)
{
    va_list ap;

    va_start(ap, a);
    while(a--)
        puts(va_arg(ap, char *));
    va_end(ap);
}

void
main (void)
{
    pf(3, "Line 1", "line 2", "line 3");
}
```

WRITE

Synopsis

```
#include <unixio.h>

size_t write (int fd, const void * buf, size_t cnt)
```

Description

The **`write()`** function will write from the buffer at **`buf`** up to **`cnt`** bytes to the file associated with the file descriptor **`fd`**. The number of bytes actually written will be returned. `EOF` or a value less than **`cnt`** will be returned on error. In any case, any return value not equal to **`cnt`** should be treated as an error (cf. `read()`).

Example

```
#include <unixio.h>

void
main (void)
{
    write(1, "A test string\r\n", 15);
}
```

See Also

`open(), close(), read()`

Return Value

The number of bytes actually written will be returned. `EOF` or a value less than **`cnt`** will be returned on error.

Note

This routine is not usable in a ROM based system.

XTOI

Synopsis

```
#include <stdlib.h>

unsigned xtoi (const char * s)
```

Description

The **`xtoi()`** function scans the character string passed to it, skipping leading blanks reading an optional sign, and converts an ASCII representation of a hexadecimal number to an integer.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    int i;

    gets(buf);
    i = xtoi(buf);
    printf("Read %s: converted to %x\n", buf, i);
}
```

See Also

`atoi()`

Return Value

A signed integer. If no number is found in the string, zero will be returned.

_GETARGS

Synopsis

```
#include <sys.h>

 char ** _getargs (char * buf, char * name)
 extern int _argc_
```

Description

This routine performs I/O redirection (CP/M only) and wild card expansion. Under MS-DOS I/O redirection is performed by the operating system. It is called from startup code to operate on the command line if the -R option is used to the C command, but may also be called by user-written code. If the **buf** argument is null, it will read lines of text from standard input. If the standard input is a terminal (usually the console) the **name** argument will be written to the standard error stream as a prompt. If the **buf** argument is not null, it will be used as the source of the string to be processed. The returned value is a pointer to an array of strings, exactly as would be pointed to by the argv argument to the main() function. The number of strings in the array may be obtained from the global **_argc_.**

There will be one string in the array for each word in the buffer processed. Quotes, either single (') or double (") may be used to include white space in "words". If any wild card characters (? or *) appear in a non-quoted word, it will be expanded into a string of words, one for each file matching the word. The usual CP/M and DOS conventions are followed for this expansion. On CP/M any occurence of the redirection characters > and < outside quotes will be handled in the following manner:

@Hanging list = > **name** will cause standard output to be redirected to the file **name**.

@Hanging list = < **name** will cause standard input to be redirected from the file **name**.

@Hanging list = >> **name** will cause standard output to append to file **name**.

White space is optional between the > or < character and the file name, however it is an error for a redirection character not to be followed by a file name. It is also an error if a file cannot be opened for input or created for output. An append redirection (>>) will create the file if it does not exist. If the source of text to be processed is standard input, several lines may be supplied by ending each line (except the last) with a backslash (\). This serves as a continuation character. Note that the newline following the backslash is ignored, and not treated as white space.

Example

```
#include <sys.h>

void
main (int argc, char ** argv)
{
    extern char ** _getargs(char *, char *);
```

10

```
    extern int     _argc_;

    if(argc == 1) {   /* no arguments */
    argv = _getargs(0, "myname");
    argc = _argc_;
    }
    .
    .
    .
}
```

Return Value

A pointer to an array of strings.

Note

Under CP/M the first element of the array returned by will not be the name of the program but will be the ***name*** *argument. This routine is not usable in a ROM based system.*

Index

ZC Command Line Options

-180	Generate code for the Z180 processor
-64180	Generate code for the 64180 processor
-A*spec*	Specify memory addresses for linking
-AAHEX	Generate an American Automation symbolic HEX file
-ALTREG	Use alternate register set
-ASMLIST	Generate assembler .LST file for each compilation
-AV	Select AVOCET format symbol table
-AVSIM	Same as -AV
-BIN	Generate a Binary output file
-Bs	Select *small* memory model
-Bl	Select *large* memory model
-Bc	Select *CP/M* memory model
-C	Compile to object files only
-CLIST	Generate C source listing file
-CPM	Generate CP/M executable file
-CR*file*	Generate cross-reference listing
-D*macro*	Define pre-processor macro
-E	Use “editor” format for compiler errors
-E*file*	Redirect compiler errors to a file
-E+*file*	Append errors to a file
-G*file*	Generate source level symbol table
-H*file*	Generate symbol table without line numbers etc.
-HELP	Print summary of options
-I*path*	Specify a directory pathname for include files
-L*library*	Specify a library to be scanned by the linker
-L-*option*	Specify -*option* to be passed directly to the linker
-M*file*	Request generation of a MAP file
-MOTOROLA	Generate a Motorola S1/S9 HEX format output file
-N*length*	Set identifier length to *length* (default is 31 characters)
-O	Enable peephole optimization
-O*file*	Specify output filename
-OF	Optimise for speed
-OMF51	Produce an OMF-51 output file
-P	Preprocess assembler files
-P8	Use 8 bit port addressing
-P16	Use 16 bit port addressing
-PROTO	Generate function prototype information
-PSECTMAP	Display complete memory segment usage after linking
-q	Specify quiet mode
-ROMDATA	Leave initialised data in ROM
-ROM*ranges*	Specify ROM ranges for code
-S	Compile to assembler source files only
-SA	Compile to Avocet AVMAC assembler source files
-STRICT	Enable strict ANSI keyword conformance
-TEK	Generate a Tektronix HEX format output file
-UBROF	Generate an UBROF format output file
-UNSIGNED	Make default character type *unsigned*
-U*symbol*	Undefine a predefined pre-processor symbol
-V	Verbose: display compiler pass command lines
-W*level*	Set compiler warning level
-X	Eliminate local symbols from symbol table
-Z180	Generate code for the Z180 processor
-Zg	Enable global optimization in the code generator

HPDZ80 menu hot keys

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-J	DOS Shell
Alt-F	Open File menu
Alt-E	Open Edit menu
Alt-I	Open Compile menu
Alt-M	Open Make menu
Alt-R	Open Run menu
Alt-T	Open Options menu
Alt-U	Open Utility menu
Alt-H	Open Help menu
Alt-P	Open Project file
Alt-W	Warning level dialog
Alt-Z	Optimization menu
Alt-D	Command.com
F3	Compile and link single file
Shift-F3	Compile to object file
Ctrl-F3	Compile to assembler code
Ctrl-F4	Retrieve last file
F5	Make target program
Shift-F5	Re-link target program
Ctrl-F5	Re-make all objects and 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
Alt-X	Cut to clipboard
Alt-C	Copy to clipboard
Alt-V	Paste from clipboard