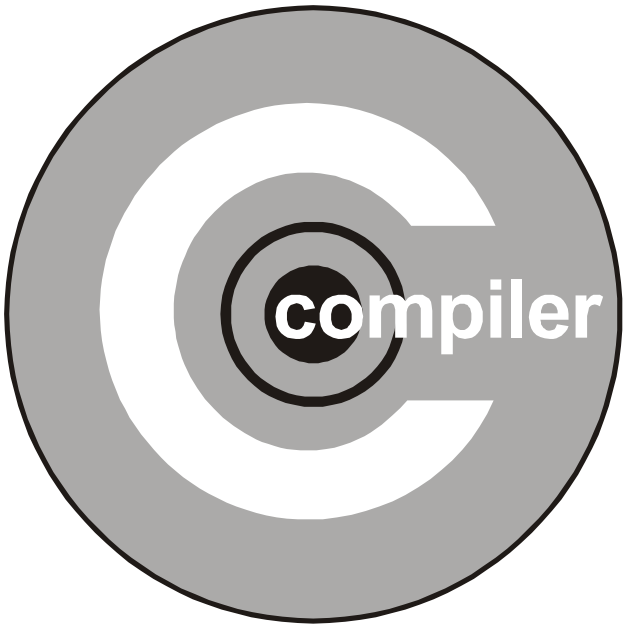
User's Guide

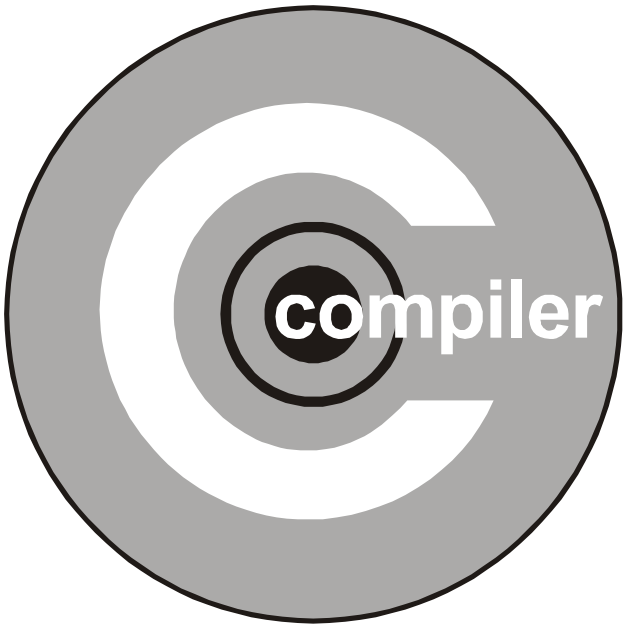


PICC

ANSI C COMPILER

HI-TECH SOFTWARE

User's Guide



PICC

ANSI C COMPILER

HI-TECH SOFTWARE

PICC Manual

Twenty-first Printing (c), June 2004

HI-TECH Software Pty Ltd ACN 002 724 549
ACN 002 724 549
PO Box 103
Alderley QLD 4051
Australia

Email: hitech@htsoft.com
Web: http://www.htsoft.com/
FTP: ftp.htsoft.com

Introduction

1.1 Typographic conventions

Different fonts and styles are used throughout this manual to indicate special words or text. Computer prompts, responses and filenames will be printed in `constant-spaced type`**.** When the filename is the name of a standard header file, the name will be enclosed in angle brackets, e.g. `<stdio.h>`. These header files can be found in the INCLUDE directory of your distribution.

Samples of code, C keywords or types, assembler instructions and labels will be printed in a **`bold constant-space type`**. When used at the beginning of a sentence, a capital letter will be used. C code is case sensitive so this capitalization may need to be removed if entering the code into a program.

Particularly useful points and new terms will be emphasised using *italicised type*. When part of a term requires substitution, that part should be printed in the appropriate font, but in *itialics*. For example: **`#include <`*`filename.h`*`>`**.

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 with references to menu items, dialogue tab labels and button names in **bold text**.

1.2 Using This Manual

This manual is a comprehensive guide and reference to using PICC. The chapters included are as follows:

- ❒ Tutorials to aid in the understanding and usage of HI-TECH's C cross compilers
- ❒ How to use the HI-TECH Professional Development (HPD) environment
- ❒ How to use the PICC command-line interface
- ❒ In-depth description of the C compiler
- ❒ How to use the assembler
- ❒ How to use the linker and other utilities
- ❒ Error messages and their meaning
- ❒ Description of provided library functions

For information on installing PICC, using the on-line manual and getting started, see the *Quick Start Guide*.

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 denote 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 20. 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 of your distribution. 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 stage and the files that are involved. Reference should be made to Figure 2 - 1 on page 21 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.

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

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

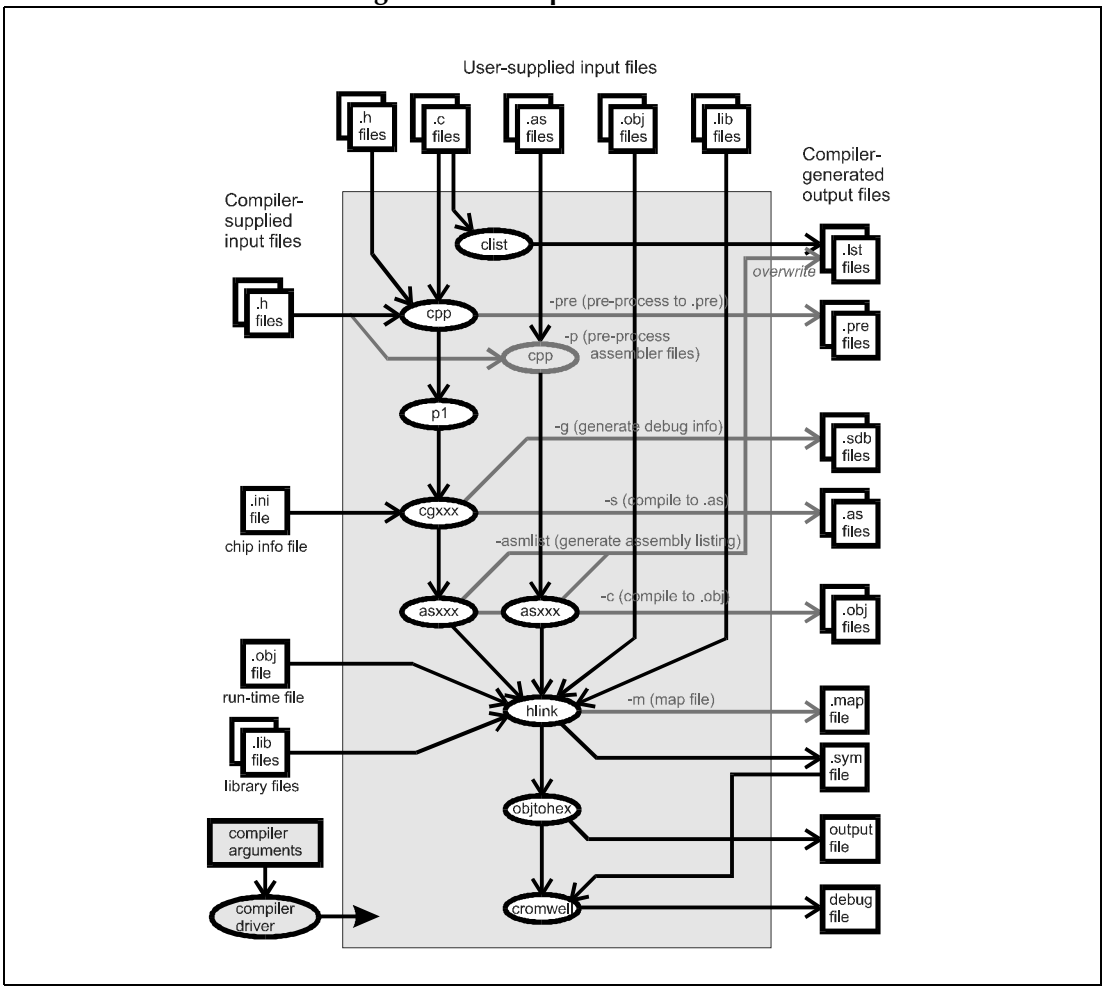
The user supplies several things to the compiler to make a program: the input files and the compiler options, whether using the CLD or HPD. The compiler accepts many different input file types. These are discussed below.

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 preprocessor directive and are often called *include files*. Since header files are referenced by a command that includes

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



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 unportable 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 23. 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. For example, if you have included `init.c` into the project, you should *not* include `init.obj` into the object file list.

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 the LIB 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 23.

Table 2 - 3 clist output

C source	C listing

`#define VAL 2`
 	
`int a, b = 1;`
 	
`void`
`main(void)`
`{`
`    /* set starting value */`
`    a = b + VAL;`
`}`
	`1: #define VAL 2`
	`2:`
	`3: int a, b = 1;`
	`4:`
	`5: void`
	`6: main(void)`
	`7: {`
	`8:     /* set starting value */`
	`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 preprocessor directives in the source. Directives may, for example, replace macros with their replacement text (e.g. **`#define`** directives) or conditionally include source code subject to certain conditions (e.g. **`#if`**, **`#ifdef`** etc. directives). The preprocessor also inserts header files, whether user- or compiler-supplied, into the source. Table 2 - 4 on page 24 shows preprocessor output for the test program.

The output of the preprocessor is C source, but it may contain code which has been included by the preprocessor from header files and conditional code may have been omitted. Thus the preprocessor output usually contains similar, but different code to the original source file. The preprocessor 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 operates 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 preprocessor output for the test program is included by the preprocessor to indicate the filename 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 preprocessor is controlled by the compiler driver (either the CLD or

Table 2 - 4 preprocessor 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 preprocessor 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 preprocessor can evaluate preprocessor directives which contain a **sizeof(*type*)** expression. The output of the preprocessor 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 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 C 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 25 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

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

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.

Table 2 - 6 Parser output

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

It is the parser that is responsible for finding a large portion of the errors in the source code. These errors 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 preprocessors 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 used to generate code for the statement highlighted in Table 2 - 7 on page 26 using 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 26 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 25 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 and 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 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.

Table 2 - 8 Assembler output

C source	Relocatable object file

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

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 28 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 29. 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..

Table 2 - 9 Assembler listing

C source	Assembler listing

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

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 a file was created. After linking, the assembler listing file will have unresolved addresses and address markers removed and replaced with their final 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 compiled 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 was 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 preprocessor allowing the using of all preprocessor directives within assembly code. The output of the preprocessor 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.

3. The generation of this file is not shown in Figure 2 - 1 on page 21 in the interests of clarity.

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 relocatable 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 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. For example, if the address of the symbol **_b** in our running example was determined to be 20000h, the linker would not be able to fit this address into the first underlined two byte "hole" in the object file shown dumped in the Table Assembler output on page 28 since 20000h is larger than two bytes long.

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.

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.

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

The OBJTOHEX application is responsible for producing the output file requested by the user. It takes the

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

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.

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.

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 variables to be placed in non-volatile areas, 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 **p**rogram **sect**ions. 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 second **`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 can be linked in three different ways: 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 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.

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

Objects that are initialised are usually placed into psects with the name "data" or a name based on "data". Variables that are qualified **near** typically use the psect `rdata`. The PIC data psects use names like `rdata_0` to indicates that they are "near" (there is no **near** qualifier on the PIC - essentially all PIC objects are **near** by default) and the digit indicates a bank number.

Uninitialised objects are placed in psects whose name is "bss" or a name based on "bss". Again, `rbss` would indicate uninitialised objects that are **near**. The PIC compiler uses names like `rbss_0`, where the digit is a bank number. The abreviation "bss" stands for **b**lock **s**tarted by **s**ymbol and was a assembler pseudo-op used in *IBM* systems back in the days when computers were coal-fired. The continued usage of this term is still appropriate as there are some similarities in the way these schemes worked.

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 XA compiler handles these two objects. Other compilers may produce differently structured code. The **PSECT** directive flags are discussed 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 are handled differently by the compilers. They may be allocated space dynamically (for example on the stack) in which case they are not stored in psects by the compiler. Compilers or memory models which do not use a hardware stack, use a *compiled stack* which is an area of memory set aside for the auto and parameter objects for each function. These object will be positioned in a psect. The psect in which they are allocated is defined by a **FNCONF** directive which is placed in the run-time startup code.

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 will be the same, as they 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. They are assigned a load address, and you will see it listed in the map file, but it is not used. The compiler usually 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.

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

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** 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 be 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.

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** itself 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 (i.e. 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 bytes long. These psects are to be placed in ROM and all have the simple definition generated by the code generator:

```
PSECT text,class=CODE
```

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.

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.
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.

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.

```
GLOBAL   start
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.

```
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.) Other compilers may also operate this way if ROM and RAM are in different memory spaces. 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 for the PIC compiler will be similar to rdata_0; those in ROM are like idata_0. The digit refers to a PIC 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 class **`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 44.

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` psects, 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 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 XA. 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 code. 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 `1.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.

2.4 Addresses used with the PIC

The PIC processor has a complicated memory map with *banked RAM* and *paged ROM*, and each memory having different word widths. One of the biggest sources of confusion regarding addresses used by the compiler stems from the fact that internal PIC RAM is one byte wide, but the ROM is either 12, 14 or 16 bits wide on baseline, midrange or highend processors, respectively. This tutorial explains the different addresses that are used by the HI-TECH compiler.

2.4.1 Code addresses

Labels used in PIC code (stored in ROM), such as C function names, are assigned *word addresses*. If you check the map file after a compilation, the addresses of C functions, assembler routines and labels shown in the symbol table are word addresses. The `text`*n* psects (where *n* is a number) are the psects which will contain the assembler generated from a C function. These psects use a **`delta`** psect flag of 2. This indicates that one addressable unit - in this case an instruction - will take up 2 bytes of memory. Even if the PIC has 12-bit wide ROM, the delta value is still 2. The delta value must be an integral number.

2.4.2 Data addresses

The addresses of data objects are a little more complicated. Consider an array of characters (bytes) that is placed in RAM. A label - the name of the array - will be used to reference the first element of the array. The array will be placed in one of the `rdata` or `rbss` psects depending on whether the array was initialised. Both these psects have a *delta* value of 1 which indicates that the addressable unit - in this case a character - is 1 byte long. The address of the array in RAM will be a *byte address*.

If the array was declared as **`const`**, then it will be stored in ROM. On baseline and midrange PICs, ROM cannot be read like RAM, so the values are stored as **`retlw`** (return with a literal (constant) in the W register) assembler instructions, each taking up one word of ROM. The instructions' data values

represent each element of the array. Since the data is actually stored as code, the *delta* value of the psect containing the data will be 2 even though the object that these instructions represent is only a byte. The *delta* value must be 2 since it refers to the contents of the psect which is code, not byte-wide data values.

The highend PIC devices can read ROM memory directly and store the array elements as their byte values in the ROM. Highend PICs also have 16-bit wide ROM so the compiler can, and does, store 2 bytes for every word location in the ROM. Although these values are stored in a 16-bit wide area of memory, the *delta* value for the psect that contains the data is 1. This implies that the addressable unit in this area is one byte. This is necessary since you must be able to address each half of the word to access each of the array elements stored there.

The above has important implications for the addresses that result during linking of programs with constant data. With baseline and mid-range PICs, the address of a constant array label will be a word address; with highend PICs, it will be a byte address. If you want to search for the array in memory when using an emulator, for example, you must remember to divide the address by 2 to convert it to a word address if you are using a highend device. The map file for highend devices will also show byte addresses for the constant data, but the code labels will be word addresses. Sometimes it appears as if constant data has overwritten code since their addresses overlap, but again, you cannot compare byte and word addresses. Always check the *delta* value of a psect if you are unsure whether you are working with a byte or word address. The *delta* value is displayed with the psect's definition which can usually be seen in the assembler listing file for that module.

The *delta* value does not indicate the size of objects stored in the psect. For example, if the array had been a **const** array of **int**s, the *delta* value will still be 1, even though the size of an **int** is 2 bytes. The addressable unit of an **int** is 1, since it is possible (via pointers for example) to only access bytes within the **int** value. On the PIC there are only two *delta* values: 2 for code, 1 for data.

As an example, consider the following do-nothing program.

```
const char array[] = {0x30, 0x31, 0x32};

void
main(void)
{
}
```

This is compiled for the 17C756 highend PIC processor. The assembler listing file includes the following information.

```
4               PSECT    cstrings,global,class=CODE,delta=1
5               PSECT    text0,local,class=CODE,delta=2
6               PSECT    text1,local,class=CODE,delta=2
```

```
...
    26              PSECT cstrings
    27  3FFC                _array
    28  3FFC  0030              DB 30
    29  3FFD  0031              DB 31
    30  3FFE  0032              DB 32
    31
    32              PSECT text0
    33  1FFA                _main
    34              ;const.c: 6: }
    35  1FFA  B020  0103 B000  ljmp start
      +       0102
Symbol Table                            Thu Jun 17 12:10:06 1999

  _array 3FFC     _main 1FFA     start 2000
```

The array was placed in the `cstrings` psect which has a *delta* value of 1, but which was positioned in ROM since the object is **`const`**. The address of the array is listed as 3FFC. Since the *delta* value is 1, this must be a byte address. The routine **`main()`** (label **`_main`**) is listed as starting at address 1FFA. Since this routine is in the `text0` psect which has a *delta* value of 2, this is a word address. To determine where each psect is positioned relative to the other, convert the byte address to a word address (3FFC becomes 1FFE) and you will notice that the string is positioned immediately after the four-word main routine in ROM, which begins at 1FFA.

The HEX file generated was loaded into a simulator (MPLAB in this case) and the program memory examined. A section of this memory is shown below.

```
1FF9  FFFF          call   0x1FFF
1FFA  B020    main  movlw  0x20
1FFB  0103          movwf  0x3
1FFC  B000          movlw  0x0
1FFD  0102          movwf  0x2
1FFE  3130          cpfseq 0x30
1FFF  0032          ret
2000  2B04    exit  setf   0x4
```

The three bytes representing the array's contents have been set in bold type. The address of the function **`main()`** and the array are underlined in the address column. Notice that 2 characters have been stored

at word address 1FFE, and the unused byte at the following address has been filled with a 0 byte. The mnemonic interpretation shown (the **cpfseq** and **ret** instructions) for the array is meaningless.

Note that the array label is not displayed in the code. Some simulators/emulators may indicate the label at the byte address, i.e at an incorrect position. Unless the simulator/emulator knows about delta values and can adjust the addresses, some labels may not shown at the correct position. Note also, that often there can be several labels associated with an address. In the above example, the **__Lcstrings** label generated by the linker will also coincide with the name of the array since there is only one array in the cstrings psect. It can be difficult for the simulator/emulator to know which of these labels it should display.

If you check for objects in the HEX file remember that the addresses used in the records will be byte addresses. These will need to be divided by 2 if you are trying to find an object using a known word address.

Another thing that users must be aware of is that addresses used in linker options use the same units as the psects which they position. For example if you wished to position the cstrings psect at word address 1000h, you will need to use a linker option like the following.

```
-pcstrings=2000h
```

since the address is specified as a byte address (*delta* is 1 for the cstrings psect). But to position a psect which has a *delta* value of 2, you need to supply a word address.

The cstrings psect may be allocated an address using a class, for example the ROMDATA class. Each class is specified as using either byte or word addresses. ROMDATA uses word address. Thus, if ROMDATA is defined to cover the address range 8000h-BFFFh, and cstrings was positioned:

```
-pcstrings=ROMDATA
```

then the cstrings psect will be placed somewhere in the word address range 8000h to BFFFh and the map file will show that the cstrings psect was placed at a byte address somewhere between 10000h and 17FFEh. That is, the linker knows that the class uses word addresses and that the psect needs a byte address (since it has a *delta* value of 1) and has automatically scaled the addresses used.

The CODE class uses word addresses. To specify addresses for the text*n* psects which are in the CODE class (and have a *delta* value of 2), the linker does not need to scale any of the addresses. For example, to position the text*n* psects in the word address range 0 to 4FFh, the linker option is straightforward:

```
-ACODE=0-4FFh
```

and the map file will show the text*n* psect as residing somewhere in the word address range 0 to 4FFh.

The difference between the cstrings and text*n* psects is that the text*n* psects are in the CODE class, but that the cstrings psect is not in the ROMDATA class, even though it can use the ROMDATA class to specify

an relocation address. To determine whether a class uses word or byte addresses, look at either the assembler listing for your program or the run-time assembler file to see what the *delta* value is for psects that are in the class in question.

2.4.3 Bit addresses

The other type of address that is sometimes seen are *bit addresses*. The compiler will produce a range of addresses that **`bit`** objects take up after compilation if there are any objects defined as type **`bit`** in the source. The addresses uses in this case are bit addresses. **`Bit`** objects can only reside in RAM and are accessed via a special bit assembler instruction. The operands used with this instruction are a byte address and a bit offset (the bit number within that byte). To convert a bit address to a byte address you simply divide it by 8 (the number of bits in a byte); the quotient is the byte address and the remainder is the bit offset. Bit psects are not usually explicitly position via a `-p` linker option since they are part of a class, but if they were positioned in this way, a bit address would be used.

2.5 Split code for PICs

With the advent of the Highend PIC devices which allow external memory to be connected, it is now possible to have a program use both internal and external memory and have each area programmed separately. Any code which is often upgraded or modified can be placed in the external memory which can be easily re-programmed without having to replace the internal code. The internal code may also include a bootloader program if the external memory is, for example, flash memory and the external code is to be downloaded rather that manually installed. This tutorial explains the steps necessary to compile one program into multiple HEX files which can be used to program different memory areas.

Note that this procedure is relatively and necessarily complex, and that a good knowledge of the compiler and linker is assumed. Not all program features are available when compiling in the manner described here.

2.5.1 Overview

For the discussion below, the source code used to program the PIC device is assumed to consist of several sections referred to as *fixed*, *replaceable* and *extra* code. The fixed code is that code which is to remain static and not change during the life of the program. Typically this would be loaded into the internal, on-chip memory of the device. Replaceable code is that code which can change and can be recompiled. Typically this would be programmed in the external code space of the device or may even be downloaded into this space by some bootloader code. The extra code is also replaceable, but is code that is added at a later date after the initial compilation of the other source files.

During this tutorial, the code space used by the fixed code shall be referred to as *internal memory* since typically this code would be stored in the PIC's internal ROM space. The replaceable and extra code will be compiled for an area of memory referred to as *external memory*[9]. This memory can be ROM, RAM or flash, but will be referred to as ROM throughout this tutorial. Note that both fixed and replaceable

code will share the on-chip RAM for variables, although the code will be stored in separate memory areas.

To generate separate files for both the internal and external memory, the full compilation process must be performed separately for fixed and replaceable code. This creates a problem as that would normally require two completely separate programs, each with their own function **main()** and startup code. The other issue is that since the fixed and replaceable code are to interact, i.e. use variables and functions defined in the other section of the code, each part of the program must use symbol information defined by the other section of code. Even with these issues resolved, other steps must be taken to ensure that some of the advanced features of the compiler, e.g. overlapping of local and parameter memory area, are disabled to allow for future changes made by the extra code. Although the fixed and replaceable code are destined for different areas of ROM, it is actually the RAM that creates the most problems as this has to be split and shared by the different code sections. This tutorial attempts to show ways of overcoming these problems.

The steps involved in the compilation process are illustrated in Figure 2 - 2 on page 55. Multi-step compilation is handled by creating a special symbol-only object file. This is an object file that contains all the symbol (address) information relating to code and data, but the code and data themselves are not present. One of these object files is created for the fixed code and is compiled into the other sections of the program to allow it to have access to symbol information from the fixed code, but without actually having to have the code and data from that section duplicated. How these object files are created is discussed below.

2.5.1.1 Code limitations and strategies

This tutorial explains how to create code that has the following properties.

- After reset, execution will commence with run-time code compiled with the fixed code and situated in internal memory. This code will clear and/or initialise any global or static variables defined in the fixed code as normal.
- The internal run-time code will call a function called **main()**. This function is defined in the fixed code and will be located in internal memory. This function may call standard library routines, or call other functions defined in the fixed code in the usual way.
- **Main()** can jump to an assembly routine referred to as the *entry routine* which is defined in the replaceable code and which is situated in the external memory at an address that will not change during subsequent compilations of the replaceable code. The entry routine will be a modified copy of the standard run-time code which is situated in internal memory.
- This entry routine will initialise and/or clear any global or static variables defined in the

9. Microchip refer to off-chip memory as external memory. In this tutorial "external memory" may be used to refer to off-chip memory, or that memory used by the replaceable code; they will refer to the same area of memory.

replaceable code and will then call the "*external main"* function which is the first C function in the replaceable code to begin execution.

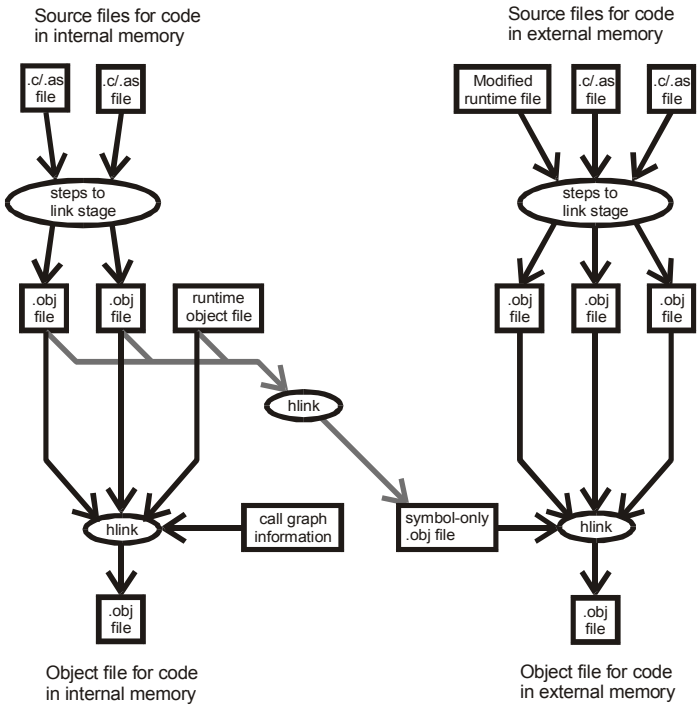
- The external main may call standard library routines, or call other functions defined in the replaceable code in the usual way. It may also call functions defined in the fixed code.
- Any library routine called by fixed code will be positioned in internal memory. Library functions called by the replaceable code, but which are not called by the fixed code, will be placed into the external memory.
- If execution of the external main function finishes, control will return back to the externally-located entry routine. Control will never return to **`main()`** within the fixed code other than after reset.
- The RAM areas (banks) of the PIC will be divided into sections: one for the fixed code's use; the other for use by the replaceable code. Common memory (if available) is shared between both the fixed and replaceable code.
- Constant data (including string literals) resides in ROM, but must be in one of two word ranges: 0h-3FFFh or 8000h-BFFFh. (See Section 0.1 on page 15 for a description of PIC addresses.) Fixed code will not be able to use **`const`** objects defined in replaceable code, and vice versa.
- Interrupt service routines (ISRs) for interrupts used must be defined in the fixed code. These ISRs may call other functions defined in the fixed code, or may call *external ISRs* that are defined in the replaceable code at an address that will not change during subsequent compilations of the replaceable code. Any functions called by the ISR must be done so using C code. Code to fully save and restore context will be generated and positioned in internal memory.

There are several methods that could be used to allow access to external code that may change at a later date, but the method described here is that an entry routine be defined in the replaceable code, whose address is always fixed. Once this function has been called, it can then call other replaceable routines as required. Even if new routines were added or the structure of the replaceable code changed significantly, the fixed code would not need to be aware of this as it would always execute replaceable code indirectly via the entry routine.

Another method would be to maintain some sort of table of function pointers, defined in the replaceable code and that would reside in the external memory space. This table would contain the addresses of the functions present in the external memory. If the addresses of functions in the replaceable code changed, the lookup table would reflect these changes. Thus, the functions in the external memory could be called indirectly by the internal code, going via the table. Since the table must be in ROM, it must be declared as **`const`**. The table itself would have to be located at a fixed address.

Another method of accessing external functions is to have each function located at a fixed address. Although this is a faster access method that does not require an entry routine or table of function

Figure 2 - 2 Overview of split code compilation



pointers, it limits the degree to which external functions can be modified in the future. These last two methods are not described in this tutorial.

2.5.2 Click here to begin

In this tutorial, examples will be applicable for the PIC17C756 processor which will be using external memory from word address 4000h to 5FFDh. Some adjustments may be necessary if you are using a different processor or need different memory settings. Note that the requirements for your system may not include all the aforementioned and that all the following steps may not need to be followed. The examples below are compiled using the `PICC` driver and `HLINK` (the linker). If you intend to use `HPDPIC`, you will need to create project files for most steps. It is recommended that you use `PICC` instead of `HPDPIC` to compile the source.

The method of code compilation described here requires that the linker options are adjusted from their default settings. This is easily done if using `HPDPIC` as the **Linker Options...** dialogue under the **Make**

menu can be edited and saved in the project. The tutorial section has a description of how to create a linker command file if you will be driving the linker explicitly from the command line - see Section 0.2.5 on page 30. Create projects files for each step of the compilation if you are using HPDPIC, or batch/ command files if using the compiler application on the command-line. Once these files are created, compilation will only take a few commands. Most of the work is associated with actually selecting the correct options to include in the files.

The code for internal and external memory may each be contained in as many files as required, and these files may be either C or assembler. Some code restrictions apply and are described in the following sections.

The following steps have bulleted sections which summarize the action required. Additional explanations are also included to explain why the action is needed.

2.5.2.1 Removing errors

It is strongly suggested that before you attempt to compile the code as described below that you simply compile the entire code together to first remove any syntax or linker errors. Debugging code whilst compiling using the method described here will be extremely difficult. Observation of the map file after this will also give you some insight into the amount of memory used by the entire program.

- Write your program as if it is a single unit, however place code in separate files. Call the external main routine and external interrupt functions directly, using a standard function call. Table 2 - 11 on page 56 shows part of some trivial, test code showing fixed code calling the external main function.
- Compile all the source files in the program (fixed and replaceable) as one program. For this example we might compile the source files as follows.

Table 2 - 11 Removing errors

Fixed code	Replaceable code

`void`
`main(void)`
`{`
`  int a, b;`
 	
`  strcpy(array, "test");`
`  a = strlen(array);`
`  b = i_two(a, 8);`
 	
`  ext_main(); // call external main`
`}`
	`void`
	`ext_main(void)`
	`{`
	`  int e, f;`
	
	`  f = 1;`
	`  replace_fred = i_one(f);`
	`  fixed_fred = i_two(f, 5);`
	`  e = strlen(array);`
	`  strcat(array, "another");`
	`}`

```
picc -17c756 -o -zg -mtest.map -asmlist f_main.c f_ext.c r_entry.c r_ext.c
```

Fix any errors or suspicious warnings. The map file `test.map` will give an indication of the amount of memory used by the entire program. Look at the section marked UNUSED ADDRESS RANGES.

- Make backup copies of the source files as they are. These files may be useful if you need to add extra code at a later date and you wish to compile the new or modified code as one program.

2.5.2.2 Preparing the code for split compilation

You now need to make some changes to your source code to allow the compilation to performed in two stages. Instead of calling the external main function, the code will now jump to (as opposed to call) the entry routine (external run-time code). How this code is generated is described further on. This entry routine will then jump to the external main function. In essence, the code for one program is split into two, each having there own run-time code.

- Decide upon the address at which the external entry routine will be located. This address cannot change at a later date so ensure that the position allows for any future changes that need to be made to the replaceable code. Remember, this is the address of the external run-time code, not the external main function. In this example the entry routine is to be placed at word address 0x4100.
- Replace the function call to the external main with a long jump to the entry routine, for example:

```
asm("ljmp 4100h");
```

Alternatively this routine could be called indirectly from C code, but this would use another level of stack unnecessarily. Jumping to the entry routine does, however, mean that the external, replaceable program cannot return back to this point once it has finished. If you intend to have the replaceable code return back to the fixed code after its execution, then this will have to be an indirect function call. Such a call might look like:

```
(*((void(*)(void))0x4100))();
```

or alternatively from in-line assembler:

```
asm("fcall 4100h");
```

With regard to interrupts, the approach used in this example is to have the ISR in fixed code simply call an external ISR which is an externally-located function that will do all the work required for the interrupt. The functions the ISRs call must be defined as ordinary functions, *not* as **`interrupt`** functions. This prevents two return-from-interrupt instructions being executed and there being two context saves.

- Decide where you are to place the external ISRs defined in the replaceable code. These addresses

must remain fixed throughout the life of the program.

- Replace any explicit function calls to external ISRs that were in the code used to remove errors with indirect calls. These must be calls, not jumps. The examples above used to call the entry routine may be used.

2

- Force the code for the external ISRs to be placed at a fixed, absolute address. To do this, it is best to have each of these routines in a separate file. At the top of each file place a pragma similar to the following.

```
#pragma psect text%%u=isr08
```

Here, the name `isr08` will be the psect in which the code for the external interrupt will reside. This will be positioned explicitly by the linker later in the compilation. Use different psect names for each routine. The entry routine is already in a unique psect name - `init` - and so no action is required to position it at this point. The entry routine is described later in this tutorial.

Table 2 - 12 on page 58 shows the fixed and replaceable interrupt code used in this example. Here, the fixed ISR code simply calls the external ISR in the replaceable code and does nothing else. The macro **`EXT_ISR08`** has been used. This would need to be defined as the address upon which you have decided to place the external ISR. When an interrupt occurs (interrupt vector 08h in this example) the interrupt function **`isr08()`** in fixed code will be called by the processor, this will call the external ISR, **`ext_isr08()`**, which will perform the required tasks. Control will then pass back to **`isr08()`** and then to the code that was interrupted. This method allows the operations of the ISR to change in future.

There is an important difference between calling the external ISR using C code rather than using

Table 2 - 12 Interrupt functions

Fixed code	Replaceable code

`void interrupt`
`isr08(void) @ 0x08`
`{`
`  ((void (*)(void))EXT_ISR08)();`
`}`
	`#pragma psect text%%u=isr08`
	
	`void ext_isr08(void)`
	`{`
	`  fixed_fred = 0x55;`
	`}`

assembler. Having in-line assembler to **`fcall`** the constant address of the external ISR would work, in that the call would be made correctly, but the context saving would be different. The code generator will try to save only those registers and objects needed by the ISRs. If the ISR calls another function, the code generator will see what objects that function uses and save those as well. If the code generator cannot determine what the called function uses, it assumes a worst case and saves everything. The code

generator cannot scan through in-line assembler code for object usage or function calls[10], hence if the external ISR is called from within assembler source the code generator will not be able to determine that this has taken place and will save only those objects used in the ISR, but not those used in the external ISR. In the above example, that would be virtually nothing. If you want the compiler to save everything before the call, then you have to let the code generator know that another function has been called. Calling a function from C (even if it is an indirect call) does just this. Since the external ISR is compiled in a separate module, the code generator cannot determine the external ISR's contents and so a full context save will occur. If the ISR saves all registers before commencing, the external ISR can change at a later time and still work, even if it uses more objects than the version that was initially used.

2.5.2.3 Creating the object files

With the source files completed, compilation can begin.

- Compile all of the source files for both the internal and external areas to relocatable object files. This can be easily done using PICC. Use the **-c** option and whatever other options are required. For example, using full optimization:

```
picc -17c756 -O -Zg -C f_main.c f_ext.c r_main.c r_ext.c
```

  This will create an object file for each source file listed with names: `f_main.obj` and `f_ext.obj` etc. If you require some source files to compiled with different options, they may be compiled separately.

If the code generator believes that there is only one page[11] of internal memory available, it will use a shorter form of function call. This cannot be allowed to happen as the fixed code will be calling the entry routine in external memory and a separate page. In addition, the replaceable code may be calling routines that will reside in the internal memory. The PIC17C756 has more than one page of on-chip ROM so far functions calls are used by default and the discussion following is not applicable. The 17C752, however, has only one internal page of ROM and so this issue becomes relevant.

If there is only one page of memory, the code generator does short calls to any functions that are called in C code. If the option `-ROM4000-5FFD` was included[12] to the above command line, the code generator is told that external memory in addition to the internal memory is present, and that it must use calls of a longer form. These are encoded as an **`fcall`** pseudo instruction in the assembler code and expand to multiple PIC instructions which handle the page bits as well as the call. Thus, if you are using a chip with only one page of internal memory, use the above option when compiling the source files to object files to force all calls to be of the longer form.

10. It is the job of the assembler to read assembler code, not the code generator.
11. The PIC processor has *banks* of RAM and *pages* of ROM.
12. The addresses are not important, as long as they are outside the page of memory already present.

Note that at this point, since the linker is not being executed (because of the **-c** option) the addresses of functions will not be determined. Although the -ROM option does affect the addresses options passed to the linker, here the -ROM option, if necessary, only affects the way functions are called and has no other affect. For this reason, the addresses used in the option are not important, as long as they represent an address range in addition to that already present.

If you intended to generate source-level debugging information use the -G option here. If you are using MPLAB to run the code, use the -FAKELOCAL option.

2.5.2.4 Creating the replaceable code's call graph information file

You now need to create a file that will contain information needed by the internal code to prevent local variables and parameters from being overlapped in memory.

The linker has the ability to overlap a function's auto and parameter areas with those of other functions if these functions are never active at the same time. This feature dramatically reduces the amount of RAM used by a program. The way it determines whether the functions are active is by following the function calls. One consequence of this is that if the linker believes that a C function is never called, it will overlay that function's local variable (auto) area and parameter area of memory since it thinks they are never used. Since the fixed code will be compiled without the replaceable code present, the linker will never see the fixed routines being called from the replaceable code. If these routines in fixed code are only ever called from the replaceable code, then their parameter and auto areas will be overlapped. This is clearly undesirable.

The contents of the call graph information file is to tell the linker, via special assembler directives, that these functions are being called. This will force the linker to use unique areas of memory for its parameter and auto areas. If you have a collection of functions in the fixed code that can be called from the replaceable code, you will need a directive for each function to declare that it has been called. The **FNCALL** assembler directive does this job.

- Create an assembler file which will be referred to as the calling graph information file.
- Add **GLOBAL** directives for **main()** and all of the function names that are called by the replaceable code, or may be called by the extra code added in the future.
- Add to this file one **FNCALL** directive for each of these functions, and indicate that these functions are called by the function **main()**.

Here is an example of what might be placed into this assembler file.

```
GLOBAL _main
GLOBAL _f_one
GLOBAL _f_two
FNCALL _main, _f_one
FNCALL _main, _f_two
```


It does not matter whether **main()** really does call these other functions or not, all this directive will do is tell the linker that **f_one()** and **f_two()** are called.

Notice that since this is assembler code, the C identifiers have had an *underscore* character prepended to their name.

- Compile the call graph information file to an object file. The options are of little importance (other than the -C and processor type) since this file will generate no code, only function records. Use a command like:

  ```
  picc -17c756 -C callgrph.as
  ```

2.5.2.5 Creating the symbol-only object file from the fixed code

If you are using PICC, you will now need to make a linker command file. This file will be copied and modified for all the stages that follow. Instructions to edit the command file are described in the tutorial Section 0.2.5 on page 30.

- Create an initial linker command file which contains the required linker command-line arguments by running PICC in verbose mode and redirecting the output to a file. Use the following command to generate the command file.

  ```
  picc -17c756 -Mf.map -L-f -L-i -V f_main.obj f_ext.obj callgrph.obj > fs.lnk
  ```

  The -L option has been used to add the -f and -i options to the linker arguments. These two options create a symbol-only object file and ignore undefined symbols, respectively. The -M option creates a map file with the name f.map. The -V selects verbose mode. Output from this command will be placed into the file fs.lnk.

Note that we are not actually concerned with what the linker does at this point; only generating the command file which is used in subsequent steps. Ignore any errors or warnings regarding the object files. If you include the options listed above, most of the linker arguments placed into the command file will be correct and not need changing. Obviously the file names in the above command line, for example f_main.obj, will need to be changed to those you are using.

- Edit the generated file as per the tutorial section mentioned above so that only the linker arguments remain. This file will be copied and changed for the subsequent steps. You may wish to make a backup copy of this file without the changes that we will make following.

The remainder of the compilation steps use the linker and linker command files that will have the linker's command-line arguments.

- Edit the linker command file so that the output object file (after the `-O` option) has a suitable name, e.g. `f_sym.obj`. The symbol file's name after the `-H+` option is not important in this step as it will not be used.

Although data objects defined in fixed and replaceable code will use their own memory addresses, symbols associated with function parameters have to be shared. Each section of the program must know about these symbols and their value cannot change. Here is why: If a function, **`f_one()`** for example, has parameters, it defines the symbol **`?_f_one`** which is used to access the area where its parameters will be stored. The compiler will generate code for this function which reads from this symbol's address (with an offset) when the function accesses its own parameters. In this way the address of the parameter area will form part of the code for the function. If another function calls **`f_one()`**, then it must load the parameter area of **`f_one()`** before the call, so this function also needs the address of **`?_f_one`**. So this address also forms part of the code for the calling function. If you have a routine in the replaceable code calling a function in the fixed code and this function has parameters, then both fixed and replaceable code must know and encode the parameter area address. The functions in the fixed code will always read from where they think their parameters are stored, even if the calling function has mistakenly stored the parameters at another address.

- Determine the amount of RAM that the fixed code requires for all available memory banks. To ensure that fixed code and replaceable code do not destroy each other's RAM space, it is best to alter the RAM bank classes during the link step so that each only knows about the area of memory that it requires. To determine the amount of RAM used by the fixed code, run the linker using the command script that you are currently editing. For example:

  ```
  hlink < fs.lnk
  ```

  This will generate a map file with the name that you specified. Look at this map file and go to the UNUSED ADDRESS RANGES listing. Listed for each bank of memory will be one or more ranges of memory that have *not* been used. From this you can determine the memory that was used, remembering that not all the banks are available for general-purpose use, i.e. the special function registers take up memory in some of the banks. Common memory areas on the PIC are shared by many parts of a program so do not adjust these addresses in the command file.

- Adjust the RAM address range linker options in the same linker command file so that now the available memory is limited to only that memory required by the fixed code. The available memory for each bank is specified in class options in the linker arguments contained in your linker command file. The classes are called: `BANK0`, `BANK1`, `BANK2` and `BANK3`, (assuming there are four banks) and there will be a `-A` option which defines each of these. For example, the default `BANK0` class definition for the PIC17C756 is:

```
-ABANK0=20h-FFh
```

If, for example, you determined that the fixed code used 10 bytes of RAM in bank 0, then you could adjust the class range in your linker command file so that only addresses 20 through 2F were available:

```
-ABANK0=20h-2Fh
```

This will mean that the fixed code has access to the lower areas of RAM. Later we will inform the replaceable code that it has access to the remaining upper areas of memory. Re-run the linker as above and re-check the map file. If you have made the available space exactly the same number of bytes as the program requires, the `BANK0` entry in the UNUSED ADDRESS RANGES will disappear. If you have not made enough bytes available, the linker will produce errors. You will need to do this for all RAM banks which your fixed code uses.

- Ensure that the available ROM memory is all internal. You should not need to adjust the linker options if you used the `PICC` command line above to generate the linker command file as this command did not use the `-ROM` option. ROM memory is specified by the `CODE` class, and for the PIC17C756 the linker class option should read:

```
-ACODE=0-1FFFh,2000h-3FFFh
```

Each range specified a ROM page. The `ROMDATA` class can remain as it is. By default it is: 0-3FFFh. This will mean that all initialised data psects (`idata_`*n* psects) and all **`const`** objects (including string literals) for the fixed code will be placed in internal memory. Any fixed code will be able to refer to these objects, but they will not be accessible by replaceable code.

- Now that the linker command file is finished, you can use your edited linker command file to create the object file that contains symbols only for all the fixed source files. For example:

```
hlink < fs.lnk
```

Ignore any warnings for undefined symbols produced by this command, however, there should be no errors. This should produce a file called, for example, `f_sym.obj`.

It is important to realise that although the symbol-only object file will not contain code or data, the linker has performed its usual task of relocation of psects and resolution of symbols. This is the only way in which it can determine the final addresses of symbols used in the code. The addresses in the object file produced here will be passed to and used by the replaceable code so before continuing you may want to check that the addresses generated look reasonable. You can do this by looking at the map file produced or using the DUMP utility and observing the object file directly.

- Localize the global symbols from the object file associated with the run-time code. This will enable you to have a duplicate copy of run-time code in external memory which can initialize and clear variables defined in the replaceable code. Use the utility `XSTRIP` which is located in the

BIN directory of your distribution.[13] Use a command line similar to:

```
xstrip -start,_exit,intlevel0,intlevel1,copy_data,clear_ram,copy_bank0,clear_bank0 f_sym.obj
```

This will remove the **GLOBAL** flag from the comma-separated list of symbols after the "-" in the above command line. In other words these symbols will then become local and will not be defined when the replaceable code is compiled. The name of the object file is the last argument to this command. This file is overwritten with the new object file data. If your fixed code also uses banks other than zero, then you will also need to add to the list of symbols: **copy_bank1**, **clear_bank1** etc. There is no problem if you include a symbol that is not present in the object file. If you do not include a symbol that is used in both the fixed and replaceable run-time code, you will be issued with a multiply defined symbol error when you compile the replaceable code later on. If this happens, take a note of any multiply defined symbols, return to this step and include them in the list of symbols passed to XSTRIP.

2.5.2.6 Creating the absolute object file for the fixed code

This step will actually create the absolute object file for the internal memory from the fixed code.

- Copy and re-name the linker command file that you used to generate the symbol-only object file for the fixed code in the previous step.
- Remove the -F and -I options from this file.
- Change the file name in the -O option so that a unique output object file will be created, for example f_abs.obj. Do not overwrite the symbol-only object file created above.
- Change the name of the map file (after the -M option) and symbol file's name (after the -H+ option) to something unique and suitable.
- Create the absolute object file by running HLINK and using this new command file. For example:

```
hlink < f_abs.lnk
```

This will create an object file called, for example, f_abs.obj. This is the file that will be used to program the internal memory and which contains all the fixed code.

2.5.2.7 Making a modified run-time file

The external memory will contain a duplicate copy of the run-time code so that it can clear and initialise objects defined in the replaceable code. The run-time code associated with the fixed code cannot be used to do this since the addresses it uses to perform the clear/copy are linked into its code. These addresses are not applicable for the replaceable code's variables.

13. This is a relatively new application. If your BIN directory does not contain this application, contact technical support: support@htsoft.com

- Make a copy of the standard run-time code which is contained in a file called `picrt66x.as` and which is located in the SOURCES directory of your distribution. You may wish to rename it, for example `extrt.as`, so it is never confused with the standard run-time module.
- Copy the file `sfr.h` from your SOURCES directory to your working directory.
- Replace *all* references to the symbol `_main` with the assembler name of the external main function which will be the function executed after the entry routine has completed its job. You may want to call this function **`ext_main()`** or some similar name. Thus **`_main`** would be replaced with **`_ext_main`**.
- Compile this modified run-time file to an object file as follows.

```
picc -17c756 -c -p extrt.as
```

  This will create a file called `extrt.obj`.

2.5.2.8 Creating the absolute object file for the external memory

Now the absolute object file for the external memory is created.

- Make a copy of the linker command file that you used to generate the absolute object file for the fixed code in a previous step.
- Change the file name in the `-O` option so that a unique output object file will be created.
- Change the name of the map file (after the `-M` option) and symbol file's name (after the `-H+` option) to something unique and suitable.
- Add the name of the symbol-only object file produced from the fixed code (and which has had some of its symbols undefined) to the list of object files that are to be linked in the command file.
- Change the name of the run-time object file (for example `picrt714.obj`) to that of your modified run-time code, for example `extrt.obj`.
- Change the names of the object files for the fixed source code to be those for the replaceable sources' object files.
- Remove the call graph information file from the list of object files to be linked.
- Alter the RAM memory settings so that the memory not being used by the fixed code is available for the replaceable code. For example if you fixed code used addresses 20 to 2F in bank 0, you should change the `BANK0` class in the linker options to:

```
-ABANK0=30h-FFh
```

  so that the replaceable code will not touch the memory areas used by the fixed code.
- Change the ROM memory settings so that only external memory is defined. This is specified in

the CODE class. For this example, the definition might be:

```
-ACODE=4000h-5FFDh
```

- Change the ROMDATA class so that it is using the external memory ROM space: 8000h-BFFFh. This will mean that all initialized data psects (idata_*n* psects) and all **const** objects (including string literals) for the replaceable code will be placed in external memory. Any replaceable code will be able to refer to these objects, but they will not be accessible by fixed code.

- Add linker options for any functions that need to be positioned at absolute addresses. This will be the entry routine and external ISRs. For example the psect which contains the external interrupt could be positioned by the following option:

```
-pisr08=4000h
```

  if, for example, you decided previously that it should be placed at word address 4000h.

- The init psect used by the entry routine in the modified run-time code will already have a linker option present, but you will need to change its address. For example, the default linker option to position the initialization psects is, by default:

```
-pintcode=08h,powerup=00h,init%2000h,end_init,clrtext
```

  can be changed to:

```
-pintcode=08h,powerup=00h,init=4100h,end_init,clrtext
```

  shows the new address (4100h) added without altering the other psect names in the -P option. There should ne no powerup or interrupt code present, so the linker options for powerup and intcode psects can be left as they are.

- Create the absolute object file by running HLINK and using this command file.

```
HLINK < r_abs.lnk
```

2.5.2.9 Creating HEX files from the absolute object files

- Convert the two absolute object files created to *Intel* HEX files by using the OBJTOHEX application. For example, for the fixed code, it can be converted as follows.

```
objtohex -i -16,2 f_abs.obj fixed.hex
```

Perform this for the replaceable code and you then have two HEX files: one for internal memory; one for the external space.

2.5.2.10 Creating COD files for debugging

COD files are used for debugging on some emulators/simulators, for example MPLAB.

- Create a COD file using the CROMWELL utility. For example:

  ```
  cromwell -f -m -P17C756 fixed.hex f_abs.sym replace.hex r_abs.sym -ocod
  ```

  A single COD file can be produced from both applications, as above. Alternatively two separate files can be produced from the HEX and SYM file from the fixed or replaceable code, if desired. You can examine the contents of a COD file by using a -D option with CROMWELL and the name of the COD file to ensure that the correct information is present within them.

2.5.2.11 Checking the result

Here are some things that can be checked to ensure that the program will operate correctly. Symbol and psect information can be found in the map files produced by the linker.

- Ensure that the addresses of all the fixed code are in internal memory and that the addresses of RAM symbols defined by the fixed code are within the RAM space that you allocated to the fixed code. Some psects may be present in the incorrect memory area, but if they are of length zero[14], then they have no content and can be ignored.
- Ensure that the addresses of all the replaceable code are in external memory and that the addresses of RAM symbols defined by the replaceable code are within the space that you allocated to the replaceable code.
- Ensure that the local and parameters areas for each function are not at the same address, that is, for example, **?_f_one** and **?a_f_one** are not at the same address.
- Ensure that functions that are active at the same time, have auto or parameter areas which do not overlap each other, that is, for example, **?_f_one** may have the same address as **?_f_main** provided that **f_main()** does not call **f_one()** or vice versa. Note that the symbols defined for the auto/parameters areas are the starting address for these areas. The size of the area is determined by the total size of local variables of parameters passed. If a function accepts a one-byte parameter then that parameter is passed to the function in the W register and so it will not use any parameter memory.
- Ensure that the value assigned to the entry routine and external ISRs, as indicated in the symbol table in the map file, are what you specified.
- Ensure that symbols that are shared, for example the parameter area symbols for functions defined in the fixed code, but which may be called by the replaceable code, have the same value in the map file for both the fixed and replaceable code.

14. The length of a psect can be determined from the symbols in the map file __Lxxxx and __Hxxxx, where xxxx is the name of the psect and the symbols represent the lower and higher bound of the psect, respectively.

Most importantly, after you are satisfied that all is working as it should, you will need to save the HEX, object files and linker command files that were used to build the project. It would be wise to make a read-only copy of the entire directory in which you developed the code.

2.5.3 Adding extra code

At a future date, it may be necessary to make changes or add to the replaceable code. In particular, you may need to change replaceable code that already exists or add new code that was never present in the original compilation.

Remember that you cannot change the addresses of the entry routine or any external ISRs, or any objects or routines that were accessed by the fixed code. You must also use the symbol-only object file that was created from the actual fixed code that is present. Any attempt to use a another symbol-only object file with different addresses will most likely fail.

The steps involved in creating the new external HEX file are:

- Remove errors from the new or altered code. Use the backup copies of the code you made when you first checked the program for errors.
- Create new object files for the replaceable and extra code.
- Make any changes to the replaceable absolute linker command file, such as adding new object files to the list of files to be linked.
- Create the new absolute object file.
- Create new HEX file.
- Create new debugging files, if required.
- Check the result using the map files.

2.5.4 Using MPLAB

MPLAB can be used to simulate the operation of the split code. If you intend to use MPLAB make sure that you have specified that you are running in extended microcontroller mode in **Hardware...** under **Processor Setup** under the **Options** menu.

Because there are two hex files to be downloaded into MPLAB, you'll have to make sure that MPLAB doesn't clear the program memory when downloading the second hex file. To do this, select **Environment Setup...** from the **Options** menu and make sure **Clear Memory on Download** is not selected. You should now be able to download both HEX files. If an MPLAB project is open, MPLAB will also automatically load any relevant COD files and you will be able to debug as usual.

If you show the list of available symbols in MPLAB, you will notice that there are many duplicates. This is simply because there are multiple symbols of the same name loaded into MPLAB. In most cases these

symbols should be assigned the same address. The only exceptions will be for those symbols that you passed to XSTRIP. For these there will be symbols with different addresses. Undefined behaviour will result if you use any of these multiply-defined symbols in debugging.

Using HPDPIC

3.1 Introduction

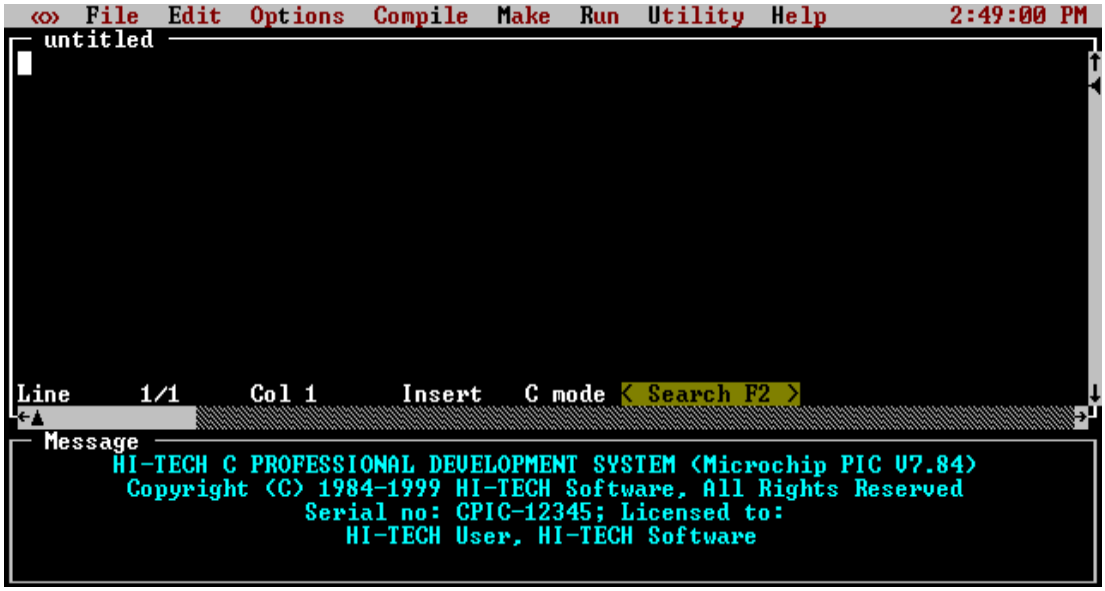
This chapter covers HPD, the **HI**-TECH C **P**rogrammer's **D**evelopment environment. It assumes that you have already installed PICC. If you have not yet installed your compiler, go to the *Quick Start Guide* and follow the installation instructions there.

HPDPIC is the version of HPD applicable to the PICC compiler.

3.1.1 Starting HPDPIC

To start HPDPIC, simply type *hpdpic* at the MS-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.

Figure 3 - 1 HPDPIC Startup Screen



The initial HPDPIC screen is broken into three windows. The top window contains the menu bar, the middle window the HPDPIC 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.

HPDPIC 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, HPDPIC 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*, HPDPIC 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, HPDPIC 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 HPDPIC is invoked with the command:

```
hpdpic 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 HPDPIC 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 Environment variables

To use the HI-TECH C compiler, only one DOS environment variable need be present. This is a path to a temporary location where intermediate files may be stored. The variable is called TEMP and it should be automatically placed into you autoexec.bat file when the compiler is installed.

As this path is used to specify the location of temporary files, it should not be very long or the command lines that are generated to drive the compiler may exceed the DOS command line size limit. Typically C:\TEMP is chosen as the temporary file path.

3.2.2 Hardware Requirements

HI-TECH Windows based applications will run on any MS-DOS based machine with a standard display 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 HPDPIC 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 HPDPIC set the screen display mode on EGA or VGA displays to show more than 25 lines. The option */screen:xx* where *xx* is one of 25, 28, 43 or 50 will cause HPDPIC 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 HPDPIC exits. The selected number of lines will be saved in the *hpdpic.ini* file and used for subsequent invocations of HPDPIC unless overridden by another */screen* option.

HPDPIC will recognize and use any mouse driver which supports the standard INT 33H interface. Almost all modern mouse drivers support this 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, HPDPIC will still work with the mouse, but the **Setup...** dialog in the **<<>>** menu will not work.

3.2.3 Colours

Colours are used in two ways in HPDPIC. 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 74 shows the colours and their values, whilst Table 3 - 2 on page 74 shows the attributes and their meaning.

Any colours are valid for the foreground but only colours 0 to 7 are valid for the background. Table 3 - 3 on page 75 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 *hpdpic.ini* file. The first value in a colour definition is the foreground 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 *hpdpic.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.

3.2.4 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.

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

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

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 75

3.2.4.1 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

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: ; ...

current context of the application. For example, the **Save project** item will not be selectable if no project has been loaded or defined.

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

3.2.4.2 Mouse Menu Selection

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.

3.2.4.3 Menu Hot Keys

When browsing through the menu system you will notice that some menu items have *hot key* sequences displayed. For example, the HPDPIC 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 HPDPIC *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 HPDPIC key equivalents is shown in Table 3 - 5 on page 77.

3.2.5 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 rear-most window is brought to the front and the other windows shuffle 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 HPDPIC compiler error window will not be made front-most if it is clicked. Instead it will

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

Table 3 - 5 HPDPIC 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	Optimisation menu
Alt-D	Run DOS command
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
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

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.

3.2.6 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 a 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 HPDPIC 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, 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.

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 activating the menu system 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.

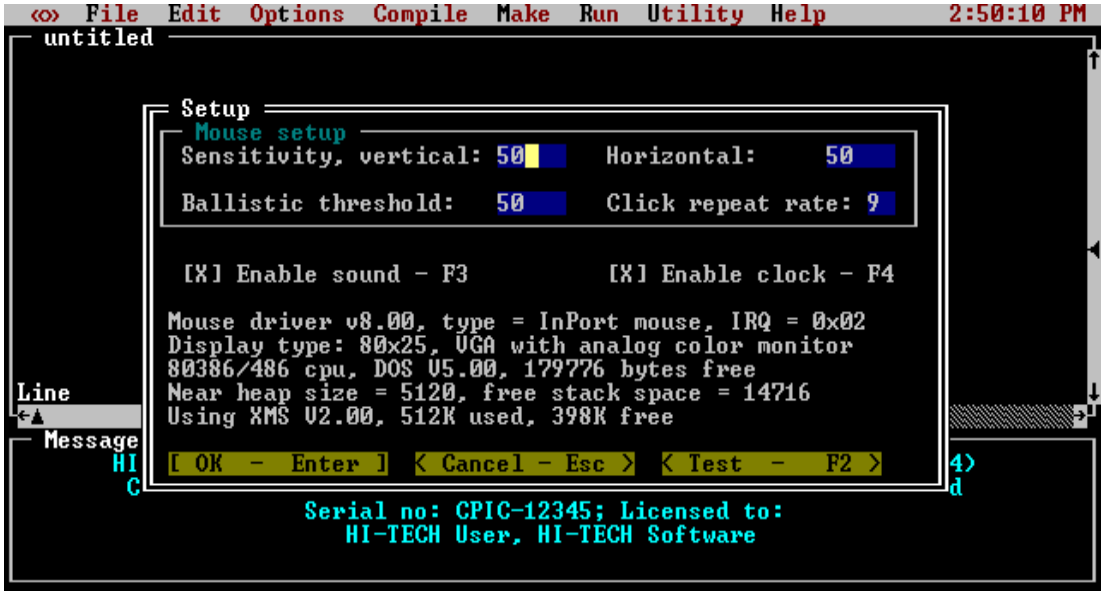
3.2.7 Buttons

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 HPDPIC error window (Figure 3 - 5 on page 84) contains a number of buttons, to select HELP you would either click the left mouse button on it or press *F1*.

3.2.8 The Setup menu

If you open the system menu, identified by the symbol **<<>>** on the menu bar, you will find two entries: the **About HPDPIC ...** entry, which displays information about the version number of HPDPIC; and the **Setup ...** entry. Selecting the Setup entry opens a dialog box as shown in Figure 3 - 2 on page 80. This box displays information about HPDPIC'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 HPDPIC's initialisation file, so they will be reloaded next time you run HPDPIC. The sound and clock settings will also be stored in the initialisation file if you select OK.

Figure 3 - 2 Setup Dialogue



3.3 Tutorial: Creating and Compiling a C Program

This tutorial should be sufficient to get you started using HPDPIC. It does not attempt to give you a comprehensive tour of HPDPIC'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 HPDPIC, 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> hpdpic
```

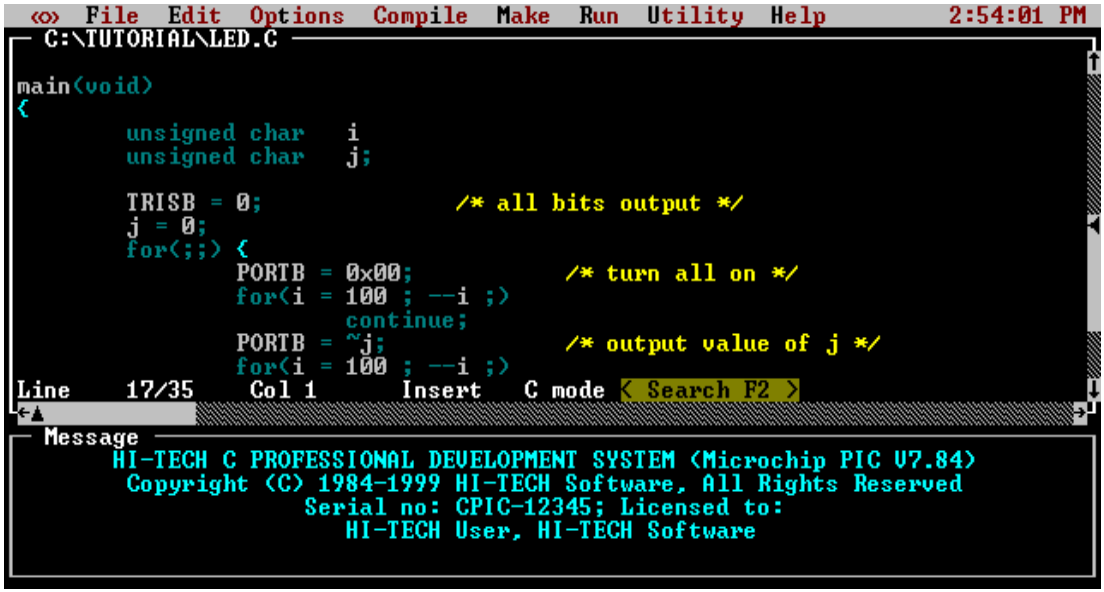
You will be presented with the HPDPIC startup screen. At this stage, the editor is ready to accept whatever text you type. A flashing block cursor should be visible in the top left corner of the edit window. You are now ready to enter your first C program using HPDPIC. This will be the LED program shown in the *Quick Start Guide*. This program may also be found as *led.c* in the *samples* directory.

Type in the LED program, pressing *enter* once at the end of each line. You can enter blank lines by pressing *enter* without typing any text. Intentionally leave out the semi-colon at the end of the first line in main, as follows:

```
...
main(void)
{
        unsigned      i
        unsigned char j;
        ...
```

Figure 3 - 3 shows the screen as it should appear after entry of the LED program.

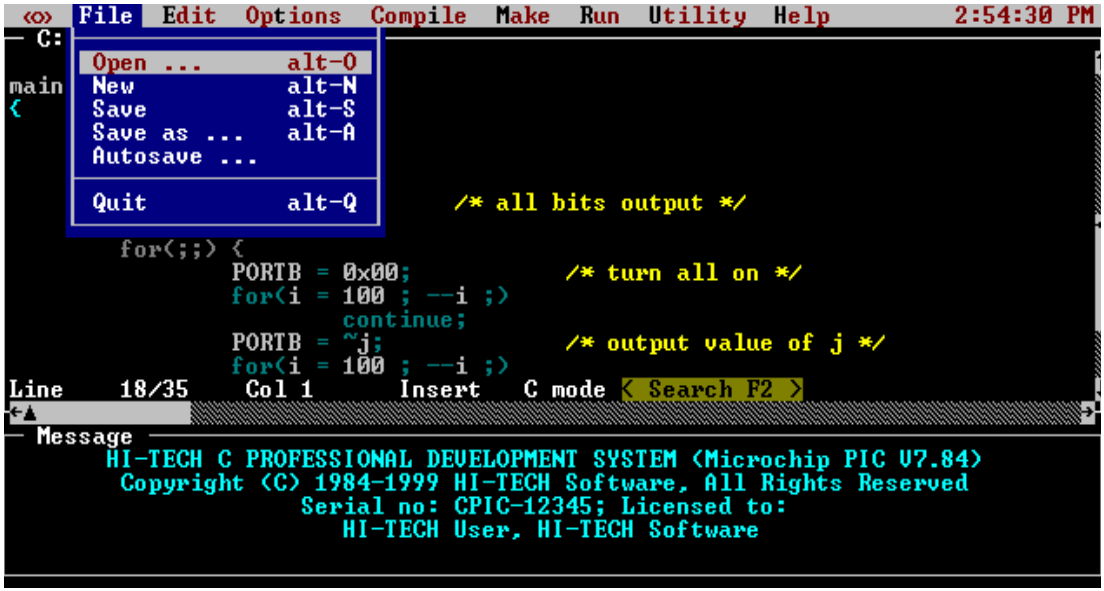
Figure 3 - 3 LED Flashing program in HPDPIC



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 82)

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

Figure 3 - 4 HPDPIC File Menu



- 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) 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 HPDPIC 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.

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 *myled.c*. Type *myled.c* and then press *enter.* There should be a brief period of disk activity as HPDPIC saves the file.

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.

You will be asked to enter some information at this point:

Select Processor
: Select the processor you are using.

Floating Point Type
: Select 24-bit (default).

Optimisation
: Select Full Optimisation, with the default Global Optimisation Level of 1

Output File Type
: Select Bytecraft .COD or Intel .HEX.

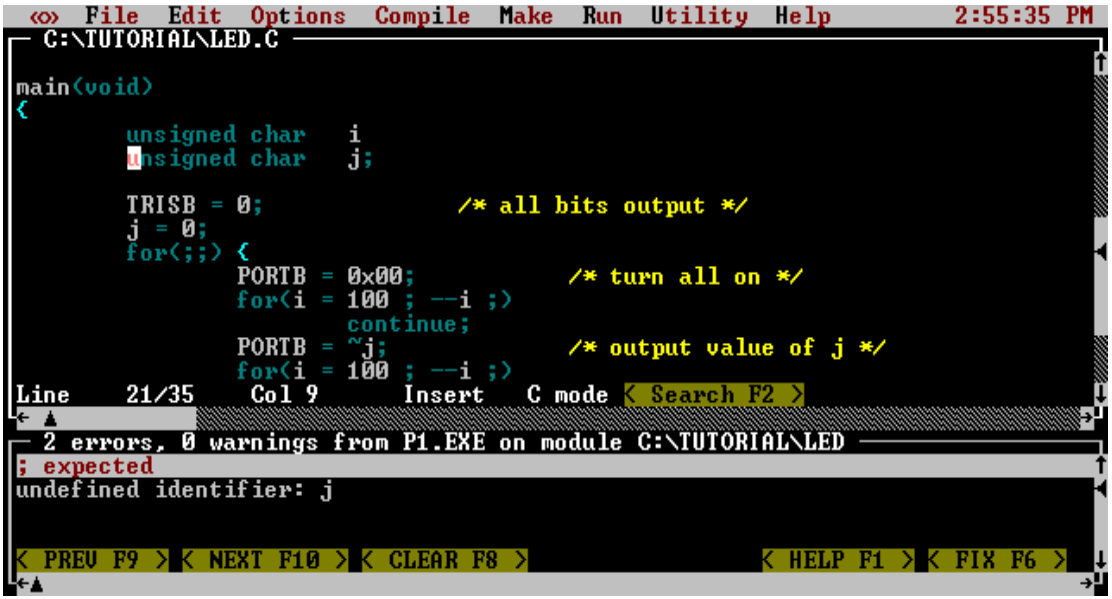
This time, the compiler will not run to completion. This is because we deliberately omitted a semicolon on the end of a line, in order to see how HPDPIC handles compiler errors. After a couple of seconds of disk activity, you should hear a “splat” noise. The message window will be replaced by a window containing a number of buttons and describing two errors as shown in Figure 3 - 5 on page 84.

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 *cgpic.exe*. *Cpp.exe* and *hlink.exe* can also return errors. In this case, the error window frame contains the message:

```
2 errors, 0 warnings from p1.exe
```

indicating that pass 1 of the compiler found 2 fatal errors, with the second error being caused by the first one. It is possible to configure HPDPIC so that non-fatal warnings will not stop compilation. If only

Figure 3 - 5 Error window



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 **`unsigned i`** declaration. 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 any 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 (*cgpic*), assembler (*aspic*) and linker (*hlink*) are handled. You may dismiss the error explanations by selecting the HIDE button (press *escape* or use the mouse).

In this instance HPDPIC 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 HPDPIC 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, HPDPIC will move to the next error.

To manually correct the error, move the cursor to the end of the declaration 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. HPDPIC 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. HPDPIC 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 HPDPIC menu system, editor and project facility.

3.4 The HPDPIC editor

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

The HPDPIC 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.
- WordStar *ctrl-k* command. See the section Keyboard Commands on page 87, 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 87, for a list of the valid *ctrl-q* commands.
- *Insert* indicates that 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 - 7, “Editor keys,” on page 88 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 HPDPIC 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 HPDPIC 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 - 8, “Block operation keys,” on page 89 lists the WordStar compatible block operations which are available**.**

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

Table 3 - 7 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	

Table 3 - 8 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

Begin Block **ctrl-K B**

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

End Block **ctrl-K K**

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

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

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

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

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

Block Hide Toggle **ctrl-K H**

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

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

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

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

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

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

Indent
This operation is available 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 HPDPIC 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 HPDPIC 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 HPDPIC menus

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

The About HPDPIC dialog displays information on the version number of the compiler.

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.

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

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.

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 80

3.5.2 File menu

The **File** menu contains file handling commands, the HPDPIC **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 non-zero, then the current edit file will automatically be saved to a temporary file at intervals. Should HPDPIC not exit normally, e.g. if your computer suffers a power failure, the next time you run HPDPIC, it will automatically restore the saved version of the file.

Quit **alt-Q**
The **Quit** command is used to exit from HPDPIC 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.

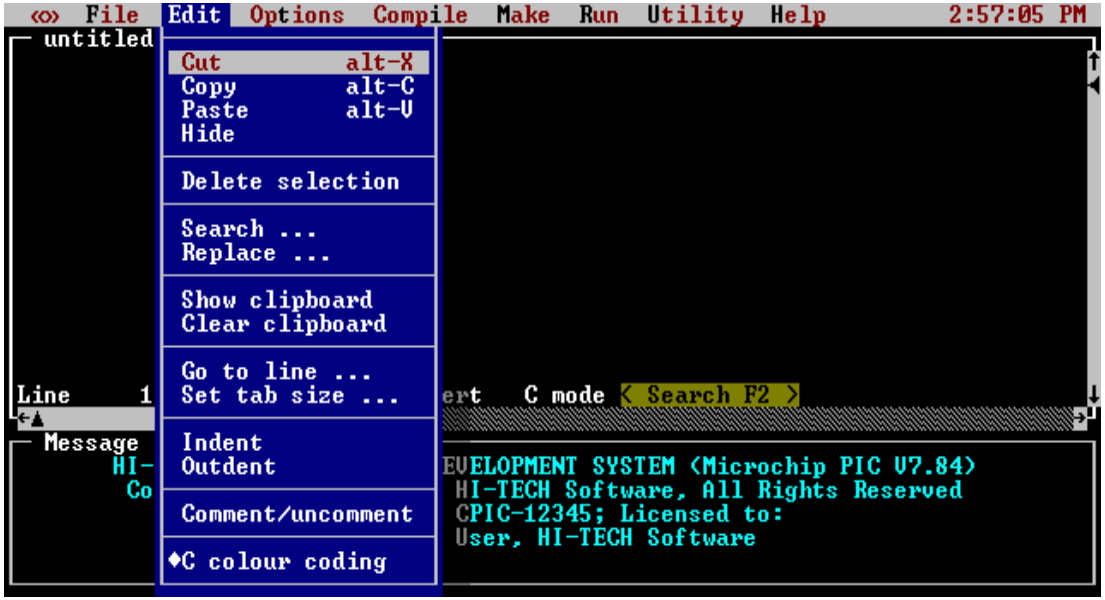
Clear pick list
This clears the list of recently-opened files which appear below this option.

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 in Figure 3 - 6.

Figure 3 - 6 HPDPIC Edit Menu



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 HPDPIC and how to select specific schemes, you should refer to the section, Colours on page 73.

3.5.4 Options menu

The **Options** menu contains commands which allow selection of compiler options, 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 - 7 on page 97..

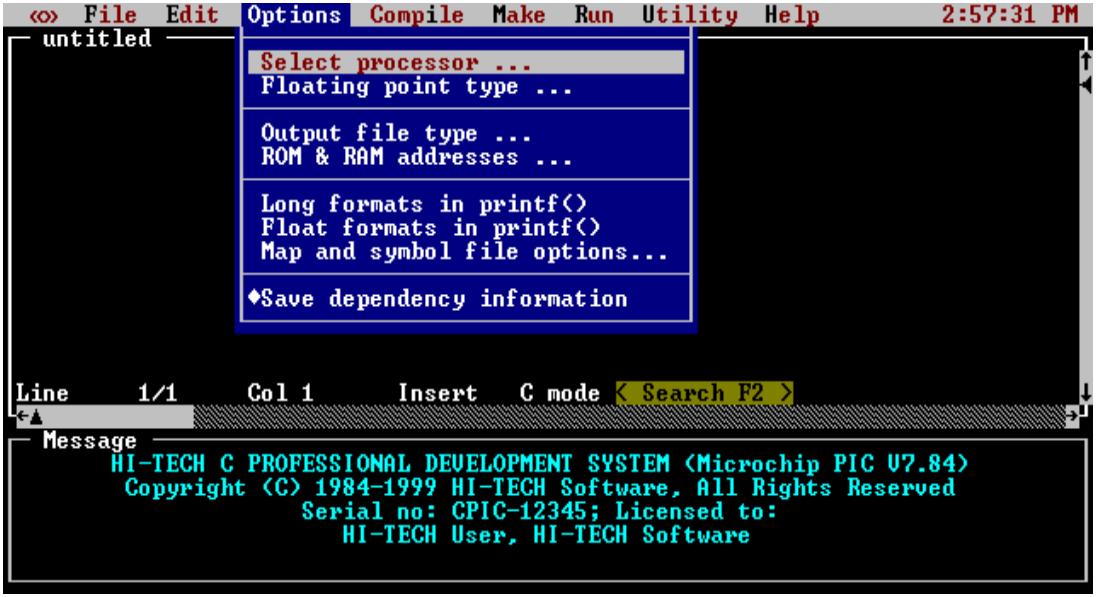
Select processor ...
This option activates a dialog box which allows you to select the processor type you wish to use. Three buttons in this dialogue allow you to swap between the Baseline, Midrange and High-End families[3]. The help string at the bottom of the dialogue indicates the amount of ROM space, in words, that each device contains. In addition the string indicates the number of RAM banks the device contains and the total amount of general-purpose RAM bytes available. This is the amount of RAM other than that used by the special function registers. If the help string displays "including common" then some of the RAM space is taken up by common memory which is used by the compiler for internal use.

Floating point type ...
This selects the format used for floating point doubles. The default format is the 24-bit truncated IEEE 754 format, but you may choose to use the 32-bit IEEE 754 format. For more information on these formats, see Floating Point Types and Variables on page 143.

Output file type ...
The default output file type is Bytecraft COD. The other choices are: Motorola S-Record HEX, Intel HEX, Binary Image, UBROF, Tektronix HEX, American Automation symbolic HEX and Intel

3. Note that the leading two digits on the PIC part name do not indicate the family of the device. For example, a 16C58 is a Baseline part and a 12C671 is a Midrange part. All Baseline parts have 12-bit instructions; Midrange: 14 bits; and High-End: 16 bit instructions.

Figure 3 - 7 Options Menu



OMF-51. This option will also allow you to specifiy that you want to create a library. A library can only be created from a project file.

ROM addresses ...
This menu is highlighted when a high-end PIC device is selected from the Select processor menu. This dialog allows you to specify the address ranges covered by external ROM devices. The addresses are used to store code and const data.

Long formats in printf()
This option is used to tell the linker that you wish to include a supplemental library which includes printf() with support for printing longs. See page 330 for more details on printf().

If you use this option the compiled code 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 include a supplemental library which includes printf() with support for printing longs and floats. See page 330 for more details on printf().

If you use this option the compiled code 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 MPLAB, 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.

Fake local symbols
This modifies the debug information produced to allow MPLAB to examine most local variables. It also adjusts source-level single stepping information to be that required by MPLAB. See also Section 5.16 on page 180.

MPLAB-ICD support
This button automatically adjusts the linker settings so that the output code is suitable for the MPLAB In-Circuit Debugger. This menu item is only highlighted if the selected processor has ICD capability.

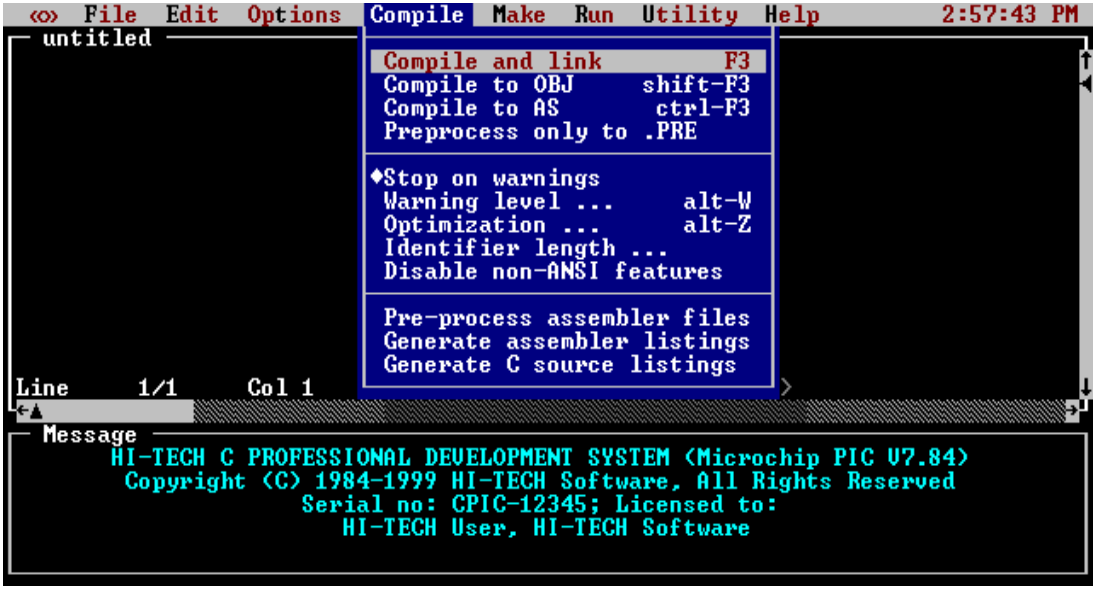
Save dependency information
With this checked (which is the default), dependency information is saved in the project file. This means that restarting the HPD is much faster for a large project.

3.5.5 Compile menu

The **Compile** menu, shown in Figure 3 - 8 on page 99, 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 other utilities 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 *led.c* would be compiled to *led.cod*.

Figure 3 - 8 HPDPIC Compile Menu



Compile to .OBJ **shift-F3**

Compiles a single source file to a *.obj* file only. The linker and objtohex are not invoked. The *.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*.

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.

Preprocess only to .PRE

This runs the source file through the C preprocessor. The output of this action is a *.pre* file of the same name as the source file.

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 PICC command.

Optimisation ... **alt-Z**

Selecting this item will open a dialog allowing you to select different kinds and levels of optimisation. 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.

Disable non-ANSI features

This option is used to enable strict ANSI conformance of all special keywords. HI-TECH C supports the special keywords such as *persistent* and *interrupt* which are used to prevent variables being cleared on startup, and to handle interrupts using C code. If this option is used, these keywords, for example, are changed to *__persistent* and *__interrupt* respectively so as to strictly conform to the ANSI standard. This is the same as the -STRICT option when compiling from the command line.

Pre-process assembler files

Selecting this item will make HPDPIC 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 *led.c* will produce a listing file called *led.lst*.

Generate C source listing

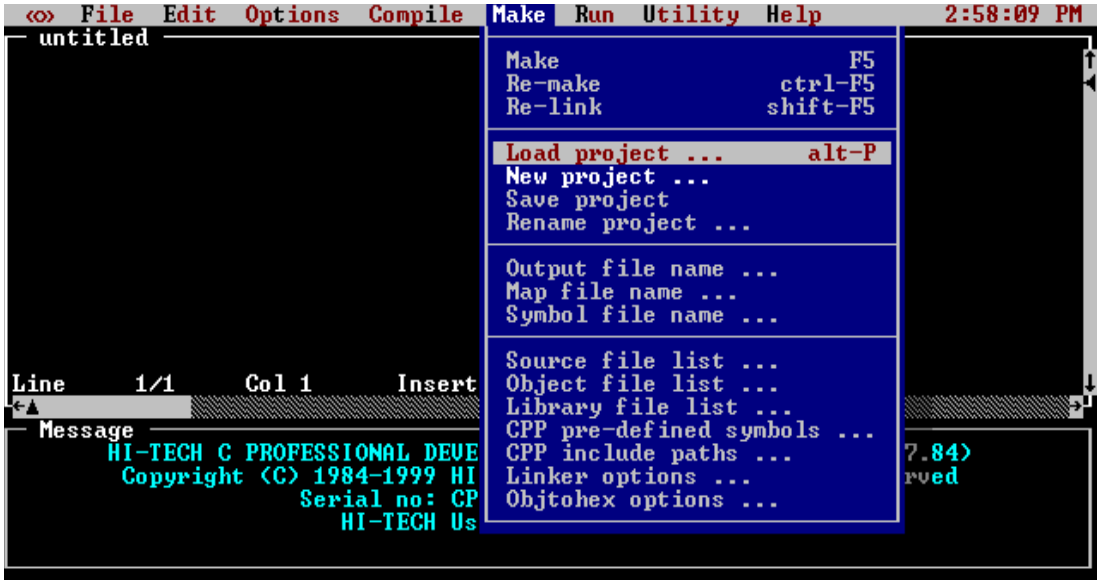
Selecting this option will cause a C source listing for each C file compiled. The listing file will be named in the same way as an assembler file (described above) but will contain the C source code with line numbers, and with tabs expanded. The tab expansion setting is derived from the editor tab stop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

You can only generate *either* a C source listing *or* an assembler listing.

3.5.6 Make menu

The **Make** menu (Figure 3 - 9 on page 101) contains all of the commands required to use the HPDPIC

Figure 3 - 9 HPDPIC Make Menu



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, HPDPIC will present several dialogs to allow you to set up options including the processor type and optimisation level.
- 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.

Make **F5**

The Make command re-compiles the current project. When Make is selected, HPDPIC re-compiles any source files which have been modified since the last Make command was issued. HPDPIC 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 object (*.obj*) files are current but the output file cannot be found, HPDPIC 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, HPDPIC will print a message in the message window indicating that nothing was done.

HPDPIC 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, HPDPIC 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 object files and then selecting **Make**.

Re-link **shift-F5**

The Re-link command relinks the current project. Any object 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 HPDPIC will present several dialogs to allow you to configure the project. These dialogs will allow you to select: processor type; float type; output file type; optimisation settings; and map and symbol file options. You will be asked to enter source file names via the *Source file list*.

Save project
This item saves the current project to a file.

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

Output file name ...
This command allows the user to select the name of the compiler output file. This name is automatically setup when a project is created. For example if a project called *prog1* is created and a COD file is being generated, the output file name will be automatically set to *prog1.cod*.

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. 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 MPLAB.

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 HPDPIC 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 module for the current processor. For example, if a project is generating code for the PIC16C54, by default this list will contain a runtime startoff module called *picrt200.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 library for the processor being used. For example, if the current project is for a PIC16C54, this list will contain the library *pic200-c.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 then the standard C library. 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 PCC. 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 is included, for example <*stdio.h*>, 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 HPDPIC to be modified. The default contents of the linker command line are generated by the compiler from information selected in the **Options** menu. **You should only use this command if you are sure you know what you are doing!**

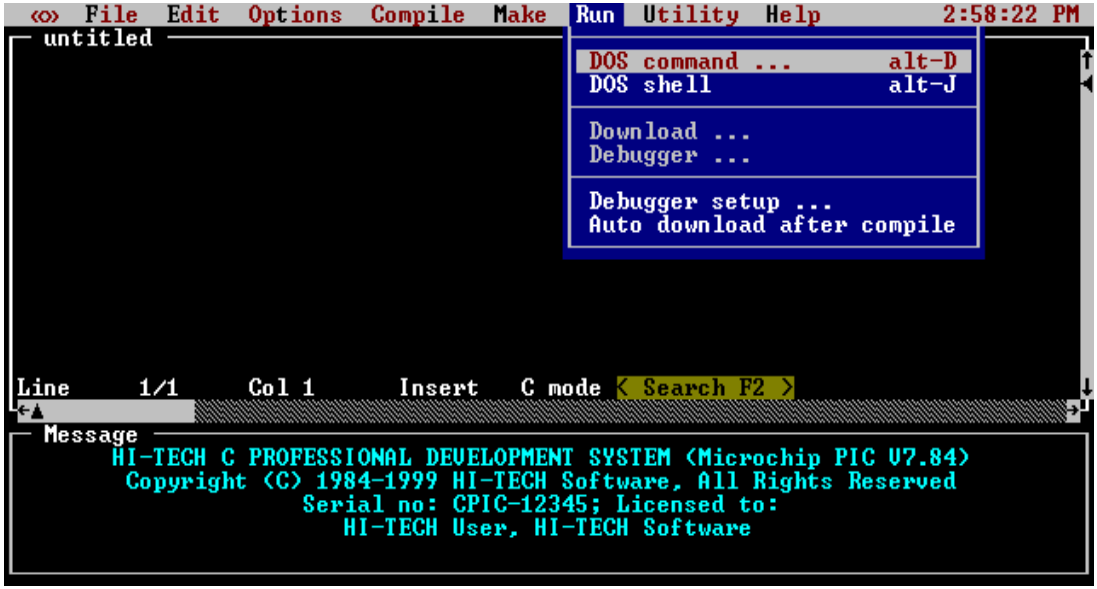
Objtohex options ...
This command allows the options passed to objtohex by HPDPIC to be modified. Normally you will not need to change these options as the generation of output files can be chosen in the Options menu. However, if you want to generate one of the unusual output formats which objtohex can 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 - 10 on page 105, contains options allowing MS-DOS commands and user programs to be executed.

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

Figure 3 - 10 HPDPIC Run Menu



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

DOS Shell **alt-J**
This item will invoke an MS-DOS *command.com* shell, i.e. you will be immediately presented with a MS-DOS prompt, unlike the **DOS command** item which prompts for a command. To return to HPDPIC, type *exit* at the MS-DOS prompt.

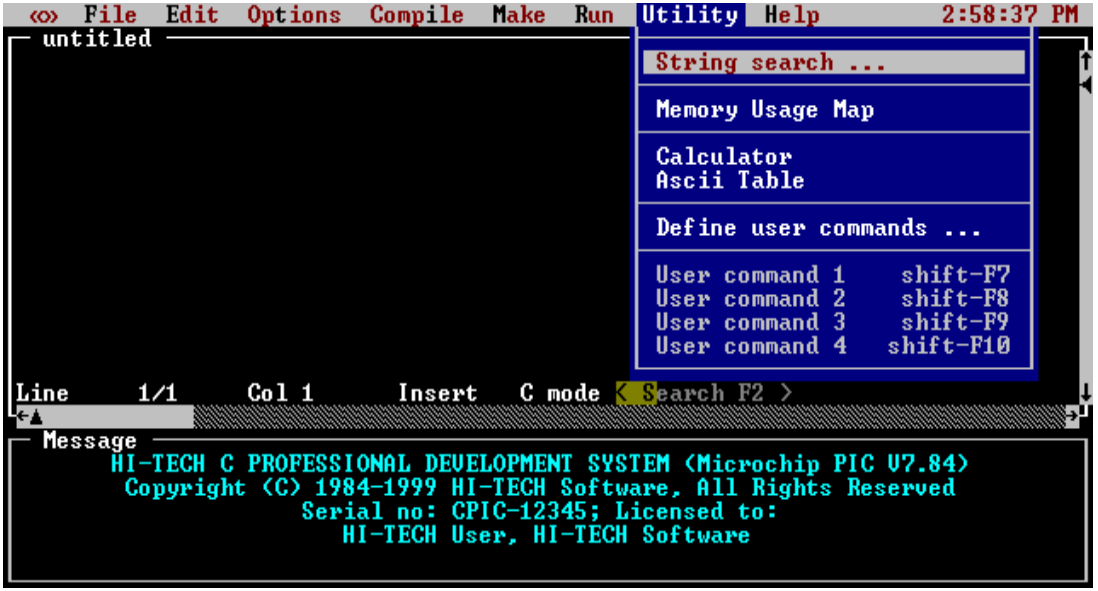
Other Options
All other options in this menu are for future enhancements to the compiler.

3.5.8 Utility menu

The **Utility** menu (Figure 3 - 11 on page 106) contains any useful utilities which have been included in HPDPIC.

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.

Figure 3 - 11 HPDPIC Utility Menu



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...
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 107. Each user-defined

Table 3 - 9 Macros usable in user commands

Macro name	Meaning
$(LIB)	Expands to the name of the system library file directory; eg *c:\ht-pic\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.

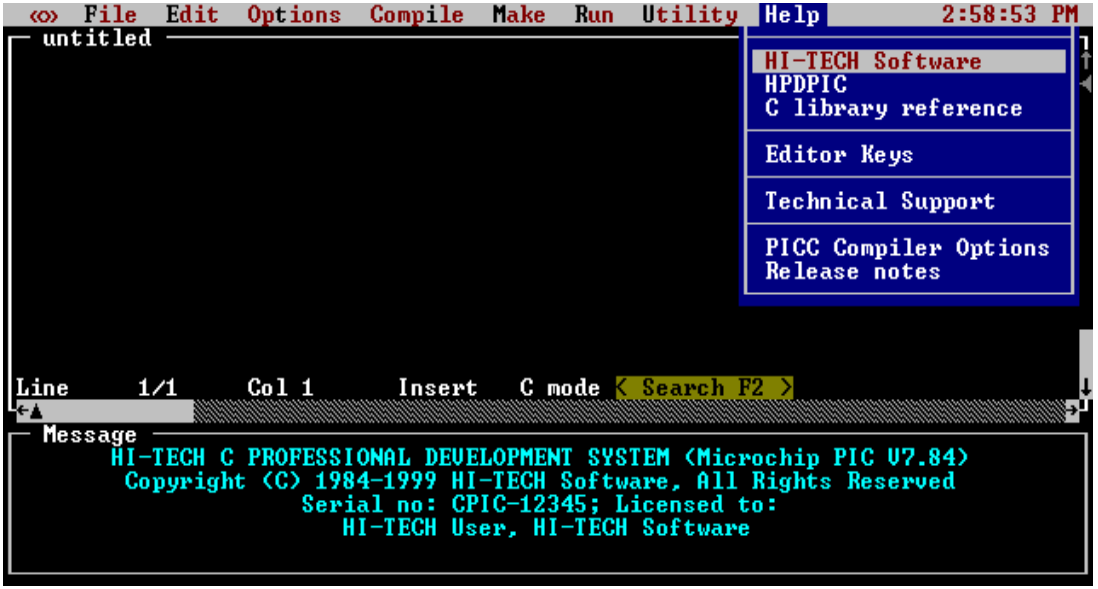
command has a hot key associated. They are *shift F7* through *shift F10*, for commands 1 to 4. When a user command is executed, the current edit file, if changed, will be saved to a temporary file, and the $(EDIT) macro will reflect the saved temp file name, rather than the original name. On return,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

3.5.9 Help menu

The **Help** menu (Figure 3 - 12 on page 108) contains items allowing you to obtain help about any topics listed.

On startup, HPDPIC 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

Figure 3 - 12 HPDPIC Help Menu



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

HI-TECH Software
This includes information on contacting HI-TECH Software and the licence agreement.

HPDPIC
This option produces a window showing all the topics for which help is available. Topics include Chip types, Compiler optimizations, Editor Searching, Floating point sizes, String search and User defined commands.

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 *tan* entry and double click on the reference to *asin*, you will be taken to the entry for *asin*.

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.

PICC Compiler Options
This option displays a window showing all the PICC 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.

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.

Command Line Compiler Driver

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

```
picc [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 120). Source files, object files and library files are distinguished by PICC solely by the file type or extension. Recognized file types are listed in Table 4 - 1. This means, for example, that an assembler file must always have a file type of .as (alphabetic case is not important).

Table 4 - 1 PICC File Types

File Type	Meaning
.c	C source file
.as	Assembler source file
.obj	Object code file
.lib	Object library file

PICC 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 PICC 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 PICC with a -C option to compile several source files to object files, then to create the final program by invoking PICC with only object files and libraries (and appropriate options).

4.1 Long Command Lines

The PICC driver is a 32-bit *Windows* application, thus it is able to process command lines exceeding 128 characters in length. The driver may be called from within a DOS batch file or passed options via a

command file. When using batch files, the command line to PICC must be contained on one line. Driver options may be spread over multiple lines in a command file by using a *space* character followed by a *backslash* "\" followed by a *return* to split the lines. For example a command file may contain:

```
-V -O -16F877 -UBROF -D32 \
file1.obj file2.obj mylib.lib
```

If this was in the file `xyz.cmd` then `PICC` would be invoked as:

```
PICC < xyz.cmd
```

Since no command line arguments were supplied, `PICC` will read `xyz.cmd` for its command line.

The command file may also be read by using the `@` symbol. For example:

```
PICC @xyz.cmd
```


4.2 Default Libraries

`PICC` will search the appropriate standard C library by default for symbol definitions. This will always be done last, after any user-specified libraries. The particular library used will be dependent on the processor selected.

The standard library contains a version of **`printf()`** that supports only integer 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 library containing the long version of **`printf()`**. For floating-point and long **`printf()`** support, use the `-Lf` option which will search the library containing the floating-point version of **`printf()`**. You do not need the `-Ll` option if you have specified the `-Lf` option.

4.3 Standard Run-Time Code

`PICC` will also automatically provide the standard run-time module appropriate. If you require any special powerup initialization, rather than replace or modify the standard run-time module, you should use the *powerup* routine feature (see page 137). If you don't want to use the included runtime startup code at all, then you can disable it with the *-NORT* option. See *4.4.29 on page 122* for further information.

4.4 PICC Compiler Options

The compiler is configured primarily for generation of ROM code. `PICC` recognizes the compiler options listed in Table 4 - 2 on page 113. The `PICC` command also allows access to a number of advanced compiler features which are not available within the HPDPIC integrated development environment.

Table 4 - 2 PICC Options

Option	Meaning
-*processor*	Define the processor
-A*spec*	Specify offset for ROM
-A-*option*	Specify -*option* to be passed directly to the assembler
-AAHEX	Generate an American Automation symbolic HEX file
-ASMLIST	Generate assembler .LST file for each compilation
-BANKCOUNT=*count*	Set number of banks to use
-BIN	Generate a Binary output file
-C	Compile to object files only
-CK*file*	Make OBJTOHEX use a checksum file
-CR*file*	Generate cross-reference listing
-D24	Use truncated 24-bit floating point format for doubles
-D32	Use IEEE754 32-bit floating point format for doubles
-D*macro*	Define pre-processor macro
-E	Define format for compiler errors
-E*file*	Redirect compiler errors to a file
-E+*file*	Append errors to a file
-FAKELOCAL	Produce MPLAB-specific debug information
-FDOUBLE	Enables the use of faster 32-bit floating point math routines.
-G*file*	Generate enhanced source level symbol table
-HELP	Print summary of options
-ICD	Compile code for MPLAB-ICD
-I*path*	Specify a directory pathname for include files
-INTEL	Generate an Intel HEX format output file (default)
-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
-MOT	Generate a Motorola S1/S9 HEX format output file
-MPLAB	Specify compilation and debugging under MPLAB IDE
-N*size*	Specify identifier length
-NORT	Do not link standard runtime module
-NO_STRING_PACK	Disables string packing optimizations
-O	Enable post-pass optimization
-O*file*	Specify output filename
-P	Preprocess assembler files
-PRE	Produce preprocessed source files
-PROTO	Generate function prototype information
-PSECTMAP	Display complete memory segment usage after linking

Table 4 - 2 PICC Options

Option	Meaning
-Q	Specify quiet mode
-RESRAM*ranges*	Reserve the specified RAM address ranges.
-RESROM*ranges*	Reserve the specified ROM address ranges.
-ROM*ranges*	Specify external ROM memory ranges available
-S	Compile to assembler source files only
-SIGNED_CHAR	Make the default char signed.
-STRICT	Enable strict ANSI keyword conformance
-TEK	Generate a Tektronix HEX format output file
-U*symbol*	Undefine a predefined pre-processor symbol
-UBROF	Generate an UBROF format output file
-V	Verbose: display compiler pass command lines
-W*level*	Set compiler warning level
-X	Eliminate local symbols from symbol table
-Zg*[level]*	Enable global optimization in the code generator

4.4.1 *-processor*: Define processor

This option defines the processor which is being used. Generally all 12, 14, 16 and 17 series Microchip PIC processors are supported, but for a complete current list see the HI-TECH web site (www.htsoft.com). You can also add your own processors to the compiler. For more information about this, see Processor Support on page 129.

4.4.2 -Aspec: Specify offset for ROM

The `-A` option is used to specify a base address for the ROM image. This option may be required with debugging tools that expect the ROM image to begin at an address other than zero. This option effects all ROM-based psects including interrupt vectors and constant data as well as code.

For high-end devices, if the base address is positioned in external memory, the `-ROM` option must be used to specifiy the range of external memory addresses available.

4.4.3 -A-option: Specify Extra Assembler Option

The `-A` option can also be used to specify an extra "-" option which will be passed directly to the assembler by `PICC`. If `-A` is followed immediately by any text starting with a "-" character, the text will be passed directly to the assembler without being interpreted by `PICC`. For example, if the option `-A-H` is specified, the `-H` option will be passed on to the assembler when it is invoked which will display constant values as hexadecimal values in the assembler output.

4.4.4 -AAHEX: Generate American Automation Symbolic Hex

The -AAHEX option directs PICC 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.

4.4.5 -ASMLIST: Generate Assembler .LST Files

The -ASMLIST option tells PICC 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.

4.4.6 -BIN: Generate Binary Output File

The -BIN option tells PICC 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.

4.4.7 -BANKCOUNT=*count*: Set number of banks to use

The -BANKCOUNT option will override the default number of banks a processor has with the supplied number in order for additional optimizations to take place. This option is useful in applications where not all of the processor's RAM banks are used. For example, if an application only requires two RAM banks but the processor it is intended for has four banks, then using the option -BANKCOUNT=2 may allow PICC to apply additional optimizations to reduce code size.

4.4.8 -C: Compile to Object File

The -C option is used to halt compilation after generating an object file. 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:

```
PICC -16C84 -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:

```
PICC -16C84 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`.

4.4.9 -CK*file*: Generate Check Sum

This option causes OBJTOHEX to use *file* for checksum specifications. See "Objtohex Options" on page 224. for further details.

4.4.10 -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`, `PICC` 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 `PICC` 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:

```
PICC -16C84 -CRmain.crf main.c module1.c nvram.c
```

4.4.11 -D24: Use 24-bit Doubles

This option is the default, causing the use of truncated 24-bit floating point format for doubles. See Floating Point Types and Variables on page 143 for more details.

4.4.12 -D32: Use 32-bit Doubles

This tells the compiler to use the IEEE754 32-bit floating point format for doubles. See Floating Point Types and Variables on page 143 for more details.

4.4.13 -D*macro*: Define Macro

The `-D` option is used to define a preprocessor 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, `-Dmacro` which is equivalent to:

```
#define  macro  1
```

or `-Dmacro=`*`text`* which is equivalent to:

```
#define  macro  text
```

Thus, the command:

```
PICC -16C84 -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
```

4.4.14 -E: Define Format for Compiler Errors

If the -E option is not used, the default behaviour 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.

4.4.14.1 Using the -E Option

Using the -E option instructs the compiler to generate error messages in a format which is acceptable to some text editors.

If the same source code as used in the example above 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.

4.4.14.2 Modifying the Standard -E Format

If the -E option does not meet your editor's requirement, you can redefine its format by setting two environment variables: HTC_ERR_FORMAT and HTC_WARN_FORMAT. These environment variables are in the form of a *printf-style* string in which you can use the specifiers shown in Table 4 - 3.

Table 4 - 3 Error Format Specifiers

Specifier	Expands To
%f	Filename
%l	Line number
%c	Column number
%s	Error string

The column number is relative to the left-most non-space character on the source line. Here is an example of setting the environment variables:

```
set HTC_WARN_FORMAT=WARNING: file %f; line %l; column %c; %s
set HTC_ERR_FORMAT=ERROR: file %f; line %l; column %c; %s
```

Using the previous source code, the output from the compiler when using the above environment variables would be:

```
ERROR: file x.c; line 4; column 9; undefined identifier: xFF
```

Remember that if these environment variables are set in a batch file, you must prepend the specifiers with an additional percent character to stop the specifiers being interpreted immediately by DOS, e.g. `%%f`.

4.4.14.3 Redirecting Errors to a File

Error output, either in standard or `-E` format, can be redirected into files using UNIX or MS-DOS style standard output redirection. The error from the example above could have been redirected into a file called `errlist` using the command:

```
PICC -16C84 -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:

```
PICC -16C84 -E x.c >> errlist
```

4.4.15 -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, `PICC` 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:

```
PICC -16C84 -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:

```
PICC -16C84 -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:

```
PICC -16C84 -Eproject.err -O -Zg -C main.c
PICC -16C84 -E+project.err -O -Zg -C part1.c
PICC -16C84 -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
```


4.4.16 -FDOUBLE

This option is used to enable the use of faster 32-bit floating point routines. These alternative routines make floating point multiplication and division many times faster than the default routines, but do require more ROM and RAM space. These routines are only available for Highend processors and for 32-bit double types.

4.4.17 -FAKELOCAL

This option should be used in conjunction with the -G option to produce debug information that is specific to *Microchip's* MPLAB. It will allow the user to debug using variables local to a function, e.g. in watch windows. The debug information associated with source-level single stepping is also modified so that a better correlation between the source and instructions in the program memory window is obtained. See also Section 5.16 on page 180.

4.4.18 -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.

Note that all source files for which source level debugging is required should be compiled with the -G option. The option is also required at the link stage, if this is performed separately. For example:

```
PICC -16C84 -G -C test.c
PICC -16C84 -C module1.c
PICC -16C84 -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.

4.4.19 -HELP: Display Help

When used with no other options present on the command line, the `-HELP` option displays information on the `PICC` options.

4.4.20 -ICD

This option can be used to indicate that the output code is to be downloaded to the MPLAB In-Circuit Debugger. It will make appropriate adjustments to the linker options required by the ICD. When used, this option defines a macro called **`MPLAB_ICD`**.

4.4.21 -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:

```
PICC -16C84 -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.

4.4.22 -INTEL: Generate INTEL Hex File

The `-INTEL` option directs `PICC` to generate an *Intel* HEX file if producing a file with `.HEX` extension. This option has no effect if used with a `.BIN` file.

4.4.23 -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 additional versions of standard library functions to be accessed. For example when compiling for the PIC16C65 processor, the floating point version of **`printf()`** can be linked by searching the library `PIC411-F.LIB` using the option `-LF`.

The argument to `-L` is a library keyword to which the prefix PIC; numbers representing the processor range, number of ROM banks and the number of RAM banks; and suffix `.LIB` are added. Thus the

option -LL will, for example, scan the library PIC411-L.LIB and the option -LXX will scan a library called PIC411-XX.LIB. All libraries must be located in the LIB subdirectory of the compiler installation directory.

If you wish the linker to scan libraries whose names do not follow this naming convention or whose locations are not in the LIB subdirectory, include the libraries' names on the command line along with your source files. Alternatively, the additional libraries can be specified in the HPDPIC **library file list** menu, or the linker may be invoked directly.

4.4.23.1 Printf with Additional Support for Longs and Floats

For *Midrange* and *High-End* processors, you can use a version of **printf()** and related functions which, in addition to supporting the printing of integers, can support printing of *long integers*. Another version supports integers, long integers and *floats*. For complete information on **printf()**, see page 330.

To use the version of **printf()** which has additional support for longs, you must include a supplementary library by using the following option:

```
-Ll
```

To use the version of printf which has additional support for longs and floats, use the option:

```
-Lf
```

In the above options, l and f are merely specifying the *Library Type* as described in the section Standard Libraries on page 135.

4.4.24 -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 PICC. 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 PICC. For example, if the option -L-FOO is specified, the -FOO option will be passed on to the linker when it is invoked. It is important to note, there is no space between -L and -FOO. 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 HPDPIC 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 **xtraram**. 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 (**xtraram psect**) were to reside at address 50h, you could use the PICC option -L-Pxtreram=50h as follows:

```
PICC -16C84 -L-Pxtraram=50h 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 `PICC` as `-L-N`.

4.4.25 -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.

4.4.26 -MOT: Generate Motorola S-Record HEX File

The `-MOT` option directs `PICC` 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.

4.4.27 -MPLAB: Compile and Debug using MPLAB IDE

The `-MPLAB` option informs the HI-TECH C that both compilation and subsequent debugging will be performed from within the *Microchip* MPLAB IDE. This option turns on source level debugging (`-G`), turns on the `-FAKELOCAL` option to allow enhanced source and variable tracking, and adjusts the compiler's error message format (`-E`) to be that required by the MPLAB IDE.

If compilation is performed under a separate make facility, but debugging is performed under the MPLAB IDE, then the `-G`, `-E` and `-FAKELOCAL` options can be used separately.

4.4.28 -N*size*: Identifier Length

This option allows the C identifier length to be increased from the default value of 31. Valid sizes are from 32 to 255. The option has no effect for all other values.

4.4.29 -NORT: Do Not Link Standard Runtime Module

Using this option will not link in the standard runtime startup module. The user should then supply their own version of the runtime startup module in the list of input files on the command line. Even if the required startup module does not contain executable code, it will almost certainly require symbol and psect definitions for successful compilation, so this module cannot simply be omitted completely. The source for the standard runtime module is supplied in the SOURCES directory of your distribution and this should be used as the basis for your own runtime module.

4.4.30 -NO_STRING_PACK: Disable string packing optimizations

This option will disable string packing optimizations. Use this option if the emulator or debugging environment you are using does not support processors which can read their own program memory.

4.4.31 -O: Invoke Optimizer

-O invokes the post-pass optimizer after the code generation pass.

4.4.32 -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:

```
PICC -16C84 -Otest.bin prog1.c part2.c
```

will produce a binary file named test.bin.

4.4.33 -P: Pre-process Assembly Files

-P causes the assembler files to be pre-processed before they are assembled.

4.4.34 -PRE: Produce Pre-processed Source Code

-PRE is used to generate pre-processed C source files with an extension .PRE. It may be useful to ensure that macros expand to what you think they should. Use of this option can also create C source files which do not require any separate header files. This is useful when sending files for technical support.

4.4.35 -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. The .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>
add(arg1, arg2)
int *   arg1;
int *   arg2;
{
        return *arg1 + *arg2;
}
```

```
void printlist(int * list, int count)
{
        while (count--)
                printf("%d ", *list++);
        putchar('\n');
}
```

If compiled with the command `PICC -16C84 -PROTO test.c`, `PICC` 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 */
```


4.4.36 -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
---------|-------------------------|--------------------
powerup  | Power on reset code     | $0000 - $0003
init     | Initialization code     | $0004 - $0007
end_init | Initialization code     | $0008 - $000B
clrtext  | Memory clearing code    | $000C - $0012
text     | Program and library code | $0745 - $074C
text1    | Program and library code | $074D - $075D
ftext    | Arithmetic routine code  | $075E - $0769
float_te | Arithmetic routine code  | $076A - $07FF
rbss_0   | Bank 0 RAM variables    | $0020 - $0022
```

```
temp     | Temporary RAM data       | $0070 - $007B

Memory Usage Map:

Program ROM   $0000 - $0012  $0013 (  19) words
Program ROM   $0745 - $07FF  $00BB ( 187) words
                             $00CE ( 206) words total Program ROM

Bank 0 RAM    $0020 - $0022  $0003 (   3) bytes
Bank 0 RAM    $0070 - $007B  $000C (  12) bytes
                             $000F (  15) bytes total Bank 0 RAM
```

4.4.37 -q: Quiet Mode

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

4.4.38 -RESRAM*ranges[,ranges]*

The -RESRAM option is used to reserve a particular section of RAM space. The address ranges must be specified in HEX. The syntax for this option is a comma separated list of address ranges. For example:

```
-RESRAM20-40
```

This will reserve the RAM addresse range from 0x20 to 0x40.

4.4.39 -RESROM*ranges[,ranges]*

The -RESROM option is used to reserve a particular section of ROM space. The address ranges must be specified in HEX. The syntax for this option is a comma separated list of address ranges. For example:

```
-RESROM1000-10FF,2000-20FF
```

This will reseve the ROM address ranges 0x1000 to 0x10FF and 0x2000 to 0x20FF.

4.4.40 -ROM*ranges*

If external ROM space is available, code can be allocated into these areas with this option. The syntax for this option is a comma separated list of address ranges. For example:

```
-ROM1000-1FFF,2000-2FFF
```

If external ROM is used, the compiler will make use of the **`fcall`** and **`ljmp`** instructions to access it. For more information about **`fcall`** and **`ljmp`**, *see "Additional Mnemonics" on page 184.*

4.4.41 -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:

```
PICC -16C84 -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.

4.4.42 -SIGNED_CHAR: Make Char Type Signed

Unless this option is used, the default behaviour of the compiler is to make all character values and variables **unsigned char** unless explicitly declared or cast to **signed char**. This option will make the default char type **signed char**. Any unsigned char will have to be explicitly declared **unsigned char**.

The range of **signed char** is -128 to +127 and the range of **unsigned char** is 0 to 255

4.4.43 -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 **bank1** type qualifier). If the -STRICT option is used, these keywords are changed to include a double underscore at the beginning (e.g. **__bank1**) 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).

4.4.44 -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.

4.4.45 -U*macro*: Undefine a Macro

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

4.4.46 -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.

4.4.47 -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.

4.4.48 -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.

4.4.49 -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.

4.4.50 -Zg*[level]*: Global Optimization

The -Zg option invokes global optimization during the code generation pass. This can result in significant reductions to code size and internal RAM usage. This optimizer is less critical than the post-pass optimizer, but can still significantly reduce the code size.

Global optimization attempts to Optimize register usage on a function-by-function basis. It also takes advantage of constant propagation in code to avoid un-necessary accesses to memory.

The default level for this option is 1 (the least optimization). The level can be set anywhere from 1 to 9 (the most optimization). The number indicates how hard the optimizer tries to reduce code size. For PICC, there is usually little advantage in using levels above 3.

Features and Runtime Environment

PICC 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.

5.1 ANSI Standard Issues

5.1.1 Divergence from the ANSI C Standard

PICC diverges from the ANSI C standard in one area: function recursion.

Due to the PIC's hardware limitations of no easily-usable stack and limited memory, function recursion is unsupported.

5.1.2 Implementation-defined behaviour

Certain sections of the ANSI standard have implementation-defined behaviour. This means that the exact behaviour of some C code can vary from compiler to compiler. Throughout this manual are sections describing how the PICC compiler behaves in such situations.

5.2 Processor-related Features

PICC has many features which relate directly to the baseline, midrange and high-end family of processors. These are detailed in the following sections.

5.2.1 Processor Support

PICC supports a wide range of processors. Additional processors may be added by editing `picinfo.ini` in the LIB directory. This file is divided into baseline, midrange and high-end sections, but user-defined processors should be placed at the end of the file. The header of the file explains how to specify a processor. Newly added processors will be available the next time you compile by selecting the name of the new processor on the command line in the usual way.

5.2.2 Configuration Fuses

The PIC processor's configuration fuses may be set using the **__CONFIG** macro as follows:

```
#include <pic.h>
__CONFIG(x);
```

where ***x*** is the word that is to be the configuration word. Special named quantities are defined in the header file appropriate for the processor you are using to help you enable the required features. Here is an example for a PIC16C5x:

```
__CONFIG(WDTDIS & XT & UNPROTECT);
```

This will disable the watchdog timer, specify an XT crystal and leave the code space unprotected. Check the appropriate header file and ensure that all the configuration bits are correctly specified in the **__CONFIG** macro for your application before programming the device.

Note that the individual selections are ANDed together. Any bits which are not selected in these macros will remain unprogrammed. You should ensure that you have specified all cbits correctly to ensure proper operation of the part when programmed. Consult your PIC datasheet for more details.

The **__CONFIG** macro does not produce executable code and should be placed outside function definitions.

5.2.3 ID Locations

Some PIC devices have locations outside the addressable memory area that can be used for storing program information, such as an ID number. The **__IDLOC** macro may be used to place data into these locations. The macro is used in a manner similar to:


```
#include <pic.h>

__IDLOC(x);
```

where ***x*** is a list of nibbles which are to be positioned in to the ID locations. Only the lower four bits of each ID location is programmed, so the following:

```
__IDLOC(15F0);
```

will attempt to fill four ID locations with the decimal values: 1, 5, 15 and 0. The base address of the ID locations is specified by the `idloc` psect which will be automatically assigned an address dependent on the type of processor selected.

5.2.4 EEPROM Data

For those PIC devices that support external programming of their EEPROM data area, the **__EEPROM_DATA()** macro can be used to place the inital EEPROM data values into the HEX file ready for programming. The macro is used as follows.

```
#include <pic.h>

__EEPROM_DATA(0, 1, 2, 3, 4, 5, 6, 7);
```

The macro accepts eight parameters, being eight data values. Each value should be a byte in size. Unused values should be specified as a parameter of zero. The macro may be called multiple times to define the required amount of EEPROM data. It is recommended that the macro be placed outside any function definitions.

The macro defines, and places the data within, a psect called `eeprom_data`. This psect is positioned by a linker option in the usual way.

This macro is not used to write to EEPROM locations during run-time. The macros **`EEPROM_READ()`** and **`EEPROM_WRITE()`**, and the function versions of these macros, can be called to read from, and write to, the EEPROM during program execution.

5.2.5 EEPROM and Flash Runtime Access

EEPROM and flash memory macros are defined for convenience and are available for chips that have EEPROM or flash memory on-board. The predefined EEPROM and flash memory macros can be used in the following manner:

To write a byte-size value to an address in EEPROM memory:

```
EEPROM_WRITE(address,value);
```

To read a byte of data from an address in EEPROM memory, and store it in a variable:

```
variable=EEPROM_READ(address);
```

For convenience, **`EEPROM_SIZE`** predefines the total size of data EEPROM available on chip.

To write a byte-size value to an address in flash memory:

```
FLASH_WRITE(address,value);
```

To read a byte of data from an address in flash memory, and store in in a variable:

```
variable=FLASH_READ(address);
```

5.2.6 Bit Instructions

Wherever possible, PICC will attempt to use the PIC bit instructions. For example, when using a bitwise operator and a mask to alter a bit within an integral type, the compiler will check the mask value to determine if a bit instruction can achieve the same functionality.

```
int foo;
foo |= 0x40;
```

will produce the instruction

```
bsf _foo,6
```

To set or clear individual bits within integral types, the following macros could be used.

```
#define bitset(var,bitno) ((var) |= 1 << (bitno))
#define bitclr(var,bitno) ((var) &= ~(1 << (bitno)))
```

To perform the same operation as above, the bitset macro could be employed as follows.

```
bitset(foo,6);
```

5.2.7 Baseline PIC special instructions

The PIC baseline devices have some registers which are not in the normal SFR area and cannot be accessed using an ordinary move instruction. The PICC compiler can be instructed to automatically use the special instructions intended for such cases when pre-defined symbols are accessed.

The definition of the special symbols make use of the **control** keyword. This keywords informs the compiler that the registers are outside of the normal address space and that a different access method is required.

5.2.8 The OPTION instruction

Some baseline PIC devices use an **option** instruction to load the OPTION register. The appropriate header files contain a special definition for a C object called **OPTION** and macros for the bit symbols which are stored in this register. PICC will automatically use the **option** instruction when an appropriate processor is selected and the **OPTION** object is accessed.

For example, to set the prescaler assignment bit so that prescaler is assigned to the watch dog timer, the following code can be used after including `pic.h`.

```
OPTION = PSA;
```

This will load the appropriate value into the W register and then call the **option** instruction.

5.2.9 The TRIS instructions

Some PIC devices use a **tris** instruction to load the TRIS register. The appropriate header files contain a special definition for a C object called **TRIS**. PICC will automatically use the **tris** instruction when an appropriate processor is selected and the **TRIS** object is accessed.

For example, to make all the bits on the output port high impedance, the following code can be used after including `pic.h`.

```
TRIS = 0xFF;
```

This will load the appropriate value into the W register and then call the **tris** instruction.

Those PIC devices which have more than one output port may have definitions for objects: TRISA, TRISB and TRISC, depending on the exact number of ports available. This objects are used in the same manneras described above.

5.2.10 Calibration Space

The Microchip-modified IEEE754 32-bit floating point format parameters in the calibration space in the PIC14000 processor may be accessed using the **`get_cal_data()`** function as described on page 310.

The byte parameters may be accessed directly using the identifiers defined in the header file.

5.2.11 Oscillator calibration constants

Some PIC devices come with an oscillator calibration constant which is pre-programmed into the device's program memory. This constant can be read and written to the OSCCAL register to calibrate the internal RC oscillator. On some baseline PIC devices the calibration constant is stored as a **`retlw`** instruction at the top of program memory, e.g. the 12C50X and 16C505 parts. On reset the program counter is made to point to this instruction and it is executed first before the program counter wraps around to 0x0000 which is the "effective" reset vector for the device. The `PICC` compiler's default startup routine will automatically include code to load the OSCCAL register with the value contained in the W register after reset on such devices. No other code is required by the programmer.

For 12C67X chips the oscillator constant is also stored at the top of program memory, but is not automatically loaded after reset. It can be read at any time during program execution using the macro **`_READ_OSCCAL_DATA()`**. To be able to use this macro, make sure that `<pic.h>` is included into the relevant modules of your program. This macro returns the calibration constant which should then stored into the OSCCAL register, as follows:

```
OSCCAL = _READ_OSCCAL_DATA();
```

The location which stores the calibration constant is never code protected and will be lost if you reprogram the device. Thus, if you are using a windowed device, the calibration constant must be saved from the last ROM location before it is erased. The constant must then be reprogrammed at the same location along with the new program and data.

If you are using an in-circuit emulator (ICE), the location used by the calibration **`retlw`** instruction may not be programmed and would be executed as some other instruction. Calling the **`_READ_OSCCAL_DATA()`** macro will not work and will almost certainly not return correctly. If you wish to test code that includes this macro on an ICE, you will have to program a **`retlw`** instruction at the appropriate location in program memory. Remember to remove this instruction when programming the actual part so you do not destroy the calibration value.

5.3 Files

5.3.1 Source Files

The extension used with source files is important as it is used by the compiler drivers to determine their content. Source files containing C code should have the extension `.c`, assembler files should have extensions of `.as`, relocatable object files require the `.obj` extension, and library files should be named with a `.lib` extension. See the tutorial Section See 2.1.2 on page 18 for more information on how these input files are processed by the compiler.

5.3.2 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.

The default behaviour of the `PICC` command is to produce *Bytecraft* COD and *Intel* HEX output. If no output filename or type is specified, `PICC` will produce a *Bytecraft* COD and *Intel* HEX file with the same base name as the first source or object file specified on the command line. Table 5 - 1 on page 134 shows the output format options available with `PICC`. The *File Type* column lists the filename extension which will be used for the output file.

Table 5 - 1 Output File Formats

Format Name	Description	PICC Option	File Type
Motorola HEX	S1/S9 type hex file	-MOT	`.hex`
Intel HEX	*Intel* style hex records (default)	-INTEL	`.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`
Bytecraft .COD	*Bytecraft* code format (default)	n/a (default)	`.cod`
Library	HI-TECH library file	n/a	`.lib`

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 filename with a `.bin` type, for example `-Otest.bin`, `PICC` 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`.

5.3.3 Symbol Files

The PICC -G option tells the compiler to produce a symbol file which can be used by debuggers and simulators to perform symbolic and source-level debugging. This option produces symbol files which contain both assembler- and C-level information. If no symbol filename is specified, by default a file called *file*.sym will be produced, where *file* is the basename of the first source file specified on the command line. For example, to produce a symbol file called test.sym which includes C source-level information:

```
PICC -16C84 -Gtest.sym test.c init.c
```

This option will also generate a different symbol file for each module compiled. These files do not contain absolute address and have the type .sdb. The base name will be the same as the base name of the module being compiled. Thus the above command line would also generate symbols files with the names test.sdb and init.sdb.

5.3.4 Standard Libraries

PICC includes a number of standard libraries, each with the range of functions described in the Library Functions chapter.

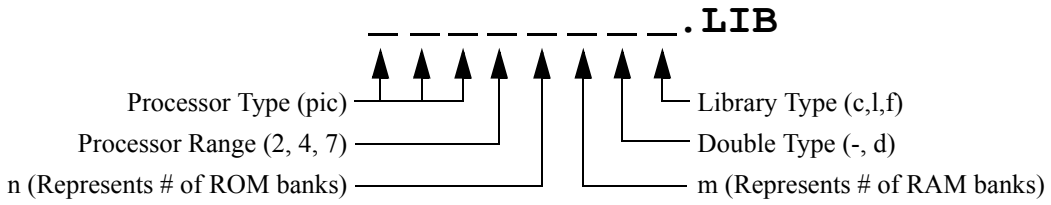
Figure 5 - 1 on page 136 illustrates the naming convention used for the standard libraries. The meaning of each field is described here, where:

- *Processor Type* is always pic.
- *Processor Range* is 2 for the Baseline range, 4 for the Midrange and 7 for the High-End range of PIC microprocessors.
- # *of ROM Banks* is 2^n.
- # *of RAM Banks* is 2^m. For Midrange processors that have common memory, a letter is used rather than a number. So for example where '0', '1' or '2' might be used for processors with no common memory, 'a', 'b', 'c' would be used for processors that do have common memory.
- *Double Type* is - for 24-bit doubles, and *d* for 32-bit doubles.
- *Library Type* is c for standard library, l for the library which contains only printf-related functions with additional support for longs, and f for the library which contains only printf-related functions with additional support for longs and floats.

5.3.4.1 Limitations of Printf

The **printf()** function is provided but some features have been removed. For more details on this function, see the documentation on page 330.

Figure 5 - 1 PIC Standard Library Naming Convention



5.3.5 Run-time startup Modules

A C program requires certain objects to be initialised and the processor to be in a particular state before it can begin execution. It is the job of the *run-time startup* code to ready the program for execution. Since this is code that executed before the C program, it is necessarily written in assembler code. The run-time startup code is executed almost immediately after reset. In fact it is called by a special *powerup* routine, described below, that is directly located at the reset vector address. For the PIC processors, the principle job of the run-time startup code is to clear unitialised variables and assign values to those variables that have been initialised.

The run-time startup code will clear, or assign the value zero, any variables which are unitialised at their definition and which are non-auto. This amounts to those objects which have been placed in the `rbss_n` or `rbit_n` psects (where *n* is a digit representing the bank number). Since these psects are defined as a contiguous block of memory, the run-time startup code calls a routine to clear a block of memory for each psect. In the following example, all but the object **`loc`** will be cleared by the startup code since it is an auto object. The initial value of **`loc`** is unknown.

```
int             i;
bank1 int       b1i;
bit             b;
void main(void)
{
        static int      sloc;
        int             loc;
...
```

The code which clears these psects is only included if it is necessary. The modules which contain the clear routines can be found in the SOURCES directory of you distribution. The file `clr.as` contains the code to clear a block of memory. This is called, if required, by code in the files `clrbankn.as` and `clrbitn.as` (where *n* is a digit representing a bank number) which initiate the clearing of each psect. Each bank must be cleared separately by separate routines.

The other function of the run-time startup code is to initialise those variables assigned a value at their definition. This amounts to a block copy of the initial values from ROM to the RAM areas designated for those objects. Code to perform the copy is only included if required. In this example:

```
int             i  = 7;
bank1 int       b1i = 6;
bit             b;
void main(void)
{
        static int      sloc = 5;
        int             loc  = 7;
...
```

The objects **i**, **b1i** and **sloc** will be initialsed by the run-time startup code. Note that you cannot initalise **bit** objects and that initialised automatic variables are assigned their starting value by code placed within the function in which they are defined.

Any objects defined in assembler code, i.e. have memory reserved using any of the **DS**, **DB** or **DW** assembler directives, will also be cleared or initialised at startup providing that the directives are placed within the compiler-generated psect used for C variables, such as rbss_0, rdata_1 or rbit_0 etc.

Some baseline PIC devices may require oscillator calibration. If required, this is also perfomed by the run-time startup code.

The run-time startup code jumps to the function **main()**, which is referred to as **_main** by the run-time startup code. Note the *underscore* "_" prepended to the function name. A jump rather than a call is executed to save one level of stack. The function **main()** is, by definition of the C language, the "main program".

The source code used to generate the run-time startup module is called picrt66x.as which is in the SOURCES directory of your distribution. In addition to this will be the routines, mentioned above, to copy data or clear memory as required. These routines are not called by name, but are linked in, if required, to a position in picrt66x.as indicated by a comment.

The run-time startup code is provided by a standard module found in the LIB directory.

5.3.6 The *powerup* Routine

Some hardware configurations require special initialisation, often within the first few cycles of execution after reset. Rather than having to modify the run-time startup 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. A “dummy” powerup routine is included in the file `powerup.as`. The file can be copied, modified and included into your project. It will replace the default powerup routine.

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 shows the default powerup routine which are in the standard library:

```
        GLOBAL powerup,start
        PSECT  powerup,class=CODE,delta=2


powerup

        ljmp   start
```

The powerup routine is generally intended to be relatively small, however, since it is linked before the interrupt vectors, it may interfere with them if it becomes to large. To avoid conflicting with the interrupt vectors, the powerup routine can be made to jump to a separate function, which will be linked at a different location, and then jumps to **`start`**. The following gives an example of this:

```
            GLOBAL   powerup,start,big_powerup
            PSECT    powerup,class=CODE,delta=2

powerup

            ljmp     big_powerup

            PSECT    big_powerup,class=CODE,delta=2

big_powerup

            ...powerup code...

            ljmp start
```

5.4 Supported Data Types and Variables

The `PICC` 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.

Table 5 - 2 shows the data types and their corresponding size and arithmetic type.

Table 5 - 2 Data Types

Type	Size (in bits)	Arithmetic Type
`bit`	1	boolean
`char`	8	signed or unsigned integer[a]
`unsigned char`	8	unsigned integer
`short`	16	signed integer
`unsigned short`	16	unsigned integer
`int`	16	signed integer
`unsigned int`	16	unsigned integer
`long`	32	signed integer
`unsigned long`	32	unsigned integer
`float`	24	real
`double`	24 or 32[b]	real

a.A char is unsigned by default, and signed if the `PICC -SIGNED_CHAR` option is used.
b.A double defaults to 24-bit, but becomes 32-bit with the `PICC -D32` option.

5.4.1 Radix Specifiers and Constants

The format of integral constants specifies their radix. `PICC` supports the ANSI standard radix specifiers as well as one which enables binary constants to specified in C code. The format used to specify the radices are given in Table 5 - 3 on page 139. The letters used to specify binary or hexadecimal radices are case insensitive, as are the letters used to specify the hexadecimal digits.

Table 5 - 3 Radix Formats

Radix	Format	Example
binary	0b*number* or 0B*number*	0b10011010
octal	0*number*	0763
decimal	*number*	129
hexadecimal	0x*number* or 0X*number*	0x2F

Any integral constant will have a type which is the smallest type that can hold the value without overflow. The suffix **l** or **L** may be used with the constant to indicate that it must be assigned either a **signed long** or **unsigned long** type, and the suffix **u** or **U** may be used with the constant to indicate that it must be assigned an unsigned type, and both **l** or **L** and **u** or **U** may be used to indicate **unsigned long int** type.

Floating-point constants have **double** type unless suffixed by **f** or **F**, in which case it is a **float** constant. The suffixes **l** or **L** specify a **long double** type which is considered an identical type to **double** by PICC.

Character constants are enclosed by single quote characters " **'** ", for example **'a'**. A character constant has char type. Multi-byte character constants are not supported.

String constants or string literals are enclosed by double quote characters " **"** ", for example **"hello world"**. The type of string constants is **const char *** and the strings are stored in ROM. Assigning a string constant to a non-**const** char pointer will generate a warning from the compiler. For example:

```
char * cp                 = "one";          // "one" in ROM,
produces warning
const char * ccp          = "two";          // "two" in ROM
char ca  []               = "two";          // "two" different
to the above
```

A non-**const** array initialised with a string, for example the last statement in the above example, produces an array in RAM which is initialised at startup time with the string **"two"** (copied from ROM), whereas a constant string used in other contexts represents an unnamed **const**-qualified array, accessed directly in ROM.

PICC will use the same storage location and label for strings that have identical character sequences, except where the strings are used to initialise an array residing in RAM as indicated in the last statement in the above example.

Two adjacent string constants (i.e. two strings separated *only* by white space) are concatenated by the compiler. Thus:

```
const char * cp = "hello " "world";
```

assigned the pointer with the string **"hello world"**.

5.4.2 Bit Data Types and Variables

PICC supports **bit** integral types which can hold the values 0 or 1. Single **bit** variables may be declared using the keyword **bit**. **Bit** objects declared within a function, for example:

```
static bit init_flag;
```

will be allocated in the bit-addressable psect `rbit`, and will be visible only in that module or function. When the following declaration is used outside any function:

```
bit init_flag;
```

`init_flag` will be globally visible, but located within the same psect.

`Bit` variables cannot be **`auto`** or parameters to a function. A function may return a **`bit`** object by using the **`bit`** keyword in the function's prototype in the usual way.

`Bit` variables behave in most respects like normal **`unsigned char`** variables, but they may only contain the values 0 and 1, and therefore provide a convenient and efficient method of storing boolean flags without consuming large amounts of internal RAM. It is not possible to declared pointers to **`bit`** variables or statically initialise **`bit`** variables. Operations on **`bit`** objects are performed using the single bit instructions wherever possible, thus the generated code to access **`bit`** objects is very efficient.

Note that when assigning a larger integral type to a **`bit`** variable, only the least-significant bit is used. For example, if the **`bit`** variable **`bitvar`** was assigned as in the following:

```
int     data = 0x54;
bit     bitvar;

bitvar = data;
```


it will be cleared by the code since the least significant bit of **`data`** is zero.

If you want to set a **`bit`** variable to be 0 or 1 depending on whether the larger integral type is zero (false) or non-zero (true), use the form:

```
bitvar = data != 0;
```

The psects in which **`bit`** objects are allocated storage are declared using the **`bit`** psect directive flag. Eight bit objects will take up one byte of storage space which is indicated by the bit psects' *scale* value of 8 in the map file. The length given in the map file for bit psects is in units of bits, not bytes. All addresses specified for bit objects are also bit addresses.

The `bit` psects are cleared on startup, but are not initialised. To create a **`bit`** object which has a non-zero initial value, explicitly initialise it at the beginning of your code.

If the `PICC` flag `-STRICT` is used, the **`bit`** keyword becomes unavailable.

5.4.2.1 Using Bit-Addressable Registers

The **`bit`** variable facility may be combined with absolute variable declarations (see page 153) to access bits at specific addresses. Absolute **`bit`** objects are numbered from 0 (the least significant bit of the first

byte) up. Therefore, bit number 3 (the fourth bit in the byte since numbering starts with 0) in byte number 5 is actually absolute bit number 43 (that is 8bits/byte * 5 bytes + 3 bits).

For example, to access the *power down detection flag* bit in the RCON register, declare **RCON** to be a C object at absolute address 03h, then declare a **bit** variable at absolute bit address 27:

```
static unsigned char RCON @ 0xFD0;

static near bit PD @ (unsigned)&RCON*8+2;
```

Note that all standard registers and bits within these registers are defined in the header files provided. The only header file you need to include to have access to the PIC registers is <pic.h> - at compile time this will include the appropriate header for the selected chip.

5.4.3 8-Bit Integer Data Types and Variables

PICC supports both **signed char** and **unsigned char** 8-bit integral types. If the **signed** or **unsigned** keyword is absent, the default type is **unsigned char** unless the PICC -SIGNED_CHAR option is used, in which case it is **signed char**. The **signed char** type is an 8-bit two's complement signed integer type, representing integral values from -128 to +127 inclusive. The **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** values in C expressions. With PICC 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 default **unsigned char** type is the most efficient data type on the PIC and maps directly onto the 8-bit bytes which are most efficiently manipulated by PIC instructions. It is suggested that **char** types be used wherever possible so as to maximize performance and minimize code size.

Variables may be declared using the **signed char** and **unsigned char** keywords, respectively, to hold values of these types. Where only **char** is used in the declaration, the type will be **unsigned char** unless the option, mentioned above, to specify **signed char** as default is used.

5.4.4 16-Bit Integer Data Types

PICC 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. All 16-bit integer values are represented in *little endian* format with the least significant byte at the lower

address. Both **int** and **short** types are 16 bits wide as this is the smallest integer size allowed by the ANSI standard for C. The sizes of the integer types 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** types wherever possible.

Variables may be declared using the **signed int**, **unsigned int**, **signed short int** and **unsigned short int** keyword sequences, respectively, to hold values of these types. Where only **int** is used in the declaration, the type will be **signed int**. When specifying a **short int** type, the keyword **int** may be omitted. Thus a variable declared as **short** will contain a **signed short int** and a variable declared as **unsigned short** will contain an **unsigned short int**.

5.4.5 32-Bit Integer Data Types and Variables

PICC 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. All 32-bit integer values are represented in *little endian* format with the least significant word and least significant byte at the lowest address. **Long** and **unsigned long** occupy 32 bits as this is the smallest long integer size allowed by the ANSI standard for C.

Variables may be declared using the **signed long int** and **unsigned long int** keyword sequences, respectively, to hold values of these types. Where only **long int** is used in the declaration, the type will be **signed long**. When specifying this type, the keyword **int** may be omitted. Thus a variable declared as **long** will contain a **signed long int** and a variable declared as **unsigned long** will contain an **unsigned long int**.

5.4.6 Floating Point Types and Variables

Floating point is implemented using the IEEE 754 32-bit format and a modified IEEE 754 (truncated) 24-bit format.

The truncated 24-bit format is used for all **float** values. For **double** values, the truncated 24-bit format is the default, but may be explicitly invoked with the `PICC -D24` option. The 32-bit format is used for doubles by using the `PICC -D32` option. The **long double** type is identical to the **double** type.

Both of these formats are described in Table 5 - 4, where:

- *sign* is the sign bit
- *exponent* is an 8-bit exponent which is stored as *excess 127* (i.e. an exponent of 0 is stored as 127)
- *mantissa* is the mantissa, which is to the right of the radix point. There is an implied bit to the left of the radix point which is always 1 except for a zero value, where the implied bit is zero. A zero value is indicated by a zero exponent.

The value of this number is $(-1)^{sign} \times 2^{(exponent-127)} \times 1.mantissa$.

Table 5 - 4 Floating Point Formats

Format	Sign	biased exponent	mantissa
IEEE 754 32-bit	x	xxxx xxxx	xxx xxxx xxxx xxxx xxxx xxxx
Modified IEEE 754 24-bit	x	xxxx xxxx	xxx xxxx xxxx xxxx

Here are some examples of the IEEE 754 32-bit and modified IEEE 754 24-bit formats:

Table 5 - 5 IEEE 754 32-bit and 24-bit Examples

Format	Number	biased exponent	1.mantissa	decimal
IEEE 754 32-bit	7DA6B69Bh	11111011b (251)	1.01001101011011010011011b (1.302447676659)	2.77000e+37
Modified IEEE 754 24-bit	42123Ah	10000100b (132)	1.001001000111010b (1.142395019531)	36.557

5

Note that the most significant bit of the mantissa column in Table 5 - 4 on page 144 (that is the bit to the left of the radix point) is the implied bit, which is assumed to be 1 unless the exponent is zero (in which case the float is zero).

The 32-bit example in Table 5 - 5 on page 144 can be calculated manually as follows.

The sign bit is zero; the biased exponent is 251, so the exponent is 251-127=124. Take the binary number to the right of the decimal point in the mantissa. Convert this to decimal and divide it by 2^{23} where 23 is the number of bits taken up by the mantissa, to give 0.302447676659. Add one to this fraction. The floating-point number is then given by:

$(-1)^0 \times 2^{(124)} \times 1.302447676659 = 1 \times 2.126764793256e+37 \times 1.302447676659 \approx 2.77000e+37$

Variables may be declared using the **`float`** and **`double`** keywords, respectively, to hold values of these types. Floating point types are always signed and the **`unsigned`** keyword is illegal when specifying a floating point type. Types declared as **`long double`** will use the same format as types declared as **`double`**.

5.4.7 Structures and Unions

PICC supports **`struct`** and **`union`** types of any size from one byte upwards. Structures and unions only differ in the memory offset applied for each member. The members of structures and unions may not be objects of type **`bit`**, but bit fields are fully supported.

Structures and unions may be passed freely as function arguments and return values. Pointers to structures and unions are fully supported.

5.4.7.1 Bit Fields in Structures

PICC fully supports *bit fields* in structures.

Bit fields are always allocated within 8-bit words. The first bit defined will be the least significant bit of the word in which it will be stored. When a bit field is declared, it is allocated within the current 8-bit unit if it will fit, otherwise a new 8-bit byte is allocated within the structure. Bit fields can never cross the boundary between 8-bit allocation units. For example, the declaration:

```
struct {
        unsigned        lo : 1;
        unsigned        dummy : 6;
        unsigned        hi : 1;
} foo;
```

will produce a structure occupying 1 byte. If **foo** was ultimately linked at address 10H, the field **lo** will be bit 0 of address 10H, **hi** will be bit 7 of address 10H. 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 10h.

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        lo : 1;
        unsigned           : 6;
        unsigned        hi : 1;
} foo;
```

If a bit field is declared in a structure that is assigned an absolute address, no storage will be allocated for the structure. Absolute structures would be used when mapping a structure over a register to allow a portable method of accessing individual bits within the register.

A structure with bitfields may be initialised by supplying a comma-separated list of initial values for each field. For example:

```
struct {
        unsigned        lo  : 1;
        unsigned        mid : 6;
        unsigned        hi  : 1;
} foo = {1, 8, 0};
```

5.4.7.2 Structure and Union Qualifiers

PICC supports the use of type qualifiers on structures. When a qualifier is applied to a structure, all of its members will inherit this qualification. In the following example the structure is qualified **const**.

```
const struct {
        int number;
        int *ptr;
} record = { 0x55, &i};
```

In this case, the structure will be placed into ROM and each member will, obviously, be read-only. Remember that all members must be initialised if a structure is **const**.

If the members of the structure were individually qualified **const** but the structure was not, then the structure would be positioned into RAM, but each member would be read-only. Compare the following structure with the above.

```
struct {
        const int number;
        int * const ptr;
} record = { 0x55, &i};
```


5.4.8 Standard Type Qualifiers

Type qualifiers provide information regarding how an object may be used, in addition to its type which defines it storage size and format. PICC supports both ANSI qualifiers and additional special qualifiers which are useful for embedded applications and which take advantage of PIC architecture.

5.4.8.1 Const and Volatile Type Qualifiers

PICC 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 psects in ROM. Obviously, a **const** object must be initialised when it is declared as it cannot be assigned a value at any point in the code following. 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 Input/Output ports and any variables which may be modified by interrupt routines should be declared **volatile**, for example:

```
volatile static near unsigned char  PORTA @ 0xF80;
```

Volatile objects may be accessed in a different way to non-**volatile** objects. For example, when assigning a non-**volatile** object the value 1, the object will be cleared and then incremented, but the same operation performed on a **volatile** object will load the W register with 1 and then store this to the appropriate address.

5.4.9 Special Type Qualifiers

PICC supports special type qualifiers, **persistent**, **bank1**, **bank2** and **bank3** to allow the user to control placement of **static** and **extern** class variables into particular address spaces. If the PICC option, -STRICT is used, these type qualifiers are changed to **__persistent**, **__bank1**, **__bank2** and **__bank3**. These type qualifiers may also be applied to pointers. These type qualifiers may not be used on variables of class **auto**; if used on variables local to a function they must be combined with the **static** keyword. For example, you may not write:

```
void test(void)
{
        /* WRONG! */
        persistent int intvar;

        .. other code ..
}
```

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

```
static persistent int intvar;
```

5.4.9.1 *Persistent* Type Qualifier

By default, any C variables that are not explicitly initialised 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 (for example, the nvram or nvram_1 psect).

There are some library routines provided to check and initialise **persistent** data - see page 328 for more information, and for an example of using **persistent** data.

5.4.9.2 *Bank1*, *Bank2* and *Bank3* Type Qualifiers

The **bank1**, **bank2** and **bank3** type qualifiers are used to place static variables in RAM Bank 1, RAM Bank 2 and RAM Bank 3 respectively. In the baseline microprocessors, pointers are unaffected by these type qualifiers.

Note that there is no bank0 qualifier. Objects default to being in bank 0 RAM if no other bank qualifier is used. All **auto** objects are positioned into bank 0 RAM, along with function parameters.

Here is an **unsigned char** in bank 3 RAM:

```
static bank3 unsigned char fred;
```

Here is a pointer to an **unsigned char** in bank 3 RAM:

```
bank3 unsigned char * ptrfred;
```

Here is another pointer to an **unsigned char** in bank 3 RAM, except this time the pointer resides in bank 2 RAM:

```
static bank 3 unsigned char * bank2 ptrfred;
```

5.4.10 Pointers Types

5

The format and use of pointers depend upon the range of processor.

5.4.10.1 Baseline Pointers

All pointers in the Baseline range are 8-bits. This describes the pointer types used.

- *RAM Pointers*
  RAM pointers point into RAM using the FSR index register.
- *Const Pointers*
  Const pointers point into ROM via a RETLW table.
- *Function Pointers*
  These pointers reference functions. A function is jumped to rather than called. A jump table is used to return control to the calling function.

5.4.10.2 Midrange Pointers

All pointers for the Midrange are the same as for the Baseline processors with the following exceptions:

- *RAM Pointers*
  Because an 8-bit pointer can only access 256 bytes, RAM pointers can only access objects in Bank 0 and Bank 1.
- *Bank2 Pointers* and *Bank3 Pointers*

These pointers are RAM pointers which are used to access Bank 2 and Bank 3 of RAM respectively.

Note that at present it is not possible to have a Midrange RAM pointer which can access objects in three or more banks, or which can access objects in bank pairs other than those mentioned above.

- *Const Pointers*
  `Const` pointers for the Midrange processors are 16-bit wide. They can be used to access either ROM or RAM.

  If the upper bit of the **`const`** pointer is non-zero, it is a pointer into RAM in nay bank. A **`const`** pointer may be used to read from RAM locations, but writing to such locations is not permitted.

  If the upper bit is zero, it is a pointer able to access the entire ROM space.

  The ability of this pointer to access both ROM and RAM is very useful in string-related functions where a pointer passed to the function may point to a string in ROM or RAM.

- *Function Pointers*
  These pointers reference functions. A function is called using the address assigned to the pointer.

5.4.10.3 High-End Pointers

- RAM Pointers
  RAM banks in the High-End processors are 0xFF bytes long so an 8-bit RAM pointer can only access objects in one bank. Thus a Bank 0 RAM pointers can only access Bank 0; a Bank 1RAM pointer can only access Bank 1, etc..

- Const Pointers
  Const pointers for the High-End processors are 16-bit wide. They can be used to access either ROM or RAM.

  If the upper bit of the **`const`** pointer is non-zero, it is a pointer into RAM in any bank. A **`const`** pointer may be used to read from RAM locations, but writing to such locations is not permitted. If the upper bit is zero, it is a pointer able to access the lower 32 kbytes (16 kwords) of ROM.

  The ability of this pointer to access both ROM and RAM is very useful in string-related functions where a pointer passed to the function may point to a string in ROM or RAM.

  `Const` pointer may be used to hold the address of objects in external memory.

- Far pointers
  Far pointers are 16-bits wide and can be used to access objects in any available bank of RAM. These pointers are similar to **const** pointers; they differ in that they may be used to indirectly read from, and write to, RAM locations. **Far** pointers can also access ROM locations as per const pointers. The **far** qualifier is used to specify the larger pointer size.

- Function Pointers
  These pointers reference functions. A function is called using the address assigned to the pointer.

5.4.10.4 Combining Type Qualifiers and Pointers

The **const**, **volatile** and **persistent** modifiers 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 modifier is to the left of the ***** in the pointer declaration, it applies to the object which the pointer addresses. If the modifier is to the right of the *****, it applies to the pointer variable itself. Using the **volatile** keyword to illustrate, the declaration:

```
volatile char *  nptr;
```

declares a pointer to a **volatile** character. The **volatile** modifier 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** keyword 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.

5.4.10.5 *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. In the baseline series, **const** pointers always access program ROM because const declared objects are stored in ROM. In the midrange series, **const** pointers can access RAM as well as ROM.

5.5 Storage Class and Object Placement

Objects stored in ROM include string literals, or constants, and any objects qualified using the **const** keyword. Placement of these objects varies on the device for which the code is compiled.

5.5.1 Baseline PICs

Baseline devices store ROM-based objects as **retlw** instructions since there is no instruction to directly read data from ROM. Psects called strings are to hold ROM-based data. These psects are positioned explicitly as indicated in the linker options.

Since these instruction are executed via a call, there may be restrictions on the amount of data which can be placed in ROM. For example, the PIC12C509 chip can only make calls to the first half of each ROM page. The linker cannot detect that such a restriction has been violated so check the map file for the total sizes of the strings psects.

5.5.2 Midrange PICs

Midrange PICs also store ROM-based objects as **retlw** instructions. Such objects are contained in psects called strings or const*n*, where *n* is a number. **Const** compound objects (for example structures or arrays) whose total size is less than 256 bytes, or **const** objects of basic type (for example **int**s) are stored in the const*n* psects. Other objects are stored in the strings psect. The strings psects is positioned explicitly whereas the const*n* psects are placed using the **CONST** class.

const*n* psects, in addition to the **retlw** instructions, begin with a **addwf pc** instruction. (This is why you cannot have exactly 256 bytes in const*n* psects.) This instruction is used by routines which perform indirect accessing of the objects.

For those midrange processors that may read their own program memory, string literals are packed 2 characters per word if it is efficient to do so.

5.5.3 High-End PICs

High-End PIC devices store **const** objects in ROM as an actual value, storing two bytes per 16-bit word and using the table read instructions to retrieve them. All ROM-based objects are placed in the cstrings psect which is part of, and placed using, the **ROMDATA** class. This class can define one of two 32 kbyte (16 kword) ranges: 0h-3FFFh or 8000h-BFFFh.

When dereferencing the address of a **const** object the address will be divided by two (shifted right one bit) to convert it from a byte address to a word address that the table read instruction is expecting. The most significant bit of the resulting word address is obtained from the most significant bit of an object called **stringbase** and this is used to determine which of the two above address ranges is applicable. **Stringbase** is a global symbol defined in the cstrings psect and so the most significant bit of its address will be set or clear depending on where the cstrings psect was positioned by the linker. Thus 15 bits of a **const** address are used to determine the offset into the **const** storage range.

5.5.4 Local Variables

A *local variable* is one which only has scope within the block in which it was defined. That is, it may only be referenced within that block. C supports two classes of local variables in functions: **auto** variables which are normally allocated in the function's auto-variable block, and **static** variables which are always given a fixed memory location and have permanent duration.

5.5.4.1 Auto Variables

Auto (short for *automatic*) variables are the default type of local variable. Unless explicitly declared to be **static** a local variable will be made **auto**, however the **auto** keyword may be used if desired. **Auto** variables are allocated in the *auto-variable block* and referenced by indexing off the symbol that represents that block. 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. The exceptions are **const** and **volatile**.

All **auto** variables are allocated memory within one bank of RAM. At present, all functions share the same bank of memory for **auto** objects. The size of a functions auto-variable block may not exceed the size of one bank, which is 100H bytes.

The auto-variable blocks for a number of functions are overlapped by the linker if those functions are never called at the same time.

Auto objects are referenced with a symbol that consists of a *question mark,* "**?**", concatenated with **a_*function*** plus some offset, where ***function*** is the name of the function in which the object is defined. For example, if the **int** object **test** is the first object placed in **main()**'s auto-variable block it will be accessed using the addresses **?a_main** and **?a_main+1** since an **int** is two bytes long.

Auto variables may be accessed using the banked instructions of the PIC. When accessing **auto** objects with banked instructions, the compiler will ensure that the bank of the auto-variable block is selected using a **movlb** instruction, and then access the locations using the appropriate instructions. In essence this amounts to an 8 bit access within the selected bank.

5.5.4.2 Static Variables

Uninitialized **static** variables are allocated in one of the bss, rbss or bigbss psects. Objects qualified **near** appear in the rbss psect; objects larger than one bank in size or byte long objects are placed in the

bigbss psect and the remainder in the bss psect. They will occupy fixed memory locations which will not be overlapped by storage for other functions. **Static** variables are local in scope to the function in which they are declared, but may be accessed by other functions via pointers since they have permanent duration. **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 PIC.

Static variables which are initialised are only done so once during the program's execution. Thus, they may be preferable over initialised **auto** objects which are assigned a value every time the block in which the definition is placed is executed.

5.5.5 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 @ 0x06;
```

will declare a variable called **Portvar** located at 06h. 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  06h
```

Note also 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 "The #pragma psect Directive" on page 174.

Absolute variables have their address supplied by the code generator, not the linker, and hence no symbols are used which require fixup by the linker. This means that the name of the object will not be present in the map file, or any symbol information produced by the linker.

5.5.6 Objects in Program Space

Const objects are usually placed in program space. On the PIC devices, the program space is byte-wide, the compiler stores one character per byte location and values are read using the table read instructions. All **const**-qualified data objects and string literals are placed in the const psect. The const psect is placed at an address above the upper limit of RAM since RAM and **const** pointers use this address to determine if an access to ROM or RAM is required. See section 5.4.10 on page 148.

5.5.7 Strings In ROM and RAM

An anonymous constant string is always placed in ROM and can only be accessed via a **const** pointer. In the following example, the string **"Hello world"** is a constant string and is stored in ROM. It is therefore accessed via a **const** pointer:

```
#define HELLO "Hello world"
SendBuff(HELLO);
```

A non-**const** array initialised with a string, for example:

```
char fred[] = "Hello world";
```

produces an array in RAM which is initialised at startup time with the string **"Hello world"** (copied from ROM), whereas a constant string used in other contexts represents an unnamed array qualified **const**, accessed directly in ROM.

If you want to pass a constant string to a function argument, or assign it to a pointer, that pointer must be a **const char ***, for example:

```
void SendBuff(const char * ptr)
```

or similar. Now you can pass either a pointer into ROM or RAM (on the midrange chips only - **const** pointers always point into ROM for baseline chips) and it will correctly fetch the data from the appropriate place.

5.6 Functions

5.6.1 Function Argument Passing

The method used to pass function arguments depends on the size of the argument or arguments.

If there is only one argument, and it is one byte in size, it is passed in the W register.

If there is only one argument, and it is greater than one byte in size, it is passed in the argument area of the called function. If there are subsequent arguments, these arguments are also passed in the argument area of the called function. The argument area is referenced by an offset from the symbol **?_*function***, where ***function*** is the name of the function concerned.

If there is more than one argument, and the first argument is one byte in size, it is passed in the W register, with subsequent arguments being passed in the argument area of the called function.

In the case of a variable argument list, which is defined by the ellipsis symbol **...**, the calling function builds up the variable argument list and passes a pointer to the variable part of the argument list in **btemp**. **Btemp** is the label at the start of the `temp` psect (the psect used for temporary data).

Take, for example, the following ANSI-style function:

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

The function **test()** will receive the parameter **b** in its function argument block and **a** in the W register. A call:

```
test( 'a', 8);
```

would generate code similar to:

```
movlw   08h
movff   wreg,?_test
movlw   0h
movff   wreg,?_test+1
movlw   061h
call    (_test)
```

In this example, the parameter **b** is held in the memory locations **?_test** and **?_test+1**.

If you need to determine, for assembler code for example, the exact entry or exit code within a function or the code used to call a function, 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 PICC -S option, allowing you to examine the assembler code.

5.6.2 Function Return Values

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

5.6.2.1 8-Bit Return Values

Eight-bit values are returned from a function in the W register. For example, the function:

```
char return_8(void)
{
        return 0;
}
```

will exit with the following code:

```
movlw    0
return
```

5.6.2.2 16-Bit and 32-bit Return Values

16-bit and 32-bit values are returned in temporary memory locations, with the least significant word in the lowest memory location. For example, the function:

```
int return_16(void)
{
        return 0x1234;
}
```

will exit with the following code:

```
movlw   34h
movwf   btemp
movlw   12h
movwf   btemp+1
return
```

5.6.2.3 Structure Return Values

Composite return values (**`struct`** and **`union`**) of size 4 bytes or smaller are returned in memory as with 16-bit and 32-bit return values. For composite return values of greater than 4 bytes in size, the structure or union is copied into the `struct` psect. Data is copied using the library routine **`structcopy`** which uses FSR0 for the source address, FSR1 for the destination address and W for the structure size. For example:

```
struct fred
{
        int ace[4];
} ;

struct fred return_struct(void)
{
        struct fred wow;

        return wow;
}
```

will exit with the following code:

```
movlw   low(?a_func+0)
movwf   fsr0l
movlw   high(?a_func+0)
```

```
movwf   fsr0h
movlw   structret
movwf   fsr1l
clrf    fsr1h
movlw   24
global  structcopy
call    structcopy
```

5.6.3 Function Calling Convention

The baseline PIC devices have a two-level deep hardware stack which is used to store the return address of a subroutine call. Typically, PICC uses a call instruction to transfer control to a C function when it is called, however on baseline processors, the size of the stack severely limits the level of nested C function calls possible.

By default, function calls on baseline PICs are implemented using a method involving an assembly jump instruction and a lookup table, or jump table. A function is "called" by jumping directly to its address after storing the address of a jump table instruction which will be able to return control back to the calling function. The address is stored as an object local to the function being called. The lookup table is accessed after the function called has finished executing. This method allows functions to be nested without overflowing the stack, however it does come at the expense of memory and program speed.

To disable the lookup-table mode of operation, a function definition can be qualified as **fastcall**, so that calls to this function are performed using the usual call assembly instruction. Extreme care must be used when functions are declared as **fastcall**, since the each nested **fastcall** will use one word of available stack space. Check the call graph in the map file to ensure that the stack will not overflow.

The function prototype for a baseline **fastcall** function might look something like:

```
fastcall void my_function(int a);
```

The midrange and high end PIC devices have larger stacks and are thus allow a higher degree of function nesting. These devices do not use the lookup table method when calling functions.

The compiler assumes that bank zero will be selected after returning from any function call. The compiler inserts the appropriate instructions to ensure this is true if required. Any functions callable from C code that are written in assembler must also ensure that bank zero is selected before the return.

5.7 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.

The memory used is based upon page and bank information in the chipinfo file (which defaults to `picinfo.ini` in the LIB directory) The linker will automatically locate code and ROM data into all the available memory pages and ensure that a psect does not straddle a page boundary.

5.7.1 External Memory

The High-End PIC devices can use external ROM to hold program code. The address ranges available to the linker may be indicated by using the `-A`*`class`* command line argument to `HLINK` (or `-L-A`*`class`* option to `PICC`) or by using the **ROM addresses...** dialogue under `HPDPIC`.

If any of the external memory address are modified, all source files must be re-compiled with the new addresses. The external ROM addresses are used by the code generator to compile the code, so it is not sufficient to simply re-link code if these addresses change.Function Argument Passing

The method used to pass function arguments depends on the size of the argument or arguments.

If there is only one argument, and it is one byte in size, it is passed in the W register.

If there is only one argument, and it is greater than one byte in size, it is passed in the argument area of the called function. If there are subsequent arguments, these arguments are also passed in the argument area of the called function.

If there is more than one argument, and the first argument is one byte in size, it is passed in the auto variable area of the called function, with subsequent arguments being passed in the argument area of the called function.

In the case of a variable argument list, which is defined by the ellipsis symbol `...`, the calling function builds up the variable argument list and passes a pointer to the variable part of the argument list in **`btemp`**. **`btemp`** is the label at the start of the `temp` psect (i.e. the psect used for temporary data).

Take, for example, the following ANSI-style function:

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

The function **`test()`** will receive all arguments in its function argument block. A call:

```
test( 0x65af, 0x7288, 0x080c);
```

would generate code similar to:

```
movlw    0AFh
movwf    (((?_test))&7fh)
movlw    065h
```

```
movwf   (((?_test+1))&7fh)
movlw   088h
movwf   ((0+((?_test)+02h))&7fh)
movlw   072h
movwf   ((1+((?_test)+02h))&7fh)
movlw   0Ch
movwf   ((0+((?_test)+04h))&7fh)
movlw   08h
movwf   ((1+((?_test)+04h))&7fh)
lcall   (_test)
```

Parameters passed to a function are referred to by a label which consists of *question mark* **?** followed by an *underscore* **_** and the name of the function to which is added an offset. So, for example in the above code, the first parameter to the function **test**, the int value 0x65af, is held in the memory locations **?_test** and **?_test+1**.

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 `PICC -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.

5.8 Register Usage

In the midrange processors, the W register is used for register-based function argument passing and for function return values. This register should be preserved by any assembly language routines which are called.

5.9 Operators

PICC supports all the ANSI operators. The exact results of some of these are implementation defined. The following sections illustrate code produced by the compiler.

5.9.1 Integral Promotion

When there is more than one operand to an operator, they typically must be of exactly the same type. The compiler will automatically convert the operands, if necessary, so they have the same type. The conversion is to a “larger” type so there is no loss of information. Even if the operands have the same type, in some situations they are converted to a different type before the operation. This conversion is called *integral promotion*. PICC performs these integral promotions where required. If you are not aware that these changes of type have taken place, the results of some expressions are not what would normally be expected.

Integral promotion is the implicit conversion of enumerated types, **signed** or **unsigned** varieties of **char**, **short int** or bitfield types to either **signed int** or **unsigned int**. If the result of the conversion can be represented by an **signed int**, then that is the destination type, otherwise the conversion is to **unsigned int**.

Consider the following example.

```
unsigned char count, a=0, b=50;
if(a - b < 10)
  count++;
```

The **unsigned char** result of **a - b** is 206 (which is not less than 10), but both **a** and **b** are converted to **signed int** via integral promotion before the subtraction takes place. The result of the subtraction with these data types is -50 (which is less than 10) and hence the body of the **if()** statement is executed. If the result of the subtraction is to be an **unsigned** quantity, then apply a cast. For example:

```
if((unsigned int)(a - b) < 10)
  count++;
```

The comparison is then done using **unsigned int**, in this case, and the body of the **if()** would not be executed.

Another problem that frequently occurs is with the bitwise compliment operator, "**~**". This operator toggles each bit within a value. Consider the following code.

```
unsigned char count, c;
c = 0x55;
if( ~c == 0xAA)
  count++;
```

If **c** contains the value 55h, it often assumed that **~c** will produce AAh, however the result is FFAAh and so the comparison in the above example would fail. The compiler may be able to issue a mismatched comparison error to this effect in some circumstances. Again, a cast could be used to change this behaviour.

The consequence of integral promotion as illustrated above is that operations are not performed with **char**-type operands, but with **int**-type operands. However there are circumstances when the result of an operation is identical regardless of whether the operands are of type **char** or **int**. In these cases, PICC will not perform the integral promotion so as to increase the code efficiency. Consider the following example.

```
unsigned char a, b, c;
a = b + c;
```

Strictly speaking, this statement requires that the values of **b** and **c** should be promoted to **unsigned int**, the addition performed, the result of the addition cast to the type of **a**, and then the assignment can take place. Even if the result of the **unsigned int** addition of the promoted values of **b** and **c** was different to the result of the **unsigned char** addition of these values without promotion, after the **unsigned int** result was converted back to **unsigned char**, the final result would be the same. An 8-bit addition is more efficient than an a 16-bit addition and so the compiler will encode the former.

If, in the above example, the type of **a** was **unsigned int,** then integral promotion would have to be performed to comply with the ANSI standard.

5.9.2 Shifts applied to integral types

The ANSI standard states that the result of right shifting (**>>** operator) signed integral types is implementation defined when the operand is negative. Typically, the possible actions that can be taken are that when an object is shifted right by one bit, the bit value shifted into the most significant bit of the result can either be zero, or a copy of the most significant bit before the shift took place. The latter case amounts to a sign extension of the number.

PICC performs a sign extension of any **signed** integral type (for example **signed char**, **signed int** or **signed long**). Thus an object with the **signed int** value 0124h shifted right one bit will yield the value 0092h and the value 8024h shifted right one bit will yield the value C012h.

Right shifts of **unsigned** integral values always clear the most significant bit of the result.

Left shifts (**<<** operator), **signed** or **unsigned**, always clear the least significant bit of the result.

5.9.3 Division and modulus with integral types

The sign of the result of division with integers when either operand is negative is implementation specific. Table 5 - 6 shows the expected sign of the result of the division of operand 1 with operand 2 when compiled with PICC.

Table 5 - 6 Integral division

Operand 1	Operand 2	Quotient	Remainder
+	+	+	+
-	+	-	-
+	-	-	+
-	-	+	-

In the case where the second operand is zero (division by zero), the result will always be zero.

5.10 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 PICC or HPDPIC 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 (this is not recommended), or write your own assembly language subroutines, you should read this section carefully.

A psect can be created in assembler code by using the **PSECT** assembler directive, see. In C, user-defined psects can be created by using the **#pragma psect** preprocessor directive.

5.10.1 Compiler-generated Psects

The code generator places code and data into psects with standard names which are subsequent positioned by the default linker options. These psects are described below.

The compiler-generated psects which are placed in ROM are:

powerup — Which contains executable code for the standard or user-supplied power-up routine.

idata_*n* — These psects (where *n* is the bank number) contain the ROM image of any initialised variables. These psects are copied into the rdata_*n* psects at startup.

text*n* — These psects (where *n* is a number) contain all executable code for the Midrange and High-end processors. They also contains any executable code after the first **goto** instruction which can never be skipped for the Baseline processors.

ctext*n* — These psects (where *n* is a number) are used only in the Baseline processors. They contain executable code from the entry point of each **fastcall** function until the first **goto** instruction which can never be skipped. Further executable code is placed in the text*n* psects.

text — Is a global psect used for executable code for some library functions.

const*n* — These psects (where *n* is a number) hold objects that are declared **const** and which are not modifiable.

strings — The strings psect is used for some **const** objects. **Const** objects whose size exceeds 256 bytes, for example **const** arrays, are positioned in this psect. It also includes 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.

`cstrings` High-End processors use the `cstrings` psect to store **`const`** objects and string literals. ROM on these devices is 16 bits wide so two characters can be stored in the one ROM word.

`stringtable` The `stringtable` psect contains the string table which is used to access objects in the `strings` psect. This psect will only be generated if there is a `strings` or baseline `jmp_tab` psect.

`jmp_tab` Only for the Baseline processors, this is another strings psect used to store jump addresses and function return values.

`config` Used to store the configuration word.

`idloc` Used to store the ID location words.

`intentry` Contains the entry code for the interrupt service routine. This code saves the necessary registers and parts of the `temp` psect.

`intcode` Is the psect which contains the executable code for the interrupt service routine.

`intret` Is the psect which contains the executable code responsible for restoring saved registers and objects after an interrupt routine has completed executing.

`init` Used by initialisation code which, for example, clears RAM.

`end_init` Used by initialisation code which, for example, clears RAM.

`float_text` Used by some library routines, and in particular by arithmetic routines.It is possible that this psect will have a non-zero length even if no floating point operations are included in a program.

`clrtext` Used by some startup routines for clearing the `rbss_`*`n`* psects.

➡The compiler generated psects which are placed in RAM are:

`rbss_`*`n`* These psects (where *n* is the bank number) contain any uninitialized variables.

`rdata_`*`n`* These psects (where *n* is the bank number) contain any initialised variables.

`nvram` This psect is used to store **`persistent`** variables. It is not cleared or otherwise modified at startup.

`rbit_`*`n`* These psects (where *n* is the bank number) are used to store all **`bit`** variables except those declared at absolute locations. The declaration:

```
static bit flag;
```

will allocate **`flag`** as a single bit in the `rbit_`*`n`* psect.

`struct`	Contains any structure which is returned from a function.
`intsave`	Holds the W register saved by the interrupt service routine. If necessary, the W register will also be saved in the `intsave_n` psects.
`intsave_n`	(Where *n* is a non-zero bank number) may also hold the W register saved by the interrupt service routine. (See the description of the `intsave` psect.)
`temp`	Is used to store scratch variables used by the compiler. These include function return values larger than a **`char`** and values passed to and returned from library routines. If possible, this will be positioned in the common area of the processor.
`xtemp`	Is used to store scratch variables used by the compiler and to pass values to and from the library routines.

5.11 Interrupt Handling in C

The compiler incorporates features allowing the PIC interrupt to be handled without writing any assembler code. The function qualifier **`interrupt`** may be applied to one function to allow it to be called directly from the hardware interrupt. The compiler will process the **`interrupt`** function differently to any other functions, generating code to save and restore any registers used and exit using the **`retfie`** instruction instead of a **`retlw`** or **`return`** instructions at the end of the function.

(If the `PICC` 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.

5.11.1 Midrange Interrupt Functions

An example of an **`interrupt`** function for a midrange PIC processor is shown here.

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

As there is a maximum of one interrupt vector in the midrange PIC series. Only one **`interrupt`** function may be defined. The interrupt vector will automatically be set to point to this function.

5.11.2 High-End Interrupt Functions

An example of an **`interrupt`** function for a High-End PIC processor is shown here.

```
long tick_count;

void interrupt tc_int(void) @ 0x10
{
        ++tick_count;
}
```

More than one interrupt vector is allocated to the High-End PIC series. An interrupt function must be defined for each interrupt vector that is to be used. The interrupt vector corresponding to an interrupt function is indicated by the `@` symbol followed by an address. Thus, in the above example, the function **`tc_int`** will be executed whenever the PC is loaded with the vector 10h.

Although you may define as many interrupt functions as required, only two interrupts may be active at any given time, and these must be set up using the **`interrupt_level`** pragma to ensure correct operation.

5.11.3 Context Saving on Interrupts

The PIC processor only saves the PC on its stack whenever an interrupt occurs. Other registers and objects must be saved in software. The PICC compiler determines which registers and objects are used by an **`interrupt`** function and saves these appropriately.

If the **`interrupt`** routine calls other functions and these functions are defined before the **`interrupt`** code in the same module, then any registers used by these functions will be saved as well. If the called functions have not been seen by the compiler, a worst case scenario is assumed and all registers and objects will be saved.

PICC does not scan assembly code which is placed in-line within the **`interrupt`** function for register usage. Thus, if you include in-line assembly code into an **`interrupt`** function, you may have to add extra assembly code to save and restore any registers or locations used.

5.11.3.1 MidRange Context Saving

The code associated with **`interrupt`** functions that do not require registers or objects is placed directly at the interrupt vector in a psect called `intcode`.

If context saving is required, this is performed by code placed in to a psect called `intentry` which will be placed at the interrupt vector. Any registers or objects to be saved are done so to areas of memory especially reserved for this purpose.

If the W register is to be saved, it is stored to memory reserved in the `intsave_0` psect which is located in Bank 0. If the processor for which the code is written has more than one RAM bank, it is impossible to swap to Bank 0 without corrupting W, so an `intsave_`*`n`* psect is allocated to each RAM bank (where *`n`* represents the bank number). The addresses of these memory areas are identical in the lower seven

bits.When the interrupt occurs, W will be saved into one of these memory areas depending on the bank in which the processor was in before the interrupt occurred.

If the STATUS register is to be saved, it is stored into memory reserved in the `intsave` psect which resides in Bank 0.

Some C code, for example division, may call an assembly routine which used temporary RAM locations. If these are used during the **`interrupt`** function, they too will be saved by separate routines which are automatically linked.

5.11.3.2 High-End Context Saving

The ALUSTA, BSR and PCLATH registers are automatically saved by code in a psect called `intcode`. This code is placed at the interrupt vector. This code includes a jump to the user's interrupt code. If the **`interrupt`** function uses the W register, this will also be saved before the jump is taken. These registers are saved to an unbanked area of RAM so that it does not matter which RAM bank is selected when the interrupt occurs.

5.11.4 Context Retrieval

Any objects saved by the compiler are automatically restored before the **`interrupt`** function returns. Midrange PIC restoration code is placed into a psect called `int_ret`. High-End PIC restoration code is placed immediately after the code associated with the **`interrupt`** function.

5.11.5 Interrupt Levels

Normally it is assumed by the compiler that any interrupt may occur at any time, and an error will be issued by the linker if a function appears to be called by an interrupt and by main-line code, or another interrupt. Since it is often possible for the user to guarantee this will not happen for a specific routine, the compiler supports an interrupt level feature.

This is achieved with the **`#pragma interrupt_level`** directive. There are two interrupt levels available, and any **`interrupt`** functions at the same level will be assumed by the compiler to be mutually exclusive. (Since the midrange PIC devices only support one active interrupt there is no value in using more than one interrupt level when these processors are selected.) This exclusion must be guaranteed by the user - the compiler is not able to control interrupt priorities. Each interrupt routine may be assigned a single level, either 0 or 1.

In addition, any non-**`interrupt`** functions that are called from an **`interrupt`** function and also from main-line code may also use the **`#pragma interrupt_level`** directive to specify that they will never be called by **`interrupt`** functions of one or more levels. This will prevent linker from issuing an error message because the function was included in more than one call graph. Note that it is entirely up to the user to ensure that the function is NOT called by both main-line and interrupt code at the same time.

This will normally be ensured by disabling interrupts before calling the function. It is not sufficient to disable interrupts inside the function after it has been called.

An example of using the interrupt levels is given below. Note that the **#pragma** directive applies to only the immediately following function. Multiple **#pragma interrupt_level** directives may precede a non-**interrupt** function to specify that it will be protected from multiple interrupt levels.

```
/* non-interrupt function called by interrupt and by main-line
code */
#pragma interrupt_level 1
void bill(){
        int     i;
        i = 23;
}

/* two interrupt functions calling the same non-interrupt
function */
#pragma interrupt_level 1

void interrupt fred(void)
{
        bill();
}

#pragma interrupt_level 1
void interrupt joh()
{
        bill();
}

main()
{
        bill();
}
```

5.11.6 Multiple Interrupts on High-End PICs

On High-End devices, the compiler allows a maximum of two **interrupt** routines to be active at any given time. This is handled by the **#pragma interrupt_level** directive, see "Interrupt Levels" on page 166. The only available interrupt levels are levels 0 and 1. Interrupts are level 0 by default.

All level 0 interrupts share the same memory for saving context when the interrupt occurs. Level 1 interrupts use a separate memory area. Thus only one interrupt of each level may be active at any given time. It is up to the user to ensure that this is the case. Interrupts may be re-enabled during an **interrupt** routine as long as the only possible interrupt will be of a different level.

5.11.7 Enabling Interrupts

Two macros are available in <pic.h> which control the masking of all available interrupts. These macros are **ei()**, which enable or unmask all interrupts, and **di()**, which disable or mask all interrupts. On High-End PIC devices, these macros affect the GLINTD bit in the CPUSTA register; in midrange PIC devices, they affect the GIE bit in the INTCON register.

These macros should be used once the appropriate interrupt enable bits for the interrupts that are required in a program have been enabled. For example:

```
ADIE = 1;         // A/D interrupts will be used
PEIE = 1;         // all peripheral interrupts are enabled
ei();             // enable all interrupts
di();             // disable all interrupts
```

5.12 Mixing C and Assembler Code

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 as separate .as source files, assembled by the assembler (ASPIC) 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 through carry instruction on the PIC:

```
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 ASPIC. The full PIC assembler code for **rotate_left()** would be something like:

```
            processor       16C84

            PSECT   text0,class=CODE,local,delta=2
            GLOBAL  _rotate_left
            SIGNAT  _rotate_left,4201
_rotate_left
            ; Fred is passed in the W register - assign it
            ; to ?a_rotate_left.
            movwf   ?a_rotate_left

            ; Rotate left. The result is placed in the W register.
            rlf     ?a_rotate_left,w

            ; The return is already in the W register as required.
            return

            FNSIZE  _rotate_left,1,0
            GLOBAL  ?a_rotate_left
            END
```


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 (via the **FNSIZE** directive), 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.

An assembler language function called from C code must ensure that bank zero is selected before returning. See 5.6.3 on page 157 for more information.

5.12.2 Accessing C objects from within assembler

Global C objects may be directly accessed from within assembly code using their name prepended with an *underscore* character. For example, the object **foo** defined globally in a C module:

```
int foo;
```

may be access from assembler as follows.

```
GLOBAL   _foo
movwf    _foo
```

If the assembler is contained in a different module, then the **GLOBAL** assembler directive should be used in the assembler code to make the symbol name available, as above. If the object is being accessed from in-line assembly in another module, then an extern declaration for the object can be made in the C code, for example:

```
extern int foo;
```

This declaration will only take effect in the module if the object is also accessed from within C code. If this is not the case then, an in-line **GLOBAL** assembler directive should be used. Care should be taken in the object is defined in a bank other than 0. The address of a C object includes the bank information which must be stripped before the address can be used in PIC instructions. Failure to do this may result in fixup errors issued by the linker. For example, if the object **fred** is defined for the PIC16C77 processor as follows:

```
bank2 int fred;
```

then writing the value 0x55 to **fred** via assembler could be performed as follows.

```
movlw    055h
bcf      3,5
bsf      3,6 ; select bank 2
movwf    _fred&07Fh
```

The PIC16C77 processor has banks which are 0x80 bytes long and hence has a 7-bit field in its instructions for RAM addresses. The mask value used, in this case 0x7F, may be different if other processors are used. Check your PIC programmer's guide for information relating to the bank sizes and instruction formats.

If in doubt as to writing assembler which access C objects, write code in C which performs a similar task to what you intend to do and study the assembler listing file produced by the compiler.

5.12.3 #asm, #endasm and asm()

PIC 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;
void main(void)
{

        var = 1;
#asm
        rlf _var,f
#endasm
        asm("rlf _var,f");
}
```

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 PICC -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. you should use only the **asm("")** form, which is interpreted as a C statement and will correctly interact with all C flow-of-control structures.

5.13 Preprocessing

All C source files are preprocessed before compilation. Assembler files can also be preprocessed if the -p command-line option is issued.

5.13.1 Preprocessor Directives

The PICC compiler accepts several specialised preprocessor directives in addition to the standard directives. These are listed in Table 5 - 7 on page 172.

Macro expansion using arguments can use the **#** character to convert an argument to a string, and the **##** sequence to concatenate tokens.

5.13.2 Predefined Macros

The compiler drivers define certain symbols to the preprocessor (CPP), allowing conditional compilation based on chip type, memory model etc. The symbols listed in Table 5 - 8 show the more common symbols defined by the drivers. Each symbol, if defined, is equated to 1 unless otherwise stated.

Table 5 - 7 Preprocessor directives

Directive	Meaning	Example
#	preprocessor null directive, do nothing	#
#assert	generate error if condition false	#assert SIZE > 10
#asm	signifies the begining of in-line assembly	#asm movlw 10h #endasm
#define	define preprocessor macro	#define SIZE 5 #define FLAG #define add(a,b) ((a)+(b))
#elif	short for #else #if	see #ifdef
#else	conditionally include source lines	see #if
#endasm	terminate in-line assembly	see #asm
#endif	terminate conditional source inclusion	see #if
#error	generate an error message	#error Size too big

#if	include source lines if constant expression true	#if SIZE < 10
		c = process(10)
		#else
		skip();
		#endif

#ifdef	include source lines if preprocessor symbol defined	#ifdef FLAG
		do_loop();
		#elif SIZE == 5
		skip_loop();
		#endif

#ifndef	include source lines if preprocessor symbol not defined	#ifndef FLAG
		jump();
		#endif

#include	include text file into source	#include <stdio.h> #include "project.h"
#line	specify line number and file name for listing	#line 3 final
#nn	(where *nn* is a number) short for #line *nn*	#20
#pragma	compiler specific options	See section 5.13.3 on page 173
#undef	undefines preprocessor symbol	#undef FLAG
#warning	generate a warning message	#warning Possible conflict

Table 5 - 8 Predefined CPP Symbols

Symbol	When set	Usage
HI_TECH_C	Always	To indicate that the compiler in use is HI-TECH C.
_HTC_VER_MAJOR_	Always	To indicate the integer component of the compiler's version number.
_HTC_VER_MINOR_	Always	To indicate the decimal component of the compiler's version number.
_HTC_VER_PATCH_	Always	To indicate the patch level of the compiler's version number.
MPC	Always	To indicate the code is compiled for the Microchip PIC family.
_PIC12	If 12-bit device	To indicate that this a Baseline PIC device.
_PIC14	If 14-bit device	To indicate that this a Midrange PIC device.
_PIC16	If 16-bit device	To indicate that this a High-End PIC device.
COMMON	If common RAM present	To indicate the presence of a common RAM area.
BANKBITS	To 0, 1 or 2	Assigned 0, 1 or 2 to indicate 1, 2 or 4 available banks of RAM.
GPRBITS	To 0, 1 or 2	Assigned 0, 1 or 2 to indicate 1, 2 or 4 available banks of general purpose RAM.
PROGMEM	If device can R/W program code. Set to 1 or 2	Defined as 1 indicates a device can read program memory, defined as 2 indicates a device can read and write program memory, otherwise undefined.
MPLAB_ICD	If using ICD	To indicate that code is being generated for the MPLAB In-Circuit Debugger

5.13.3 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 - 9 on page 175. Those keywords not discussed elsewhere are detailed below.

5.13.3.1 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 disabled by default.

5.13.3.2 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) const`** to enable this checking for **`printf()`**. You may also use this for any user-defined function that accepts printf-style format strings. The qualifier following the function name is to allow automatic conversion of pointers in variable argument lists. The above example would cast any pointers to strings in RAM to be pointers of the type (**`const char *`**)

Note that the warning level must be set to -1 or below for this option to have effect.

5.13.3.3 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. Code and data for any of the standard C psects may be redirected using a **`#pragma psect`** directive. For example, if all the uninitialised global data in a particular C source file is to be placed into a psect called `otherram`, the following directive should be used:

```
#pragma psect rbss_0=otherram
```

This directive tells the compiler that anything which would normally be placed in the `rbss_0` psect should now be placed in the `otherram` psect.

Placing code in a different psect is slightly different. Code is placed in a multiple psects which have names like: `text0`, `text1`, `text2` and so on. Thus you do not know the exact psect in which code will reside. To redirect code the following preprocessor directive can be used.

```
#pragma psect text%%u=othercode
```

The **`%u`** sequence corresponds to the internal representation of the text psect number. The additional *percent* character is used to ensure that the psect name is scanned correctly.

If you wish to redirect more than one function in a module to individually numbered psects, use the pragma directive for each function as follows.

Table 5 - 9 Pragma Directives

Directive	Meaning	Example
interrupt_level	Allow interrupt function to be called from main-line code. See section 5.11.5 on page 166	`#pragma interrupt_level 1`
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) const`
psect	Rename compiler-defined psect	`#pragma psect text=mytext`
regsused	Specify registers which are used in an interrupt	`#pragma regsused w`
switch	Specify code generation for switch statements	`#pragma switch direct`

```
#pragma psect text%%u=othercode%%u
void function(void)
{
  // function definition etc

#pragma psect text%%u=othercode%%u
void another(void)
{
  // etc
```

This example will define the psect `othercode0` for **`function()`**'s code, and `othercode1` for **`another()`**'s code. The **`%u`** sequence will be incremented for each function.

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 **`#include`** directives and above any other declarations. For example, to declare a group of uninitialized variables which are all placed in a psect called `otherram`, the following technique should be used:

```
--File OTHERRAM.C
#pragma psect   rbss_0=otherram
char  buffer[5];
int   var1, var2, var3;
```

Any files which need to access the variables defined in `otherram.c` should **`#include`** the following header file:

```
--File OTHERRAM.H
extern char  buffer[5];
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 need to use the PICC -L option to specify an extra linker option, or use the linker manually, or use an HPDPIC project to compile and link your code. If you want a nearly standard configuration with the addition of only an extra psect like otherram, you can use the PICC -L option to add an extra -P specification to the linker command. For example:

```
PICC -L-Potherram=50h/400h -16C84 test.obj otherram.obj
```

will link test.obj and otherram.obj with a standard configuration, and the extra otherram psect at 50h in RAM, but not overlapping any valid ROM load address. If you are using the HPDPIC 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 ...**.

5.13.3.4 The #pragma regsused Directive

PICC will automatically save context when an interrupt occurs. The compiler will determine only those registers and objects which need to be saved for the particular interrupt function defined. The **#pragma regsused** directive allows the programmer to further limit the registers and objects that the compiler might save and retrieve on interrupt.

Table 5 - 10 on page 177 shows registers names that would commonly be used with this directive. The register names are not case sensitive and a warning will be produced if the register name is not recognised.

This pragma affects the first interrupt function following in the source code. Code for High-End devices which contains multiple interrupt functions should include one directive for each interrupt function.

For example, to limit the compiler to saving no registers other than the W register and FSR register for an interrupt function, use:

```
#pragma regsused w fsr
```

Even if a register, other than W or FSR, has been used and that register would normally be saved, it will not be saved if this pragma is in effect. The W and/or FSR register will only be automatically saved by the compiler if required.

Table 5 - 10 Valid regsused Register Names

Register Name	Description
w	W register
btemp, btemp+1...btemp+11	btemp temporary area
fsr	indirect data pointer
tablreg	table registers: low and high byte of table pointer and table latch

5.13.3.5 The #pragma switch Directive

Normally the compiler decides the code generation method for switch statements which results in the smallest possible code size. Specifying the **direct** option to the **#pragma switch** directive forces the compiler to generate the table look-up style switch method. This is mostly useful where either timing or code size is an issue for switch statements (ie: state machines) and a jump table is preferred over direct comparison or vice versa. This pragma affects all code generated onwards. The **auto** option may be used to revert to the default behaviour.

5.14 Linking Programs

The compiler will automatically invoke the linker unless requested to stop after producing assembler code (PICC -S option) or object code (PICC -C option).

PICC and HPDPIC by default generate *Intel* HEX files and *Bytecraft* COD. If you use the -BIN option or specify an output file with a .bin filetype using the PICC -O option the compiler will generate a binary image instead. After linking, the compiler will automatically generate a memory usage map which shows the address used by, and the total sizes of, all the memory areas which are used by the compiled code. Note that bit objects are shown separately. For example:

```
Memory Usage Map:
Program ROM  $0000 - $001A   $001B (   27) words
Program ROM  $07EE - $07FF   $0012 (   18) words
                             $002D (   45) words total Program ROM

Bank 0 RAM   $0020 - $0022   $0003 (    3) bytes total Bank 0 RAM
Bank 1 RAM   $00A0 - $00A2   $0003 (    3) bytes total Bank 1 RAM
Bank 0 Bits  $0118 - $0119   $0002 (    2) bits  total Bank 0 Bits
```

The program statistics shown after the summary provides more concise information based on each memory area of the device. This can be used as a guide to the available space left in the device. The access bank is shown in this summary which is indicated as near RAM. This is the memory area used by near-qualified objects. Such objects should be used wherever possible to reduce code size.

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

5.14.1 Replacing Library Modules

Although PICC comes with a librarian (LIBR) which allows you to unpack a library files and replace modules with your own modified versions, you can easily replace a module within a library without having to do this. If you add the source file which contains the library routine you wish to replace on the command-line list of source files then the routine will replace the routine in the library file with the same name. For example, if you wished to make changes to the library function **max()** which resides in the file max.c in the SOURCES directory, you could make a copy of this source file, make the appropriate changes and then compile and use it as follows.

```
PICC -16c926 main.c init.c max.c
```

The code for **max()** in max.c will be linked into the program rather than the **max()** function contained in the standard libraries. Note, that if you replace an assembler module, you may need the -P option to preprocess assembler files as the library assembler files often contain C preprocessor directives.

5.14.2 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 returning a **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 PICC -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 ASPIC **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

```
PICC -S x.c
```

The resultant assembler code includes the following line:

```
SIGNAT  _widget,8297
```

The **SIGNAT** pseudo-op tells the assembler to include a record in the .obj file which associates the value 8297 with symbol **_widget**. The value 8297 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);
```

then a different signature will be generated and the linker will report a signature mis-match which will alert you to the possible existence of incompatible calling conventions.

5.14.3 Linker-Defined 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***.

5.15 Standard I/O Functions and Serial I/O

A number of the standard I/O functions are provided in the C library with the 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 - 11. More details of these functions are in the Library Functions chapter.

Table 5 - 11 Supported STDIO Functions

Function name	Purpose
printf(const char * s, ...)	Formatted printing to stdout
sprintf(char * buf, const char * s, ...)	Writes formatted text to buf

Before any characters can be written or read using these functions, the **putch()** and **getch()** functions must be written. Other routines which may be required include **getche()** and **kbhit()**.

You will find samples of serial code which implements the **putch()** and **getch()** functions in the file serial.c in the SAMPLES directory.

5.16 MPLAB-specific Debugging Information

Certain options and compiler features are specifically intended to help MPLAB perform symbolic debugging. The -FAKELOCAL switch (**Fake local symbols**) performs two functions, both specific to MPLAB. Since MPLAB does not read the local symbol information produced by the compiler, this options generates additional global symbols which can be used to represent local symbols in a program. The format for the symbols is ***function_name.symbol_name***. Thus, if a variable called **foo** was defined inside the function **main()**, MPLAB would allow access to a global object called **main.foo**. This symbol format is not available in assembler code. References to this object in assembler would be via the symbol **_main$foo**. Although this information allows access to most local objects, if there are two or more objects with the same name in the same function, then you will not be able to examine both as they redefine the same symbol.

The -FAKELOCAL switch also alters the line numbering information produced so that MPLAB can better follow the C source when performing source-level stepping.

The -ICD switch (**MPLAB-ICD support**) produces code suitable for the MPLAB In-Circuit Debugger. This debugger is available on some chips and requires certain ROM and RAM addresses to be reserved. The compiler reads information in the picinfo.ini file to determine whether a chip supports the debugger and adjusts the linker options accordingly. When using the debugger, the first instruction at the reset vector is not executed. This instruction will be the first instruction in the powerup.as file which is automatically linked in by the compiler. This should not alter the operation of the program or the debugger and does not necessarily have to be a **nop** instruction.

PICC Macro Assembler

The HI-TECH PIC Macro Assembler assembles source files for the Microchip PIC series of microprocessors. This chapter describes the usage of the assembler and the directives (assembler pseudo-ops and controls) accepted by the assembler.

The PIC Macro Assembler package includes a linker, librarian, cross reference generator and an object code converter.

6.1 Assembler Usage

The assembler is called ASPIC and is available to run on PC and Unix machines. 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:

```
aspic [ 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 182, and a full description of each option follows.

6.2 Assembler Options

The command line options recognised by ASPIC are as follows:

-*processor* This option defines the processor which is being used. See the HI-TECH web site for a current list of supported processors. You can also add your own processors to the compiler. For more information about this, See "Processor Support" on page 129..

-*A* An assembler file with an extension *.opt* will be produced if this option is used. This is useful when checking the optimized assembler produced using the -O option.

-*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

Table 6 - 1 ASPIC Assembler options

Option	Meaning	Default
-*processor*	Define the processor	
-A	Produce assembler output	Produce object code
-C	Produce cross-reference	No cross reference
-C*chipinfo*	Define the chipinfo file	lib\picinfo.ini
-E*format*	Set error format	
-F*length*	Specify listing form length	66
-H	Output hex values for constants	Decimal values
-I	List macro expansions	Don't list macros
-icd	Assemble for use with MPLAB-ICD	No ICD support
-L*listfile*	Produce listing	No listing
-O	Perform optimization	No optimization
-O*outfile*	Specify object name	srcfile.OBJ
-R*address*	Maximum ROM size	
-S	No size error messages	
-U	No undefined symbol messages	
-V	Produce line number info	No line numbers
-W*width*	Specify listing page width	80
-X	No local symbols in OBJ file	

information. The cross reference utility CREF must then be run to produce the formatted cross reference listing.

-*Cchipinfo* Define the chipinfo file to use. This option is not normally required as the chipinfo file *lib\picinfo.ini* is normally not used.

-*E* The default format for an error message is in the form:

```
filename: line: message
```

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.

-*Flength* The default listing pagelength is 66 lines (11 inches at 6 lines per inch). The `-F` option allows a different page length to be specified.

-*H* Particularly useful in conjunction with the `-A` option, this option specifies to output constants as hexadecimal values rather than decimal values.

-*I* This option forces listing of macro expansions and unassembled conditionals which would otherwise be suppressed by a `NOLIST` assembler control. The `-L` option is still necessary to produce a listing.

-*icd* This option is used to tell the assembler to take into account the reduced rom size available when assembling for use with the MPLAB-ICD.

-*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.

-*O* This requests the assembler to perform optimisation on the assembly code. Note that the use of this option slows the assembly down, as the assembler must make an additional pass over the input code.

-*Ooutfile* 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.

-*Raddress* The value, *address*, passed to the assembler with this option is the highest address used by code in ROM. From this value, the assembler can determine how many page bits need to be adjusted for an **`fcall`** or **`ljmp`** pseudo instruction.

-*S* If a byte-size memory location is initialized 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 PIC Assembly Language

The source language accepted by the HI-TECH Software PIC Macro Assembler is described below. All opcode mnemonics and operand syntax are strictly PIC assembly language. Additional mnemonics are documented in this section.

6.3.1 Additional Mnemonics

Apart from the PIC assembly language mnemonics, the PIC assembler includes the *fcall* and *ljmp* mnemonics. These instructions implement *call* and *goto* instructions but with the added job of setting the necessary bits in PCLATH. These additional mnemonics should be used where possible as they make assembler code independent of the final position of the routines that are to be executed.

6.3.2 Assembler Format Deviations

The HI-TECH PIC assembler uses a slightly modified form of assembly language to that specified by Microchip. Certain PIC instructions used by Microchip assembler use the operands ",0" or ",1" to specify the destination for the result of that operation. The HI-TECH PIC assembler uses the more-readable operands ",w" and ",f" to specify the destination register. The W register is selected as the destination when using the ",w" operand, and the file register is selected when using the ",f" operand or if no destination operand is specified. The case of the letter in the destination operand in not important. The Microchip numerical operands cannot be used with the HI-TECH PIC assembler.

6.3.3 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.

6.3.4 Constants

6.3.4.1 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.

Table 6 - 2 ASPIC 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*

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.

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.

In expressions, real numbers are accepted in the usual format, and are interpreted as IEEE 32-bit format. A real number may be converted into the truncated IEEE 24-bit format by using the float24 pseudo-function. Here is an example of its use:

```
movlw   low(float24(31.415926590000002))
```

6.3.4.2 Character Constants

A character constant is a single character enclosed in single quotes ('). Multi character constants may be specified using double quotes. See “Strings” on page 188.

6.3.5 Delimiters

All numbers and identifiers must be delimited by white space, non-alphanumeric characters or the end of a line.

6.3.6 Special Characters

There are a few characters that are special in certain contexts. Within a macro body, the character & is used for token concatenation. To use the bitwise & operator within a macro body, escape it by using && instead. In a macro argument list, the angle brackets < and > are used to quote macro arguments.

6.3.7 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
```

6.3.7.1 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.

6.3.7.2 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.

6.3.7.3 Location Counter

The current location within the active program section is accessible via the symbol `$`.

6.3.7.4 Register Symbols

The working register (W) is available by using its standard name, and case of the register name is not significant. Special Function Registers (SFRs) are available using their standard names, as long as they are already defined. The standard SFRs for each PIC processor are defined in the corresponding header file.

It is not possible to equate a symbol to a register.

6.3.7.5 Labels

A label is a name at the beginning of a statement which is assigned a value equal to the current offset within the current psect (program section). There are two types of labels; symbol labels and 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.

6.3.7.6 Symbolic Labels

A symbolic label may be any symbol, and may or may not be followed by a colon. Here are two examples of legitimate labels:

```
frank

simon44:
```

Symbols not interpreted in any other way are assumed to be labels. Thus the code:

```
movlw 23h
```

```
bananas

movf 37h
```

defines a symbol called `bananas`. Mis-typed assembler instructions can sometimes be treated as labels without an error message being issued. Indentation of a label does not force the assembler to treat it as an mnemonic.

6.3.7.7 Numeric Labels

The assembler implements a system of numeric labels (as distinct from the local labels used in macros) which relieves the programmer from creating new labels within a block of code. A numeric label is a numeric string followed by a colon, 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                        ; Referenced from somewhere
else
1
        .
        .
        .
        decfsz  _counter
        goto    1b
        goto    1f
        .
        .
        .
1                                   ; End of the function
        return
        end
```

Note that even though there are two 1: labels, no ambiguity occurs, since each is referred to uniquely. The *goto 1b* refers to a label further back in the source code, while *goto 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.

6.3.8 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.

6.3.9 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. The usual rules governing the syntax of expressions apply.

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.

6.3.10 Statement Format

Legal statement formats are shown in Table 6 - 4. The second form is only legal with 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, may or may not followed by a colon. There is no limitation on what column or part of the line any part of the statement should appear in.

6.3.11 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 192 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 *text0* psect, and some data being placed in the *rbss_0* psect.

Table 6 - 3 Operators

Operator	Purpose
*	Multiplication
+	Addition
-	Subtraction
/	Division
= *or* eq	Equality
> *or* gt	Signed greater than
>= *or* ge	Signed greater than or equal to
< *or* lt	Signed less than
<= *or* le	Signed less than or equal to
<> *or* ne	Signed not equal to
low	Low byte of operand
high	High byte of operand
highword	High 16 bits of operand
mod	Modulus
&	Bitwise AND
^	Bitwise XOR (exclusive or)
|	Bitwise OR
not	Bitwise complement
<< *or* shl	Shift left
>> *or* shr	Shift right
rol	Rotate left
ror	Rotate right
seg	Segment (bank number) of address
float24	24-bit version of real operand
nul	Tests if macro argument is null

Table 6 - 4 ASPIC Statement formats

Format 1:	label	opcode	operands	; comment
Format 2:	name	pseudo-op	operands	; comment
Format 3:	; comment only			

```
            processor 16C84

            psect   text0,class=CODE,local,delta=2
            adjust
            lcall   clear_fred
            movf    flag
            btfss   3,2
            goto    1f
            incf    fred
            goto    2f
1           decf    fred
2

            psect   rbss_0,class=BANK0,space=1
flag
            ds      1
fred
            ds      1

            psect   text0,class=CODE,local,delta=2
clear_fred
            clrf    fred
            bcf     status,5
            return
```

Note that even though the two blocks of code in the *text0* psect are separated by a block in the *rbss_0* psect, the two *text0* psect blocks will be contiguous when loaded by the linker. In other words, the **`decf fred`** instruction will fall through to the label **`clear_fred`** 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.

6.3.12 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 6 - 5 on page 191, and are detailed below.

Table 6 - 5 ASPIC 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
DB	Define constant byte(s)
DW	Define constant word(s)
DS	Reserve storage
IF	Conditional assembly
ELSEIF	Alternate conditional assembly
ELSE	Alternate conditional assembly
ENDIF	End conditional assembly
FNADDR	Inform linker that a function may be indirectly called
FNARG	Inform linker that evaluation of arguments for one function requires calling another
FNBREAK	Break call graph links
FNCALL	Inform linker that one function calls another
FNCONF	Supply call graph configuration info for linker
FNINDIR	Inform linker that all functions with a particular signature may be indirectly called
FNROOT	Inform linker that a function is the "root" of a call graph
FNSIZE	Inform linker of argument and local variable sizes for a function
MACRO	Macro definition
ENDM	End macro definition
LOCAL	Define local tabs
ALIGN	Align output to the specified boundary
PAGESEL	Generate set/reset instruction to set PCLATH for this page
PROCESSOR	Define the particular chip for which this file is to be assembl.ed.
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

6.3.12.1 GLOBAL

GLOBAL declares a list of symbols which, if defined within the current module, are made public. If the symbols are not defined in the current module, it is a reference to symbols in external modules. Example:

```
GLOBAL  lab1,lab2,lab3
```

6.3.12.2 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
```

6.3.12.3 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 - 6 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 - 6 PSECT flags

Flag	Meaning
ABS	Psect is absolute
BIT	Psect holds bit objects
CLASS	Specify class name for psect
DELTA	Size of an addressing unit
GLOBAL	Psect is global (default)
LIMIT	Upper address limit of psect
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
SIZE	Maximum size of psect
SPACE	Represents area in which psect will reside
WITH	Place psect in the same page as specified 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 BIT flag specifies that a psect hold objects that are 1 bit long. Such psects have a *scale* value of 8 to indicate that there are 8 addressable units to each byte of storage.
- 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.
- The DELTA flag defines the size of an addressing unit. In other words, the number of bytes covered for an increment in the address. This should be DELTA=2 for ROM (i.e. a word) and DELTA=1 (which is the default delta value) for RAM.
- 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.
- The LIMIT flag specifies a limit on the highest address to which a psect may extend.
- 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 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.
- The SPACE flag is used to differentiate areas of memory which have overlapping addresses, but which are distinct. Psects which are positioned in ROM and RAM have a different SPACE value to indicate that ROM address zero, for example, is a different location to RAM address zero. Objects in different RAM banks have the same SPACE value as their full addresses (including bank information) are unique.
- The WITH flag allows a psect to be placed in the same page *with* a specified psect. For example WITH=text0 will specify that this psect should be placed in the same page as the *text0* psect.

Some examples of the use of the PSECT directive follow:

```
PSECT   fred
PSECT   bill,size=100h,global
PSECT   joh,abs,ovrld,class=CODE,delta=2
```

6.3.12.4 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
```

will move the location counter to the beginning of the current psect + 100h. The actual location will not be known until link time. It is possible to move the location counter backwards.

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, delta=2
org     addr
```

where *addr* is an absolute address.

6.3.12.5 EQU

This pseudo-op defines a symbol and equates its value to an expression. For example

```
thomas EQU 123h
```

The identifier thomas will be given the value 123h. EQU is legal only when the symbol has not previously been defined. See also SET on page 194

6.3.12.6 SET

This pseudo-op is equivalent to EQU except that allows a symbol to be re-defined. For example

```
thomas SET 0h
```

6.3.12.7 DEFL

DEFL (define label) is identical to EQU except that it may be used to re-define a symbol.

6.3.12.8 DB

DB is used to initialize storage as bytes. The argument is a list of expressions, each of which will be assembled into one byte. 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,
```

Note that because the size of an address unit in ROM is 2 bytes, the DB pseudo-op will initialise a word with the upper byte set to zero.

6.3.12.9 DW

DW operates in a similar fashion to DB, 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:

```
        DW      -1, 3664h, 'A', 3777Q
```

6.3.12.10 DS

This directive reserves, but does not initialize, memory locations. The single argument is the number of bytes to be reserved. Examples:

```
alabel: DS      23      ;Reserve 23 bytes of memory
xlabel: DS      2+3     ;Reserve 5 bytes of memory
```

6.3.12.11 FNADDR

This directive tells the linker that a function has its address taken, and thus could be called indirectly through a function pointer. For example

```
        FNADDR  _func1
```

tells the linker that *func1()* has its address taken.

6.3.12.12 FNARG

The directive

```
        FNARG   fun1,fun2
```

tells the linker that evaluation of the arguments to function *fun1* involves a call to *fun2*, thus the memory argument memory allocated for the two functions should not overlap. For example, the C function calls

```
fred(var1, bill(), 2);
```

will generate the assembler directive

```
        FNARG   _fred,_bill
```

thereby telling the linker that *bill()* is called while evaluating the arguments for a call to *fred()*.

6.3.12.13 FNBREAK

This directive is used to break links in the call graph information. The form of this directive is as follows:

```
FNBREAK fun1,fun2
```

and is automatically generated when the **interrupt_level** pragma is used. It states that any calls to **fun1** in trees other than the one rooted at **fun2** should not be considered when checking for functions that appear in multiple call graphs. **Fun2()** is typically **intlevel0** or **intlevel1** in compiler-generated code when the **interrupt_level** pragma is used. Memory for the **auto**/parameter area for a **fun1** will only be assigned in the tree rooted at **fun2**.

6.3.12.14 FNCALL

This directive takes the form:

```
FNCALL  fun1,fun2
```

FNCALL is usually used in compiler generated code. It tells the linker that function *fun1* calls function *fun2*. This information is used by the linker when performing call graph analysis. If you write assembler code which calls a C function, use the FNCALL directive to ensure that your assembler function is taken into account. For example, if you have an assembler routine called *_fred* which calls a C routine called *foo()*, in your assembler code you should write:

```
FNCALL  _fred,_foo
```

6.3.12.15 FNCONF

The FNCONF directive is used to supply the linker with configuration information for a *call graph*. FNCONF is written as follows:

```
FNCONF  psect,auto,args
```

where `psect` is the psect containing the call graph, `auto` is the prefix on all *auto* variable symbol names and `args` is the prefix on all function argument symbol names. This directive normally appears in only one place: the runtime startoff code used by C compiler generated code. For the HI-TECH PIC Compiler the PICRT66X.AS routine should include the directive:

```
FNCONF  rbss,?a,?
```

telling the linker that the call graph is in the *rbss* psect, auto variable blocks start with `?a` and function argument blocks start with `?`.

6.3.12.16 FNINDIR

This directive tells the linker that a function performs an indirect call to another function with a particular signature (see the SIGNAT directive). The linker must assume worst case that the function could call any other function which has the same signature and has had its address taken (see the FNADDR directive). For example, if a function called *fred()* performs an indirect call to a function with signature 8249, the compiler will produce the directive:

```
FNINDIR _fred,8249
```

6.3.12.17 FNSIZE

The FNSIZE directive informs the linker of the size of the local variable and argument area associated with a function. These values are used by the linker when building the call graph and assigning addresses to the variable and argument areas. This directive takes the form:

```
FNSIZE  func,local,args
```

The named function has a local variable area and argument area as specified, for example

```
FNSIZE  _fred, 10, 5
```

means the function *fred()* has 10 bytes of local variables and 5 bytes of arguments. The function name arguments to any of the call graph associated directives may be local or global. Local functions are of course defined in the current module, but most be used in the call graph construction in the same manner as global names.

6.3.12.18 FNROOT

This directive tells the assembler that a function is a "root function" and thus forms the root of a call graph. It could either be the C *main()* function or an interrupt function. For example, the C main module produce the directive:

```
FNROOT  _main
```

6.3.12.19 IF, ELSEIF, ELSE and ENDIF

These directives implement conditional assembly. The argument to `IF` and `ELSEIF` 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 `ENDIF` will terminate the conditional assembly block. Example:

```
IF      ABC
lcall   aardvark
ELSEIF  DEF
```

```
lcall    denver
ELSE
lcall    grapes
ENDIF
```

In this example, if *ABC* is non-zero, the first *lcall* instruction will be assembled but not the second or third. If *ABS* is zero and *DEF* is non-zero, the second *lcall* will be assembled but the first and third will not. If both *ABS* and *DEF* are zero, the third *lcall* will be assembled. Conditional assembly blocks may be nested.

6.3.12.20 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:

```
;macro: swap
;args:  arg1, arg2 - the NUMBERS of the variables to swap
;     arg3 - the NAME of the variable to use for temp storage;
;descr: Swaps two specified variables, where the variables
;       are named:
;                        var_x
;       and x is a number.
;uses:  Uses the w register.

swap     macro   arg1, arg2, arg3
         movf    var_&arg1,w
         movwf   arg3
         movf    var_&arg2,w
         movwf   var_&arg1
         movf    arg3,w
         movwf   var_&arg2
         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:

```
         swap     2,4,tempvar
```

expands to:

```
        movf    var_2,w
        movwf   tempvar
        movf    var_4,w
        movwf   var_2
        movf    tempvar,w
        movwf   var_4
```

A point to note in the above example: the **&** character is used to permit the concatenation of macro parameters with other text, but is removed in the actual expansion. The NUL operator may be used within a macro to test a macro argument, for example:

```
if      nul     arg3            ; argument was not supplied.
        ...
else                            ; argument was supplied
        ...
endif
```

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 (;;).

6.3.12.21 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:

```
down    macro   count
        local   more
        movlw   count
        movwf   tempvar
more    decfsz  tempvar
        goto    more
        endm
```

when expanded will include a unique assembler generated label in place of *more*. For example:

```
        down    4
```

expands to:

```
        movlw   4
        movwf   tempvar
??0001  decfsz  tempvar
        goto    ??0001
```

if invoked a second time, the label *more* would expand to *??0002*.

6.3.12.22 ALIGN

The **ALIGN** directive aligns whatever is following, data storage or code etc., to the specified boundary in the psect in which the directive is found. The boundary is specified by a number following the directive and is a number of bytes. For example, to align output to a 2 byte (even) address within a psect, the following could be used.

```
ALIGN 2
```

Note, however, that what follows will only begin on an even absolute address if the psect begins on an even address. The ALIGN directive can also be used to ensure that a psect's length is a multiple of a certain number. For example, if the above ALIGN directive was placed at the end of a psect, the psect would have a length that was always an even number of bytes long.

6.3.12.23 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
addwf   fred, fred
andwf   fred, w
endm
```

will expand to

```
addwf   fred, fred
andwf   fred, w
addwf   fred, fred
andwf   fred, w
addwf   fred, fred
andwf   fred, w
```

6.3.12.24 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:

```
        psect   idata_0

        irp     number,4865h,6C6Ch,6F00h
        dw      number
        endm

        psect   text0
```

would expand to:

```
        psect   idata_0

        dw      4865h
        dw      6C6Ch
        dw      6F00h

        psect   text0
```

Note that you can use *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:

```
        psect   idata_0

        irpc    char,ABC
        dw      'char'
        endm

        psect   text0
```

will expand to:

```
        psect   idata_0

        dw      'A'
```

```
        dw      'B'
        dw      'C'

        psect   text0
```

6.3.12.25 PAGESEL

It's sometimes necessary to set the current PCLATH bits so that a goto will jump to a location in the current page of ROM. The LJMP and FCALL instructions automatically generate the necessary code to set or reset the PCLATH bits, but at other times an explicit directive PAGESEL is used, e.g.

```
PAGESEL         $
```

6.3.12.26 PROCESSOR

The output of the assembler depends on which chip it is desired to assemble for. This can be set on the command line, or with this directive, e.g.

```
PROCESSOR       16C84
```

6.3.12.27 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.

6.3.13 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.

6.3.14 Assembler Controls

Assembler controls may be included in the assembler source to control such things as listing format. These keywords have no significance anywhere else in the program. Some keywords are followed by one or more parameters.

A list of keywords is given in Table 6 - 7, and each is described further below.

Table 6 - 7 ASPIC Assembler controls

Control[a]	Meaning	Format
COND*	Include conditional code in the listing	opt cond
GEN	Expand macros in the listing output	opt gen
INCLUDE	Textually include another source file	include <pathname>
LIST*	Define options for listing output	opt list [<listopt>, ..., <listopt>]
NOCOND	Leave conditional code out of the listing	opt nocond
NOGEN*	Disable macro expansion	opt nogen
NOLIST	Disable listing output	nolist
TITLE	Specify the title of the program	opt title "<title>"
SUBTITLE	Specify the subtitle of the program	opt subtitle "<subtitle>"
PAGELENGTH	Specify the length of the listing form	opt pagelength <n>
PAGEWIDTH	Allows the listing line width to be set	opt pagewidth <n>

a.The default options are listed with an asterix (*).

Please note that most of these controls require *opt* to be prepended to the keyword.

6.3.14.1 COND

Any conditional code will be included in the listing output. See also the NOCOND control.

6.3.14.2 GEN

When GEN is in effect, the code generated by macro expansions will appear in the listing output. See also the NOGEN control.

6.3.14.3 INCLUDE

This control causes the file specified by pathname to be textually included at that point in the assembly file. The INCLUDE control must be the last control keyword on the line.

6.3.14.4 LIST

If the listing was previously turned off using the NOLIST control, the LIST control on its own will turn the listing on.

Alternatively, the LIST control may includes options to control the assembly and the listing. The options are listed in Table 6 - 8. See also the NOLIST control.

Table 6 - 8 LIST Control Options

List Option	Default	Description
c=nnn	80	Set the page (i.e. column) width.
n=nnn	59	Set the page length.
t=ON|OFF	OFF	Truncate listing output lines. The default wraps lines.
p=<processor>	n/a	Set the processor type.
r=<radix>	hex	Set the default radix to hex, dec or oct.
x=ON|OFF	OFF	Turn macro expansion on or off.

6.3.14.5 NOCOND

Any conditional code will not be included in the listing output. See also the COND control.

6.3.14.6 NOGEN

NOGEN disables macro expansion in the listing file. The macro call will be listed instead. See also the GEN control.

6.3.14.7 NOLIST

This control turns the listing output off from this point onwards. See also the LIST control.

6.3.14.8 TITLE

This control keyword defines a title to appear at the top of every listing page. The string should be enclosed in single or double quotes. See also the SUBTITLE control.

6.3.14.9 PAGELENGTH

This control keyword specifies the length of the listing form. The default is 66 (11 inches at 6 lines per inch).

6.3.14.10 PAGEWIDTH

PAGEWIDTH allows the listing line width to be set.

6.3.14.11 SUBTITLE

`SUBTITLE` defines a subtitle to appear at the top of every listing page, but under the title. The string should be enclosed in single or double quotes. See also the `TITLE` control.

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 (HI-TIDE, 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 the **`PSECT`** directive **`class=`** in assembler code. See *The Macro Assembler* chapter for more information on **`psect`** options.

7.5 Global Symbols

The linker handles only symbols which have been declared as **`GLOBAL`** to the assembler. The code generator generates these assembler directives whenever it encounters global C objects. At 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:

```
hlink[1] options files ...
```

Options is zero or more linker options, each of which modifies the behaviour of the linker in some way. *Files* is one or more object files, and zero or more library names. The options recognised by the linker are listed in Table 7 - 1 on page 209 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

1. In earlier versions of HI-TECH C the linker was called LINK.EXE

Table 7 - 1 Linker Options

Option	Effect
-O*outfile*	Specify name of output file
-P*spec*	Specify psect addresses and ordering
-Q*processor*	Specify the processor type (for cosmetic reasons only)
-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=`** directive 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 parameter 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 217.

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* value, or class.

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 the link address is a negative number, the psect is linked in reverse order with the top of the psect appearing at the specified address minus one. Psects following a negative address will be placed before the first psect in memory. 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. Again:

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

will link in assending order `bss`, `data` then `text` with the top of `text` appearing at address 0ffh.

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 of *num* >= 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 objects of type **char** 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 **sprintf()**. 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.

Table 7 - 2 Librarian Options

Option	Effect
-P*width*	specify page width
-W	suppress non-fatal errors

The key letters are listed inTable 7 - 3.

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 object 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 224. Where an address is required, the format is the same as for HLINK:.

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

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 1.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 words 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
-*n*,*m*	Format either Motorola or Intel HEX file where *n* is the max number of bytes per record and *m* specifies the multiple to which the record size is rounded

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 225. Each option is described in more

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*

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 \.

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 -L55 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. -P132 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 -Xxyz would be used. If a digit appears in the character sequence then this will match any digit in the symbol, e.g. -XX0 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 Cromwell

The CROMWELL utility converts code and symbol files into different formats. The formats available are shown in Table 7 - 6.

The general form of the CROMWELL command is:

```
cromwell options input_files -okey output_file
```

where *options* can be any of the options shown in Table 7 - 7. *Output_file* (optional) is the name of the output file. The *input_files* are typically the HEX and SYM file. CROMWELL automatically searches for the SDB files and reads those if they are found. The options are further described in the following paragraphs.

7.13.1 -Pname

The -P options takes a string which is the name of the processor used. CROMWELL may use this in the generation of the output format selected.

Table 7 - 6 Format Types

Key	Format
cod	*Bytecraft* COD file
coff	COFF file format
elf	ELF/DWARF file
eomf51	Extended OMF-51 format
hitech	HI-TECH Software format
icoff	ICOFF file format
ihex	*Intel* HEX file format
omf51	OMF-51 file format
pe	P&E file format
s19	*Motorola* HEX file format

7.13.2 -D

The -D option is used to display to the screen details about the named input file in a readable format.

Table 7 - 7 Cromwell Options

Option	Description
-P*name*	Processor name
-D	Dump input file
-C	Identify input files only
-F	Fake local symbols as globals
-O*key*	Set the output format
-I*key*	Set the input format
-L	List the available formats
-E	Strip file extensions
-B	Specify big-endian byte ordering
-M	Strip underscore character
-V	Verbose mode

The input file can be one of the file types as shown in Table 7 - 6.

7.13.3 -C

This option will attempt to identify if the specified input files are one of the formats as shown in Table 7 - 6. If the file is recognised, a confirmation of its type will be displayed.

7.13.4 -F

When generating a COD file, this option can be used to force all local symbols to be represented as global symbols. The may be useful where an emulator cannot read local symbol information from the COD file.

7.13.5 -Okey

This option specifies the format of the output file. The *key* can be any of the types listed in Table 7 - 6.

7.13.6 -Ikey

This option can be used to specify the default input file format. The *key* can be any of the types listed in Table 7 - 6.

7.13.7 -L

Use this option to show what file format types are supported. A list similar to that given in Table 7 - 6 will be shown.

7.13.8 -E

Use this option to tell CROMWELL to ignore any filename extensions that were given. The default extension will be used instead.

7.13.9 -B

In formats that support different endian types, use this option to specify big-endian byte ordering.

7.13.10 -M

When generating COD files this option will remove the preceeding *underscore* character from symbols.

7.13.11 -V

Turns on verbose mode which will display information about operations CROMWELL is performing.

7.14 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 this application 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.

7.14.1 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 - 8 on page 229. *File* is the name

Table 7 - 8 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*.

7.14.1.1 -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.

7.14.1.2 -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.

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.

'with=' flags are cyclic ***(Assembler)***
If Psect A is to be placed 'with' Psect B, and Psect B is to be placed 'with' Psect A, there is no hierarchy. Remove a 'with' flag from one of the psect declarations.

(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 macro name cannot also be a label *(Assembler)*
A label has been found with the same name as a macro. This is not allowed.

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.

a psect may only be in one class *(Assembler)*
You cannot assign a psect to more than one class. The psect was defined differently at this point than when it was defined elsewhere.

a psect may only have one 'with' option *(Assembler)*
A psect can only be placed 'with' one other psect.

add_reloc - bad size *(Assembler)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

ambiguous chip type * -> * or * *(Driver)*
The chip type specified on the command line is not complete and could refer to more than one chip. Specify the full name of the chip type.

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 option: * ***(Driver)***
The format of a -A option to shift the ROM image was not correct. The -A should be immediately followed by a valid hex number.

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 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 constant in expression ***(Assembler)***
The character constant was expected to consist of only one character, but was found to be greater than one character.

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 common spec in -p option *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

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 extraspecial * *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad format for -p option *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

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 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 popreg: * ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

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

bad pushreg: * *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad putwsize *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

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 switch size * *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

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 uconval - * *(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.

bit variables must be global or static *(Code Generator)*
A bit variable cannot be of type auto. If you require a bit variable with scope local to a block of code or function, qualify it static.

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.

c= must specify a positive constant *(Assembler)*
The parameter to the LIST assembler control's 'C' option (which sets the column width of the listing output) must be a positive constant number.

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 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 '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 initialize bit type *(Code Generator)*
Variables of type bit cannot be initialised.

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 * for input *(Cref)*
Cref cannot open the specified file.

can't open * for output *(Cref)*
Cref cannot open the specified file.

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 chip info file * *(Assembler)*
The chipinfo file (libpicinfo.ini by default) could not be opened. It may have been incorrectly specified.

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.

case 55 on pic17! ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

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.

chip name * not found in chipinfo file *(Driver)*
The chip type specified on the command line was not found in the chipinfo INI file. The compiler doesn't know how to compile for this chip. If this is a device not yet supported by the compiler, you might be able to add the memory specifications to the chipinfo file and try again.

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 may not be in absolute psect *(Assembler)*
If a symbol is defined as common, you cannot place it in a psect which is absolute.

common symbol psect conflict: * *(Linker)*
A common symbol has been defined to be in more than one psect.

compiler already in use - try again later *(Driver)*
The driver detected the presence of a lock file that indicates that another instance of the compiler is still running. If you are running on a network, then there may well be someone else using the compiler, and you only have a single user licence. If the compiler was not quit properly then this lock file may not have been deleted. Run the following command:

HPDPIC clear_locks

The argument must be in lower case.

compiler not installed properly - reinstall and try again ***(Driver)***
This is a message from the compiler's security system. Firstly, to move the compiler from one drive to another, or even from one directory to another, you must reinstall. You cannot copy the installed compiler (even backing up and restoring will not work unless you simply restore over the existing files). If you have reinstalled, then it is possible that you are running an older version of the same compiler still installed on your machine. Check your PATH environment variable to make sure you're running what you think you are, i.e. make sure your PATH specifies the newly installed compiler.

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.

conversion to shorter data type ***(Code Generator)***
Truncation may occur in this expression as the lvalue is of shorter type than the rvalue.

copyexpr: can't handle v_rtype = * ***(Assembler)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

couldn't create error file: * ***(Driver)***
The error file specified after the -Efile or -E+file options could not be opened. Check to ensure that the file or directory is not specified read only.

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.

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.

delta= must specify a positive constant ***(Assembler)***
The parameter to the PSECT assembler directive's 'DELTA' option must be a positive constant number.

demo version will only generate * lines of code ***(Assembler)***
The demonstration version of the compiler is limited in the size of the code it will generate. The retail version has no limitation on the number of lines of code generated.

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.

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.

ds argument must be a positive constant ***(Assembler)***
The argument to the DS assembler directive must be a positive constant.

duplicate * for * in chipinfo file at line * ***(Assembler, Driver)***
The chipinfo file (libpicinfo.ini by default) has a processor section with multiple values for the ARCH, BANKS, INTSAVE, LIB, ROMSIZE, SPAREBIT, START or ZEROREG field. Only one value is allowed per chip.

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 arch for * in chipinfo file at line * *(Assembler)*
The chipinfo file (libpicinfo.ini by default) has a processor section with multiple ARCH values. Only one ARCH value is allowed.

duplicate banks for * in chipinfo file at line * *(Assembler)*
The chipinfo file (libpicinfo.ini by default) has a processor section with multiple BANKS values. Only one BANKS value is allowed.

duplicate case label * *(Code Generator)*
There is more than one case label with this value in a 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 lib for * in chipinfo file at line * *(Assembler)*
The chipinfo file (libpicinfo.ini by default) has a processor section with multiple LIB values. Only one LIB value is allowed.

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.

duplicate romsize for * in chipinfo file at line * *(Assembler)*
The chipinfo file (libpicinfo.ini by default) has a processor section with multiple ROMSIZE values. Only one ROMSIZE value is allowed.

duplicate sparebit for * in chipinfo file at line * *(Assembler)*
The chipinfo file (libpicinfo.ini by default) has a processor section with multiple SPAREBIT values. Only one SPAREBIT value is allowed.

duplicate zeroreg for * in chipinfo file at line * *(Assembler)*
The chipinfo file (libpicinfo.ini by default) has a processor section with multiple ZEROREG values. Only one ZEROREG value is allowed.

empty chip info file * *(Assembler)*
The chipinfo file (libpicinfo.ini by default) contains no data.

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.

field width not valid at this point ***(Parser)***
A field width may not appear at this point in a printf() type format specifier.

file locking not enabled on network drive ***(Driver)***
The driver has attempted to modify the lock file located in the LIB directory butwas unable to do so. This has probably resulted fro mthe network drive used to hold the compiler being read only.

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.

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.

form length must be >= 15 ***(Assembler)***
The form length specified using the -Flength option must be at least 15 lines.

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!

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 instruction for this processor ***(Assembler)***
The instruction is not supported by this processor.

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. Re-compile 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.

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 function * may only have one interrupt level *(Code Generator)*
Only one interrupt level may be associated with an interrupt function. Check to ensure that only one interrupt_level pragma has been used with the function specified.

interrupt function requires an address *(Code Generator)*
The Highend PIC devices support multiple interrupts. An "@ address" is required with the interrupt definition to indicate with which vector this routine is associated.

interrupt functions not implemented for 12 bit pic *(Code Generator)*
The 12-bit range of PIC processors do not support interrupts.

interrupt level may only be 0 (default) or 1 *(Code Generator)*
The only possible interrupt levels are 0 or 1. Check to ensure that all interrupt_level pragmas use these levels.

interrupt_level should be 0 to 7 *(Parser)*
The pragma 'interrupt_level' must have an argument from 0 to 7.

invalid * limits in chipinfo file at line * *(Driver)*
The ranges of addresses for the ram banks or common memory supplied in the chipinfo INI file is not valid for architecture specified.

invalid address after 'end' directive ***(Assembler)***
The start address of the program which is specified after the assembler 'end' directive must be a label in the current file.

invalid argument to float24 ***(Assembler)***
An argument to the float24 directive must be a number or a symbol which has been equated to a number.

invalid character ('*') in number ***(Assembler)***
A number contained a character that was not part of the range 0-9 or 0-F.

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.

invalid size for fnsize directive ***(Assembler)***
The assembler FNSIZE assembler directive arguments must be positive constants.

inverted common bank in chipinfo file at line * ***(Assembler, Driver)***
The second hex number specified in the COMMON field in the chipinfo file (libpicinfo.ini by default) must be greater in value than the first.

inverted ICD ROM address in chipinfo file at line * ***(Driver)***
The second hex number specified in the ICD ROM address field in the chipinfo file (libpicinfo.ini by default) must be greater in value than the first.

inverted ram bank in chipinfo file at line * ***(Assembler, Driver)***
The second hex number specified in the RAM field in the chipinfo file (libpicinfo.ini by default) must be greater in value than the first.

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 arch specification for * in chipinfo file *(Assembler)*
The chipinfo file (libpicinfo.ini by default) has a processor section without an ARCH values. The architecture of the processor must be specified.

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.

n= must specify a positive constant *(Assembler)*
The parameter to the LIST assembler control's 'N' option (which sets the page length for the listing output) must be a positive constant number.

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 case labels *(Code Generator)*
There are no case labels in this switch statement.

no common RAM in PIC17Cxx device *(Driver)*
The chipinfo INI file did not specify a range of common memory for a PIC16 architecture chip.

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 interrupt strategy available *(Code Generator)*
The processor does not support saving and subsequent restoring of registers during an interrupt service routine.

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 valid entries in chipinfo file ***(Assembler)***
The chipinfo file (libpicinfo.ini by default) contains no valid processor descriptions.

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".

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.

org argument must be a positive constant *(Assembler)*
An argument to the ORG assembler directive must be a positive constant or a symbol which has been equated to a positive constant.

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 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.

out-of-range case label * *(Code Generator)*
This case label is not a value that the controlling expression can yield, and thus this label will never be selected.

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.

overfreed *(Assembler)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

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.

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.

popreg: bad reg (*) *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

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.

processor type not defined ***(Assembler)***
The processor must be defined either from the command line (eg. -16c84), via the PROCESSOR assembler directive, or via the LIST assembler directive.

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 delta redefined ***(Assembler)***
The DELTA parameter to the PSECT assembler directive's is different from a previous PSECT directive.

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 relocability redefined *(Assembler)*
The RELOC parameter to the PSECT assembler directive's is different from a previous PSECT directive.

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.

psect space redefined *(Assembler)*
The psect space has already been defined using the psect SPACE flag elsewhere.

pushreg: bad reg (*) *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

qualifiers redeclared *(Parser)*
This function has different qualifiers in different declarations.

radix must be from 2 - 16 *(Assembler)*
The radix specified using the RADIX or LIST assembler directive must be in the range from 2 (binary) to 16 (hexadecimal).

range check too complex *(Assembler)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

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.

reloc= must specify a positive constant *(Assembler)*
The parameter to the PSECT assembler directive's 'RELOC' option must be a positive constant number.

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.

set_fact_bit on pic17! *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

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.

size= must specify a positive constant *(Assembler)*
The parameter to the PSECT assembler directive's 'SIZE' option must be a positive constant number.

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.

space= must specify a positive constant ***(Assembler)***
The parameter to the PSECT assembler directive's 'SPACE' option must be a positive constant number.

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.

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 * is not external *(Assembler)*
A symbol has been declared as EXTRN but is also defined in the current module.

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.

symbol cannot be both extern and public *(Assembler)*
If the symbol is declared as extern, it is to be imported. If it is declared as public, it is to be exported from the current module. It is not possible for a symbol to be both.

symbol has been declared extern *(Assembler)*
A symbol has been declared in the current module, but has previously been declared extern. A symbol cannot be both local and extern.

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 chipinfo file at line * *(Assembler)*
The chipinfo file (libpicinfo.ini by default) contains non-standard syntax at the specified line.

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 common lines in chipinfo file for * *(Assembler, Driver)*
The chipinfo file (libpicinfo.ini by default) contains a processor section with too many COMMON fields. Only one COMMON field is allowed.

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 object files *(Driver)*
A maximum of 128 object files may be passed to the linker. The driver exceeded this amount when generating the command line for the linker.

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 rambank lines in chipinfo file for * *(Assembler, Driver)*
The chipinfo file (libpicinfo.ini by default) contains a processor section with too many RAMBANK fields. Reduce the number of values.

too many references to * *(Cref)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

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 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 addressing mode * ***(Assembler, Optimiser)***
An unknown addressing mode was used in the assembly file.

unknown architecture in chipinfo file at line * ***(Assembler, Driver)***
An chip architecture (family) that is unknown was encountered when reading the chip INI file. Valid architectures are: PIC12, PIC14 and PIC16, representing baseline, midrange and highend devices, respectively.

unknown argument to 'pragma switch': * *(Code Generator)*
The '#pragma switch' directive has been used with an invalid switch code generation method. Possible arguments are: auto, simple and direct.

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 op * in emobj *(Assembler)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

unknown op * in size_psect *(Assembler)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

unknown op in emasm(): * *(Assembler)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

unknown option * *(Preprocessor)*
This option to the preprocessor is not recognized.

unknown pragma * *(Parser)*
An unknown pragma directive was encountered.

unknown predicate * *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

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.

unrecognised line in chipinfo file at line * *(Assembler)*
The chipinfo file (libpicinfo.ini by default) contains a processor section with an unrecognised line. Look in the chipinfo file for the possibilities.

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 * must be qualified 'const' to be initialised *(Parser)*
Any initialised variable must be declared 'const', as all initialised variables are placed in ROM, with no copy placed in RAM.

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 PICC 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.

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.

ATAN

Synopsis

```
#include <math.h>

double atan (double x)
```

Description

This function returns the arc tangent of its argument, i.e. it returns an angle `e` in the range $-\pi/2$ to $\pi/2$ such that `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.

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.

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));
}
```

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.

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 <pic.h>

void ei(void)
void di(void)
```

Description

ei and ***di*** enable and disable interrupts respectively. These are implemented as macros defined in ***pic.h***. They will expand to an in-line assembler instruction that sets or clears the interrupt enable 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 <pic.h>

long count;

void interrupt tick(void)
{
    count++;
}

long getticks(void)
{
    long val;    /* Disable interrupts around access
         to count, to ensure consistency.*/
    di();
    val = count;
    ei();
    return val;
}
```

DIV

Synopsis

```
#include <stdlib.h>

div_t div (int numer, int demon)
```

Description

The **div()** function computes the quotient and remainder of the numerator divided by the denominator.

Example

```
#include <stdlib.h>
#include <stdio.h>

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;
```

EEPROM_READ, EEPROM_WRITE

Synopsis

```
#include <pic.h>

unsigned char eeprom_read (unsigned char addr);
void eeprom_write (unsigned char addr, unsigned char value);
```

Description

These function allow access to the on-board eeprom (when present). The eeprom is not in the directly-accessible memory space and a special byte sequence is loaded to the eeprom control registers to access the device. Writing a value to the eeprom is a slow process and the **`eeprom_write()`** function polls the appropriate registers to ensure that any previous writes have completed before writing the next piece of data. Reading data is completed in the one cycle and no polling is necessary to check for a read completion.

Example

```
#include <pic.h>

void
main (void)
{
    unsigned char data;
    unsigned char address;

    address = 0x10;
    data = eeprom_read(address);
}
```

Note

It may be necessary to poll the eeprom registers to ensure that the write has completed if an ***`eeprom_write()` call is immediately followed by an `eeprom_read()`. The global interrupt enable bit (GIE) is now restored by the `eeprom_write()` routine. The EEIF interrupt flag is not reset by this function.***

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**.

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()`

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));
}
```

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()

GET_CAL_DATA

Synopsis

```
#include <pic.h>

double get_cal_data (const unsigned char * code_ptr)
```

Description

This function returns the 32-bit floating point calibration data from the PIC 14000 calibration space. Only use this function to access KREF, KBG, VHTHERM and KTC (that is, the 32-bit floating point parameters). FOSC and TWDT can be accessed directly as they are bytes.

Example

```
#include <pic.h>

void
main (void)
{
    double x;
    unsigned char y;

        /* Get the slope reference ratio. */
    x = get_cal_data(KREF);

        /* Get the WDT time-out. */
    y =TWDT;
}
```

Return Value

The value of the calibration parameter

Note

This function can only be used on the PIC 14000.

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;
};
```

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()`

KBHIT

Synopsis

```
#include <conio.h>

bit 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. *Note:* the return value is a bit.

Note

The body of the routine will need to be implemented by the user. The skeleton function will be found in the sources direstory.

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.

MEMCHR

Synopsis

```
#include <string.h>

          /* For baseline and midrange processors */
const void * memchr (const void * block, int val, size_t length)

          /* For high-end processors */
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>

          /* For baseline and midrange processors */
void * memcpy (void * d, const void * s, size_t n)

          /* For high-end processors */
far void * memcpy (far 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>

          /* For baseline and midrange processors */
void * memmove (void * s1, const void * s2, size_t n)

          /* For high-end processors */
far void * memmove (far 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>

          /* For baseline and midrange processors */
void * memset (void * s, int c, size_t n)

          /* For high-end processors */
far void * memset (far 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**.

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

Synopsis

```
#include <stdio.h>

unsigned char printf (const char * fmt, ...)
```

Description

The **printf()** function is a formatted output routine, operating on stdout. There are corresponding routines operating 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. Field widths and precision are only supported on the midrange and high-end processors, with the precision specification only applicable to **%s**.

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 for the Baseline series are:

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. 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.

For the Midrange and High-end series, the types of conversions are as for the Baseline with the addition of:

l

Long integer conversion - Preceding the integer conversion key letter with an **l** indicates that the argument list is long.

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.

Example

```
printf("Total = %4d%%", 23)
       yields 'Total =   23%'

printf("Size is %lx" , size)
       where size is a long, prints size
       as hexadecimal.

Note that the precision number is only available when using Midrange
and High-end processors when using the %s  placeholder.
printf("Name = %.8s", "a1234567890")
       yields 'Name = a1234567'

Note that the variable width number is only available when using Midrange
and High-end processors placeholder.
printf("xx%*d", 3, 4)
       yields 'xx  4'

/* vprintf example */

#include <stdio.h>

int
error (char * s, ...)
{
    va_list ap;

    va_start(ap, s);
    printf("Error: ");
    vprintf(s, ap);
    putchar('\n');
    va_end(ap);
}

void
```

```
main (void)
{
    int i;

    i = 3;
    error("testing 1 2 %d", i);
}
```

See Also

`sprintf()`

Return Value

The **`printf()`** routine returns the number of characters written to `stdout`.
NB The return value is a char, NOT an `int`.

Note

Certain features of printf are only available for the midrange and high-end processors. Read the description for details. Printing floating point numbers requires that the float to be printed be no larger than the largest possible long integer. In order to use long or float formats, the appropriate supplemental library must be included. See the description on the PICC -L option and the HPDPIC Options/Long formats in printf menu for more details.

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.***

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

Synopsis

```
#include <stdio.h>

          /* For baseline and midrange processors */
unsigned char sprintf (char *buf, const char * fmt, ...)

          /* For high-end processors */
unsigned char sprintf (far char *buf, const char * fmt, ...)
```

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.

See Also

`printf()`

Return Value

The **sprintf()** routine returns the number of characters placed into the buffer.
NB: The return value is a `char` not an `int`.

Note

For High-end processors the buffer is accessed via a far pointer.

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()`

STRCAT

Synopsis

```
#include <string.h>

          /* For baseline and midrange processors */
char * strcat (char * s1, const char * s2)

          /* For high-end processors */
far char * strcat (far 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>

          /* For baseline and midrange processors */
const char * strchr (const char * s, int c)
const char * strichr (const char * s, int c)

          /* For high-end processors */
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

The functions 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>

          /* For baseline and midrange processors */
char * strcpy (char * s1, const char * s2)

          /* For high-end processors */
far char * strcpy (far 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.

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.

STRNCAT

Synopsis

```
#include <string.h>

          /* For baseline and midrange processors */
char * strncat (char * s1, const char * s2, size_t n)

          /* For high-end processors */
far char * strncat (far 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>

          /* For baseline and midrange processors */
char * strncpy (char * s1, const char * s2, size_t n)

          /* For high-end processors */
far char * strncpy (far 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>

          /* For baseline and midrange processors */
const char * strpbrk (const char * s1, const char * s2)

          /* For high-end processors */
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>

          /* For baseline and midrange processors */
const char * strrchr (char * s, int c)
const char * strrichr (char * s, int c)

          /* For high-end processors */
char * strrchr (char * s, int c)
char * strrichr (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>

          /* For baseline and midrange processors */
const char * strstr (const char * s1, const char * s2)
const char * stristr (const char * s1, const char * s2)

          /* For high-end processors */
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>

          /* For baseline and midrange processors */
char * strtok (char * s1, const char * s2)

          /* For high-end processors */
far char * strtok (far 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.***

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.

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");
}
```

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.

Index

Symbols

Numerics

A

B

C

D

E

F

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

PICC Options

Option	Meaning
-*processor*	Define the processor
-A*spec*	Specify offset for ROM
-A-*option*	Specify -*option* to be passed directly to the assembler
-AAHEX	Generate an American Automation symbolic HEX file
-ASMLIST	Generate assembler .LST file for each compilation
-BANKCOUNT=*count*	Set number of banks to use
-BIN	Generate a Binary output file
-C	Compile to object files only
-CK*file*	Make OBJTOHEX use a checksum file
-CR*file*	Generate cross-reference listing
-D24	Use truncated 24-bit floating point format for doubles
-D32	Use IEEE754 32-bit floating point format for doubles
-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
-FAKELOCAL	Produce MPLAB-specific debug information
-FDOUBLE	Enables the use of faster 32-bit floating point math routines.
-G*file*	Generate enhanced source level symbol table
-HELP	Print summary of options
-ICD	Compile code for MPLAB-ICD
-I*path*	Specify a directory pathname for include files
-INTEL	Generate an Intel HEX format output file (default)
-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
-MPLAB	Specify compilation and debugging under MPLAB IDE
-MOT	Generate a Motorola S1/S9 HEX format output file
-N*size*	Specify identifier length
-NORT	Do not link standard runtime module
-NO_STRING_PACK	Disables string packing optimizations
-O	Enable post-pass optimization
-O*file*	Specify output filename
-P	Preprocess assembler files
-PRE	Produce preprocessed source files only
-PROTO	Generate function prototype information
-PSECTMAP	Display complete memory segment usage after linking
-Q	Specify quiet mode
-RESRAM*ranges*	Reserve the specified RAM address ranges.
-RESROM*ranges*	Reserve the specified ROM address ranges.
-ROM*ranges*	Specify external ROM memory range available
-S	Compile to assembler source files only
-SIGNED_CHAR	Make the default char signed.
-STRICT	Enable strict ANSI keyword conformance
-TEK	Generate a Tektronix HEX format output file
-U*symbol*	Undefine a predefined pre-processor symbol
-UBROF	Generate an UBROF format output file
-V	Verbose: display compiler pass command lines
-W*level*	Set compiler warning level
-X	Eliminate local symbols from symbol table
-Zg	Enable global optimization in the code generator