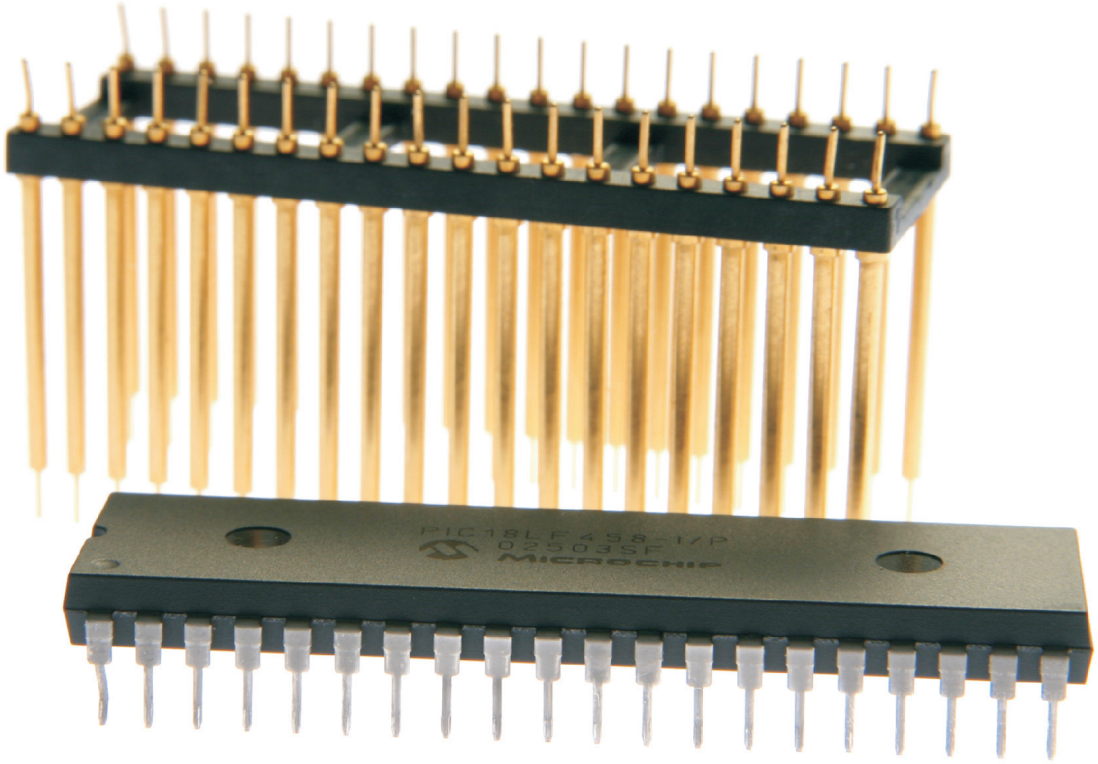
HI-TECH SOFTWARE

Manual

PICC18™ STD

ANSI C COMPILER



HI-TECH PICC-18 STD Compiler

HI-TECH Software.

Copyright (C) 2010 HI-TECH Software.
All Rights Reserved. Printed in Australia.
PICC-18 is licensed exclusively to HI-TECH Software
by Microchip Technology Inc.
Produced on: December 7, 2010

HI-TECH Software Pty. Ltd.
ACN 002 724 549
45 Colebard Street West
Acacia Ridge QLD 4110
Australia

email: hitech@htsoft.com
web: http://www.htsoft.com
ftp: ftp://www.htsoft.com

Contents

List of Tables

Chapter 1

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 also be printed in a `constant-space` type. Assembler code is printed in a font similar to that used by C code.

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

Chapter 2

PICC-18 Command-line Driver

PICC18 is the driver invoked from the command line to compile and/or link C programs. PICC18 has the following basic command format:

```
PICC18 [options] files [libraries]
```

It is conventional to supply the options (identified by a leading *dash* "-" or *double dash* "–") before the filenames.

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. *Libraries* are a list of library names, or -L options, see Section 2.4.7. Source files, object files and library files are distinguished by PICC18 solely by the *file type* or *extension*. Recognized file types are listed in Table 2.1. This means, for example, that an assembler file must always have a .as extension (alphabetic case is not important).

PICC18 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,

Table 2.1: PICC18 file types

File Type	Meaning
.c	C source file
.as	Assembler source file
.obj	Relocatable object code file
.lib	Relocatable object library file
.hex	Intel HEX file

all object files resulting from compilation or assembly, or those listed explicitly on the command line, will be linked together with the standard runtime code and libraries and any user-specified libraries. Functions in libraries will be linked into the resulting output file only if referenced in the source code.

Invoking `PICC18` with only object files specified as the file arguments (i.e. no source files) will mean only the link stage is performed. It is typical in Makefiles to use `PICC18` with a `-C` option to compile several source files to object files, then to create the final program by invoking `PICC18` again with only the generated object files and appropriate libraries (and appropriate options). If a .lib output file type is selected, the object files will be stored in a library instead of going through to the final link.

When a HEX file is given on the command line, `PICC18` will invoke the Hexmate utility and will merge the named hex file with the hex file currently being generated. This feature can be useful when, for example, a single hex file is desired which contains a bootloader and application program.

2.1 Long Command Lines

The PICC18 driver is capable of processing command lines exceeding any operating system limitation. To do this, the driver may be passed options via a command file. The command file is read by using the `@` symbol. For example:

```
PICC18 @xyz.cmd
```

2.2 Default Libraries

PICC18 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.

2.3 Standard Runtime Code

PICC18 will automatically generate standard runtime start-up code appropriate for the processor and options selected unless you have specified to disable this via the `--RUNTIME` option. If you require any special powerup initialization, you should use the *powerup* routine feature (see Section 3.3.5.4).

2.4 PICC18 Compiler Options

Most aspects of the compilation can be controlled using the command-line driver, PICC18. The driver will configure and execute all required applications, such as the code generator, assembler and linker.

PICC18 recognizes the compiler options listed in the table below. The case of the options is not important, however command shells in UNIX based operating systems are case sensitive when it comes to names of files.

PICC18 Command-line Options

Option	**Meaning**
-C	Compile to object files only
-D*macro*	Define preprocessor macro
-E+*file*	Redirect and optionally append errors to a file
-G*file*	Generate source-level debugging information
-I*path*	Specify a directory pathname for include files
-L*library*	Specify a library to be scanned by the linker
-L-*option*	Specify *-option* to be passed directly to the linker
-M*file*	Request generation of a MAP file
-N*size*	Specify identifier length
-O*file*	Output file name
-P	Preprocess assembler files
-Q	Specify quiet mode
-S	Compile to assembler source files only
-U*symbol*	Undefine a predefined preprocessor symbol
-V	Verbose: display compiler pass command lines
-X	Eliminate local symbols from symbol table
--ASMLIST	Generate assembler .LST file for each compilation
--CALLGRAPH=*type*	Control the level of information displayed in the call graph
--CHAR=*type*	Make the default char signed or unsigned
--CHIP=*processor*	Selects which processor to compile for
--CHIPINFO	Displays a list of supported processors
--CODEOFFSET=*address*	Offset program code to address
--CP=*size*	Set size of pointers to code space
--CR=*file*	Generate cross-reference listing
--DEBUGGER=*type*	Select the debugger that will be used
continued...	

PICC18 Command-line Options

Option	**Meaning**
--DOUBLE=*type*	Selects size/kind of double types
--EMI=*type*	Select the type of external memory interface used
--ERRATA=*type*	Add or remove specific software workarounds for silicon errata issues.
--ERRFORMAT<=*format*>	Format error message strings to the given style
--ERRORS=*number*	Sets the maximun number of errors displayed
--FILL=*opcode*	Specify a hexadecimal opcode to program in all unused program memory locations.
--GETOPTION=*app*,*file*	Get the command line options for the named application
--HELP<=option>	Display the compiler's command line options
--IDE=*ide*	Configure the compiler for use by the named IDE
--LANG=*language*	Specify language for compiler messages
--MAPFILE<=*file*>	Generates a map file
--MEMMAP=*file*	Display memory summary information for the map file
--MSGFORMAT<=*format*>	Format general message strings to the given style
--NODEL	Do not remove temporary files generated by the compiler
--NOEXEC	Go through the motions of compiling without actually compiling
--OPT<=*type*>	Enable general compiler optimizations
--OUTDIR	Specify output files directory
--OUTPUT=*type*	Generate output file type
--PRE	Produce preprocessed source files
--PROTO	Generate function prototype information
--RAM=lo-hi<,lo-hi,...>	Specify and/or reserve RAM ranges
--ROM=lo-hi<,lo-hi,...>	Specify and/or reserve ROM ranges
--RUNTIME=*type*	Configure the C runtime libraries to the specified type
--SCANDEP	Generate file dependency ".DEP files"
--SERIAL=*code*@*address*	Store this hexadecimal code at an address in program memory
--SETOPTION=*app*,*file*	Set the command line options for the named application
--SETUP=argument	Setup the product
continued...	

PICC18 Command-line Options

Option	Meaning
`--STRICT`	Enable strict ANSI keyword conformance
`--SUMMARY=`*`type`*	Selects the type of memory summary output
`--VER`	Display the compiler's version number
`--WARN=`*`level`*	Set the compiler's warning level
`--WARNFORMAT=`*`format`*	Format warning message strings to given style

All single letter options are identified by a leading *dash* character, "-", e.g. `-C`. Some single letter options specify an additional data field which follows the option name immediately and without any whitespace, e.g. `-Ddebug`.

Multi-letter, or word, options have two leading *dash* characters, e.g. `--ASMLIST`. (Because of the double *dash*, you can determine that the option `--ASMLIST`, for example, is not a `-A` option followed by the argument `SMLIST`.) Some of these options define suboptions which typically appear as a *comma*-separated list following an *equal* character, `=`, e.g. `--OUTPUT=hex,cof`. The exact format of the options varies and are described in detail in the following sections.

Some commonly used suboptions include `default`, which represents the default specification that would be used if this option was absent altogether; `all`, which indicates that all of the available suboptions should be enabled as if they had each been listed; and `none`, which indicates that all suboptions should be disabled. Some suboptions may be prefixed with a plus character, `+`, to indicate that they are in addition to the other suboptions present, or a minus character "-", to indicate that they should be excluded. In the following sections, *angle brackets,* < >, are used to indicate optional parts of the command.

2.4.1 `-B`*`model`*: Select Memory Model

The `-B`*`model`* option tells PICC18 that this build is configured for memory model, *`model`*. The memory models available for selection are *`s`* (*small*) and *`l`* (*large*). See Section 3.6.3 for details on the differences between each of these memory models. If unspecified the default memory model selection is *large*, which is the same as specifying `-Bl`.

2.4.2 `-C`: Compile to Object File

The `-C` option is used to halt compilation after generating a relocatable 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. The object files will be placed in the directory in which PICC18 was invoked, to handle situations where source files are

located in read-only directories. To compile three source files `main.c`, `module1.c` and `asmcode.as` to object files you could use a command similar to:

```
PICC18 --CHIP=18F242 -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 an *Intel* HEX file using the command:

```
PICC18 --CHIP=18F242 main.obj module1.obj asmcode.obj
```

2.4.3 -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
```

placed at the top of each module compiled using this option, or `-Dmacro=`*text* which is equivalent to:

```
#define  macro  text
```

where ***text*** is the textual substitution required. Thus, the command:

```
PICC18 --CHIP=18F242 -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
```

2.4.4 -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, PICC18 allows an error listing filename 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:

```
PICC18 --CHIP=18F242 -Ex.err x.c
```

The -E option also allows errors to be appended to an existing file by specifying an *addition* character, +, at the start of the error filename, for example:

```
PICC18 --CHIP=18F242 -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:

```
PICC18 --CHIP=18F242 -Eproject.err -O -C main.c
PICC18 --CHIP=18F242 -E+project.err -O -C part1.c
PICC18 --CHIP=18F242 -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
```

2.4.5 -G*file*: Generate Source-level Symbol File

The -G option generates a *source-level symbol file* (i.e. a file which allows tools to determine which line of source code is associated with machine code instructions, and determine which source-level variable names correspond with areas of memory, etc.) for use with supported debuggers and simulators such as *HI-TIDE*® and *MPLAB*®. If no filename is given, the symbol file will have the same base name as the first source or object file specified on the command line, and an extension of .sym. For example the option -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:

```
PICC18 --CHIP=18F242 -G -C test.c
PICC18 --CHIP=18F242 -C module1.c
PICC18 --CHIP=18F242 -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.

The `--IDE` option will typically enable the `-G` option.

2.4.6 `-Ipath`: 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 is always searched even if no `-I` option is present. The default search path is searched after any user-specified directories have been searched. For example:

```
PICC18 --CHIP=18F242 -C -Ic:\include -Id:\myapp\include test.c
```

will search the directories `c:\include` and `d:\myapp\include` for any header files included into the source code, then search the default include directory (the include directory where the compiler was installed).

It is strongly advised not to use `-I` to add the compiler's default include path, not only because it is unnecessary but in the event that the build tool changes, the path specified here will be searched prior to searching the new compiler's default path.

This option has no effect for files that are included into assembly source using the `INCLUDE` directive.

2.4.7 `-Llibrary`: 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.

The argument to -L is a library keyword to which the prefix `pic8`; numbers and letters representing the build configuration and applicable errata workarounds; and the suffix `.lib` are added. Thus the option `-LL` when compiling for a 18F452 will, for example, scan the library `pic86l-l.lib` and the option `-Lxx` will scan a library called `pic86c-xx.lib`. All libraries must be located in the LIB subdirectory of the compiler installation directory. As indicated, the argument to the `-L` option is *not* a complete library filename.

If you wish the linker to scan libraries whose names do not follow the above naming convention or whose locations are not in the LIB subdirectory, simply include the libraries' names on the command line along with your source files. Alternatively, the linker may be invoked directly allowing the user to manually specify all the libraries to be scanned.

2.4.8 `-L-option`: Adjust Linker Options Directly

The `-L` option can also be used to specify an extra "-" option which will be passed directly to the linker by PICC18. If `-L` is followed immediately by any text starting with a *dash* character "-", the text will be passed directly to the linker without being interpreted by PICC18. For example, if the option `-L-FOO` is specified, the `-FOO` option will be passed on to the linker when it is invoked.

The `-L` option is especially useful when linking code which contains extra program sections (or *psects*), as may be the case if the program contains C code which makes use of the `#pragma psect` directive or assembler code which contains user-defined psects. See Section 3.12.3.3 for more information. If this `-L` option did not exist, it would be necessary to invoke the linker manually to link code which uses the extra psects.

One commonly used linker option is `-N`, which sorts the symbol table in the map file by address, rather than by name. This would be passed to PICC18 as the option `-L-N`.

The `-L` option can also be used to replace default linker options. If the string starting from the first character after the -L up to the = character matches a default option, then the default option is replaced by the option specified. For example, `-L-preset=100h` will inform the linker to replace the default option that places the `reset` psect to be one that places the psect at the address 100h. The default option that you are replacing must contain an *equal* character.

2.4.9 `-Mfile`: Generate Map File

The `-M` option is used to request the generation of a map file. The map is generated by the linker and includes information about where objects are located in memory. If no filename is specified, then the name of the map file will have the same name as the first file listed on the command line, with the extension `.map`.

2.4.10 `-Nsize`: Identifier Length

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

2.4.11 `-Ofile`: Specify Output File

This option allows the name of the output file(s) to be specified. If no `-O` option is given, the output file(s) will be named after the first source or object file on the command line. The files controlled are any produced by the linker or applications run subsequent to that, e.g. `CROMWELL`. So for instance the HEX file, map file and SYM file are all controlled by the `-O` option.

The `-O` option can also change the directory in which the output file is located by including the required path before the filename, e.g. `-Oc:\project\output\first.hex`. This will then also specify the output directory for any files produced by the linker or subsequently run applications.

2.4.12 -P: Preprocess Assembly Files

The -P option causes the assembler files to be preprocessed before they are assembled thus allowing the use of preprocessor directives, such as #include, with assembler code. By default, assembler files are not preprocessed.

2.4.13 -Q: Quiet Mode

This option places the compiler in a *quiet mode* which suppresses the HI-TECH Software copyright notice from being displayed.

2.4.14 -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:

```
PICC18 --CHIP=18F242 -S test.c
```

will produce an assembler file called test.as which contains the code generated from test.c. This option is particularly useful for checking function calling conventions and signature values when attempting to write external assembly language routines. The file produced by this option differs to that produced by the --ASMLIST option in that it does not contain op-codes or addresses and it may be used as a source file and subsequently passed to the assembler to be assembled.

2.4.15 -Umacro: Undefine a Macro

The -U option, the inverse of the -D option, is used to *undefine* predefined macros. This option takes the form -Umacro. The option, -Udraft, for example, is equivalent to:

```
#undef  draft
```

placed at the top of each module compiled using this option.

2.4.16 -V: Verbose Compile

The -V is the *verbose* option. The compiler will display the full command lines used to invoke each of the compiler applications or compiler passes. This option may be useful for determining the exact linker options if you need to directly invoke the HLINK command.

2.4.17 `-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.

2.4.18 `--ASMLIST`: Generate Assembler .LST Files

The `--ASMLIST` option tells PICC18 to generate an *assembler listing file* for each module being compiled. The list file shows both the original C code, and the generated assembler code and the corresponding binary op-codes. The listing file will have the same name as the source file, and a file type (extension) of `.lst`. Provided the link stage has successfully concluded, the listing file will be updated by the linker so that it contains absolute addresses and symbol values. Thus you may use the assembler listing file to determine the position of, and exact op codes corresponding to, instructions.

2.4.19 `--CALLGRAPH=type`: Control level of information displayed in call graph

This option allows control over the type of call graph produced in the map file. Allowable suboptions include: `none`, to specify that no call graph should be produced; and `full` to indicate that the full call graph be displayed in the map file. In addition, the suboption `std` can be specified to indicate that a shorter form, without redundant information relating to ARG functions be produced; or `crit`, to inidcate that only critical path information be displayed in the call graph.

2.4.20 `--CHAR=type`: Make Char Type Signed or Unsigned

Unless this option is used, the default behaviour of the compiler is to make all undesignated character types, `unsigned char`, unless explicitly declared or cast to `signed char`. If `--CHAR=signed` is used, the default char type will become `signed char`.

The range of a `signed` character type is -128 to +127 and the range of similar `unsigned` objects is 0 to 255.

2.4.21 `--CHECKSUM=start-end@address<,specs>`: Calculate, store and verify a checksum

When this option is used, a checksum is calculated over the given range and the result stored at the address specified. Additional specifications are permitted to customize the type of checksum that is used. Each specification is entered as a comma separated *spec=value* pair, details of which are in table 2.3.

Table 2.3: Additional --checksum specifications

Specification	Purpose	Possible values	Default
algorithm	Select a checksum algorithm	See table 5.10	1
offset	Add a value to result	0 - 0xFFFFFFFF	0
width	Byte width of result	1 to 4. (Little-endian: -1 to -4)	-2

By default, if the destination of the checksum is an address in program memory the generated startup routine will include a built-in verification of this checksum. If this is not desired, this test can be disabled with the option --runtime=-checksum. For reliable runtime verification of the checksum, all unused locations within the checksum range should contain a known value. This option will prefill all unused locations within the range of calculation with value FFh. This prefilling will not be performed if the --FILL option is also used to allow a different code to be used for prefilling.

2.4.22 --CHIP=*processor*: Define Processor

This option defines the processor which is being used. To see a list of supported processors that can be used with this option, use the --CHIPINFO option.

2.4.23 --CHIPINFO: Display List of Supported Processors

The --CHIPINFO option simply displays a list of processors the compiler supports. The names listed are those chips defined in the chipinfo file and which may be used with the --chip option or refer to Appendix C of this manual.

2.4.24 --CODEOFFSET: Offset Program Code to Address

In some circumstances, such as bootloaders, it is necessary to shift the program image to an alternate address. This option is used to specify a base address for the program code image. With this option, all code psects (including interrupt vectors and constant data) that the linker would ordinarily control the location of, will be adjusted.

2.4.25 --CP=*size*: Set the Size of Pointers to Code Space

Pointers to code/functions, pointers to const, pointers to far and pointers to void can be set to either 16 bits wide or 24 bits wide. If this option is not specified the default setting is 16 bits. It is sufficient to use --CP=16 for chips that have less than 64K of program memory, or if only the first 64K of program memory will be dereferenced with a pointer. If a particular program uses pointers

to program space beyond the 64K boundary that it is necessary to enable 24 bit code pointers. This is also applicable to projects and PIC18 devices which utilize the external memory interface, as it is more than likely that the additional memory will be mapped beyond the 64K boundary.
It will also be necessary to enable 24 bit pointers to reference an object in code space which is located at an address that would also be valid in the general-purpose data space. For more information on pointers refer to Section 3.4.12.
Note that it is critical that the compilation stages for all source files in a project and the link stage apply consistent code pointer sizes.

2.4.26 `--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 `--CR=test.crf`, PICC18 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 PICC18 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:

```
PICC18 --CHIP=18F242 --CR=main.crf main.c module1.c nvram.c
```

2.4.27 `--DEBUGGER=type`: Select Debugger Type

This option specifies to the compiler that the code being built must be runnable on the nominated debugger. PICC18 supports the Microchip ICD2 debugger and using this option will configure the compiler to conform to the requirements of the debugger (reserving memory addresses, etc.). For example:

```
PICC18 --CHIP=18F242 --DEBUGGER=icd2 main.c
```

2.4.28 `--DOUBLE=type`: Select kind of Double Types

This option allows the kind of double types to be selected. By default the compiler will choose the truncated IEEE754 24-bit implementation for double types. With this option, this can be changed to 32-bits. A fast implementation, at the cost of code size, is also available.

Table 2.4: Supported Double Types

Suboption	Type
24	Truncated IEEE754 24-bit doubles
32	IEEE754 32-bit doubles
fast32	Faster implementation of 32-bit doubles

2.4.29 --EMI=*type*: Select operating mode of the external memory interface (EMI)

Those PIC18 devices which can interface with an external memory are capable of operating in several modes. The mode selected is determined by the type of memory available and the connection method used. The interface can operate in 16-bit modes; *word write* and *byte select* mode or in an 8-bit mode: *byte write* mode. Valid types that can be specified to this option are: `wordwrite`, `byteselect` or `bytewrite`. Which mode is selected will affect the code generated when writing to the external data. In word write mode, dummy reads and writes may be added to ensure that an even number of bytes are always written. In byte select or byte write modes dummy reads and writes are not generated and can result in more efficient code. Note that this option does not in any way pre-configure the device for operation in the selected mode.

2.4.30 --ERRATA=*type*: Specify to add or remove specific errata workarounds

This option allows specification of the types of software workarounds to apply in order to overcome documented silicon errata issues. The chip configuration file nominates a default set of errata issues that apply to each device. To compile for an ideal chip, that is, apply no additional workarounds use `--ERRATA=none`.

Errata name	Errata description	Workaround details
4000	Execution of some flow control operations may yeild unexpected results when certain instructions vector code execution across the 4000h address boundary.	A continuous block of program code is not allowed to grow over the 4000h address boundary. Additional `NOP` instructions are inserted at prescribed locations.
LFSR	Using the `LFSR` instruction to load a value into a specified FSR register may also corrupt a RAM location.	The compiler will load FSR registers without using the `LFSR` instruction.
DAW	The `DAW` instruction may improperly clear the `CARRY` bit (`STATUS<0>`) when executed.	The compiler is not affected by this issue.

MINUS40	Table read operations above the user program space (>1FFFFFh) may yield erroneous results at the extreme low end of the device's rated temperature range (-40°C).	Affected library sources *config.c*, *idloc.c* and *devread.c* employ additional NOP instructions at prescribed locations.
EEPROMRD	When reading EEPROM, the contents of the EEDATA register may become corrupted in the second instruction cycle after setting the RD bit (EECON1<0>).	The EEPROM_READ macro and eeprom_read library function read EEDATA immeadiately.
EEPROMADR	The result returned from an EEPROM read operation can be corrupted if the RD bit is set immediately following the loading of the EEADR register.	The compiler is not affected by this issue.
EEPROMLVD	Writes to EEPROM memory may not succeed if the internal voltage reference is not set.	No workaround applied
FLASHLVD	Writes to program memory may not succeed if the internal voltage reference is not set.	No workaround applied
RESET	A GOTO instruction placed at the reset vector may not execute.	Additional NOP instruction inserted at reset vector if following instruction is GOTO.
FASTINTS	If a high-priority interrupt occurs during a two-cycle instruction which modifies WREG, BSR or STATUS, the fast-interrupt return mechanism (via shadow registers) will restore the value held by the register before the instruction.	Additional code reloads the shadow registers with the correct values of WREG, STATUS and BSR.
BSR15	Peripheral flags may be erroneously affected if the BSR register holds the value 15, and an instruction is executed that holds the value C9h in its 8 least significant bits.	A warning will be issued if the instruction MOVLB 15 instruction is detected in the execuble code.
TBLWTINT	If a peripheral interrupt occurs during a TBLWT operation, data can be corrupted.	Library routine *flash_write()* will temporarily disable all applicable interrupt-enable bits before execution of a TBLWT instruction.

FW4000	Self write operations initiated from and acting upon a range within the same side of the 4000h boundary may fail based on sequences of instructions executed following the write.	No workaround applied

2.4.31 --ERRFORMAT=*format*: Define Format for Compiler Messages

If the --ERRFORMAT option is not used, the default behaviour of the compiler is to display any 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: _PA = xFF;
                ^ (192) undefined identifier: xFF
```

This standard format is perfectly acceptable to a person reading the error output, but is not generally usable with environments which support compiler error handling. The following sections indicate how this option may be used in such situations.

This section is also applicable to the --WARNFORMAT and --MSGFORMAT options which adjust the format of warning and advisory messages, respectively.

2.4.31.1 Using the Format Options

Using the these options instructs the compiler to generate error, warning and advisory messages in a format which is acceptable to some text editors and development environments.

If the same source code as used in the example above were compiled using the --ERRFORMAT option, the error output would be:

```
x.c 4: (192) undefined identifier: xFF
```

indicating that the error number 192 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.

Table 2.6: Error format specifiers

Specifier	Expands To
%f	Filename
%l	Line number
%c	Column number
%s	Error string
%a	Application name
%n	Message number

2.4.31.2 Modifying the Standard Format

If the message format does not meet your editor's requirement, you can redefine its format by either using the --ERRFORMAT, --WARNFORMAT or --MSGFORMAT option or by setting the environment variables: HTC_ERR_FORMAT, HTC_WARN_FORMAT or HTC_MSG_FORMAT. These options are in the form of a printf-style string in which you can use the specifiers shown in Table 2.6. For example:

```
--ERRFORMAT="file %f; line %l; column %c; %s"
```

The column number is relative to the left-most non-space character on the source line.

The environment variables can be set in a similar way, for example setting the environment variables from within DOS can be done with the following DOS commands:

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

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

```
ERROR: parser: file x.c; line 4; column 6; (192) 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. the filename specifier would become %%f.

2.4.32 --ERRORS=*number*: Maximum Number of Errors

This option sets the maximum number of errors each component of the compiler will display before stopping. By default, up to 20 error messages will be displayed.

2.4.33 `--FILL=opcode`: Fill Unused Program Memory

This option allows specification of a hexadecimal opcode that can be used to fill all unused program memory locations with a known code sequence. Multi-byte codes should be entered in little endian byte order.

TUTORIAL

One practical application of this is filling unused space with instructions to could correct the program if it starts executing in memory that is out-of-bounds. In this event, it might be desirable to jump to an error-logging routine or invoke reset. The PIC18 opcode for the `RESET` instruction is `00FFh`. To fill all unused program memory locations with this instruction, use the option `--FILL=FF00` (Note that byte order is little endian).

If the fill value is only to be applied to a restricted address range, the restriction can be specified by using `--FILL=opcode@start_address-end_address`. This facility also makes it possible to allow a fill value to be applied to address ranges outside of program memory (as addressed in the hex file), for example EEPROM. If an address restriction is not specified, the fill value will be applied to all of the device's program memory.

2.4.34 `--GETOPTION=app,file`: Get Command-line Options

This option is used to retrieve the command line options which are used for named compiler application. The options are then saved into the given file. This option is not required for most projects.

2.4.35 `--HELP<=option>`: Display Help

The `--HELP` option displays information on the PICC18 compiler options. To find out more about a particular option, use the option's name as a parameter. For example:

```
PICC18 --help=warn
```

This will display more detailed information about the `--WARN` option.

2.4.36 `--IDE=type:` Specify the IDE being used

This option is used to automatically configure the compiler for use by the named Integrated Development Environment (IDE). The supported IDE's are shown in Table 2.7.

Table 2.7: Supported IDEs

Suboption	IDE
hitide	HI-TECH Software's HI-TIDE
mplab	Microchip's MPLAB

Table 2.8: Supported languages

Suboption	Language
en, english	English
fr, french,francais	French
de, german, deutsch	German

2.4.37 --LANG=*language*: Specify the Language for Messages

This option allows the compiler to be configured to produce error, warning and some advisory messages in languages other than English. English is the default language and some messages are only ever printed in English regardless of the language specified with this option.

Table 2.8 shows those langauges currently supported.

2.4.38 --MEMMAP=*file*: Display Memory Map

This option will display a memory map for the specified map file. This option is seldom required, but would be useful if the linker is being driven explicitly, i.e. instead of in the normal way through the driver. This command would display the memory summary which is normally produced at the end of compilation by the driver.

2.4.39 --MSGFORMAT=*format*: Set Advisory Message Format

This option sets the format of advisory messages produced by the compiler. See Section 2.4.31 for full information.

2.4.40 --NOEXEC: Don't Execute Compiler

The --NOEXEC option causes the compiler to go through all the compilation steps, but without actually performing any compilation or producing any output. This may be useful when used in conjunction with the -V (verbose) option in order to see all of the command lines the compiler uses to drive the compiler applications.

Table 2.9: Optimization Options

Option name	File format
1..9	Select code generation level 1 through 9
asm	Select assembler optimizations
debug	Favor accurate debugging over optimization
all	Enable all compiler optimizations
none	Do not use any compiler optimziations

Table 2.10: Output file formats

Option name	File format
lib	Library File
intel	*Intel* HEX
tek	Tektronic
aahex	*American Automation* symbolic HEX file
mot	*Motorola* S19 HEX file
ubrof	UBROF format
bin	Binary file
mcof	Microchip PIC COFF
cof	Common Object File Format
cod	Bytecraft COD file format
elf	ELF/DWARF file format

2.4.41 `--OPT<=type>`: Invoke Compiler Optimizations

The `--OPT` option allows control of all the compiler optimizers. By default, without this option, all optimizations are enabled. The options `--OPT` or `--OPT=all` also enable all optimizations. Optimizations may be disabled by using `--OPT=none`, or individual optimizers may be controlled, e.g. `--OPT=asm` will only enable the assembler optimizer. Table 2.9 lists the available optimization types.

2.4.42 `--OUTPUT=type`: Specify Output File Type

This option allows the type of the output file to be specified. If no `--OUTPUT` option is specified, the output file's name will be derived from the first source or object file specified on the command line. The available output file formats are shown in Table 2.10.

2.4.43 --PRE: Produce Preprocessed Source Code

The --PRE option is used to generate preprocessed C source files with an extension .pre. This may be useful to ensure that preprocessor macros have expanded 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.

2.4.44 --PROTO: Generate Prototypes

The --PROTO option 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:

```
PICC18 --CHIP=18F242 --PROTO test.c
```

PICC18 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 */
```

2.4.45 `--RAM=lo-hi,<lo-hi,...>`: Specify Additional RAM Ranges

This option is used to specify memory, in addition to any RAM specified in the chipinfo file, which should be treated as available RAM space. Strictly speaking, this option specifies the areas of memory that may be used by writable (RAM-based) objects, and not necessarily those areas of memory which contain physical RAM. The output that will be placed in the ranges specified by this option are typically variables that a program defines.

Some chips have an area of RAM that can be remapped in terms of its location in the memory space. This, along with any fixed RAM memory defined in the chipinfo file, are grouped an made available for RAM-based objects.

For example, to specify an additional range of memory to that present on-chip, use:

```
--RAM=default,+100-1ff
```

for example. To only use an external range and ignore any on-chip memory, use:

```
--RAM=0-ff
```

This option may also be used to reserve memory ranges already defined as on-chip memory in the chipinfo file. To do this supply a range prefixed with a *minus* character, -, for example:

```
--RAM=default,-100-103
```

will use all the defined on-chip memory, but not use the addresses in the range from 100h to 103h for allocation of RAM objects.
Sections of peripheral memory can sometimes be referred to with a tag, for example `--RAM=default,+USBRAM`. For more information see Section 3.2.5.

2.4.46 --ROM=*lo-hi,<lo-hi,...>|tag:* Specify Additional ROM Ranges

This option is used to specify memory, in addition to any ROM specified in the chip configuration file, which should be treated as available ROM space. Strictly speaking, this option specifies the areas of memory that may be used by read-only (ROM-based) objects, and not necessarily those areas of memory which contain physical ROM. The output that will be placed in the ranges specified by this option are typically executable code and any data variables that are qualified as `const`.

When producing code that may be downloaded into a system via a bootloader the destination memory may indeed be some sort of (volatile) RAM. To only use on-chip ROM memory, this option is not required. For example, to specify an additional range of memory to that on-chip, use:

```
--ROM=default,+100-2ff
```

for example. To only use an external range and ignore any on-chip memory, use:

```
--ROM=100-2ff
```

This option may also be used to reserve memory ranges already defined as on-chip memory in the chip configuration file. To do this supply a range prefixed with a *minus* character, -, for example:

```
--ROM=default,-100-1ff
```

will use all the defined on-chip memory, but not use the addresses in the range from 100h to 1ffh for allocation of ROM objects.
Sections of peripheral memory can sometimes be referred to with a tag, for example `--ROM=default,-BOOTROM`. For more information see Section 3.2.5.

2.4.47 --RUNTIME=*type*: Specify Runtime Environment

The `--RUNTIME` option is used to control what is included as part of the runtime environment. The runtime environment encapsulates any code that is present at runtime which has not been defined by the user, instead supplied by the compiler, typically as library code.

All runtime features are enabled by default and this option is not required for normal compilation. The usable suboptions include those shown in Table 2.11.

2.4.48 --SCANDEP: Scan for Dependencies

When this option is used, a `.dep` (dependency) file is generated. The dependency file lists those files on which the source file is dependant. Dependencies result when one file is `#included` into another.

Table 2.11: Runtime environment suboptions

Suboption	Controls	On (+) implies
`init`	The code present in the startup module that copies the `idata`, `ibigdata` and `ifardata` psects' ROM-image to RAM.	The `idata`, `ibigdata` and `ifardata` psects' ROM image is copied into RAM.
`checksum`	Built-in verification of `--checksum` specifications.	If PICC18's `--checksum` option is also used, the generated startup routines will include verification against the stored checksum. The checksum result is assumed to be stored in program memory. If the verification fails, library function `__checksum_failed()` will be called. This function is expected to be customized to suit the final application.
`clib`	The inclusion of library files into the output code by the linker.	Library files are linked into the output.
`clear`	The code present in the startup module that clears the `bss`, `bigbss`, `rbss` and `farbss` psects.	The `bss`, `bigbss`, `rbss` and `farbss` psects are cleared.
`download`	Conditioning of the Intel hex file for use with bootloaders.	Data records in the Intel hex file are padded out to 16 byte lengths and will align on 16 byte boundaries. Startup code will not assume reset values in certain registers.
`keep`	Whether the start-up module source file is deleted after compilation.	The start-up module is not deleted.
`ramtest`	Perform a RAM integrity test before clearing/initializing memory.	Generated start up code will loop through every location of general purpose RAM and call library routine, `__ram_cell_test`. Variables qualified as *persistent* are backed up during cell testing.

2.4.49 --SERIAL=*hexcode*@*address*: Store a Value at this Program Memory Address

This option allows a hexadecimal code to be stored at a particular address in program memory. A typical application for this option might be to position a serial number in program memory. The byte-width of data to store is determined by the byte-width of the hexcode parameter in the option. For example to store a one byte value, zero, at program memory address 1000h, use `--SERIAL=00@1000`. To store the same value as a four byte quantity use `--SERIAL=00000000@1000`. This option is functionally identical to the corresponding hexmate option. For more detailed information and advanced controls that can be used with this option, refer to Section 5.15.1.14 of this manual.

2.4.50 --SETOPTION=*app*,*file*: Set The Command-line Options for Application

This option is used to supply alternative command line options for the named application when compiling. This option is not required for most projects.

2.4.51 --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 the `persistent` type qualifier). If the `--STRICT` option is used, these keywords are changed to include two *underscore* characters at the beginning of the keyword (e.g. `__persistent`) 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>`).

2.4.52 --SUMMARY=*type*: Select Memory Summary Output Type

Use this option to select the type of memory summary that is displayed after compilation. By default, or if the `mem` suboption is selected, a memory summary is shown. This shows the memory usage for all available linker classes.

A psect summary may be shown by enabling the `psect` suboption. This shows individual psects, after they have been grouped by the linker, and the memory ranges they cover. Table 2.12 shows what summary types are available.

2.4.53 --VER: Display The Compiler's Version Information

The `--VER` option will display what version of the compiler is running.

Table 2.12: Memory Summary Suboptions

Suboption	Controls	On (+) implies
psect	Summary of psect usage.	A summary of psect names and the addresses they were linked at will be shown.
mem	General summary of memory used.	A general summary of memories used will be shown.
hex	Summary of address used within the hex file.	A summary of addresses and hex files which make up the final output file will be shown.
file	Whether summary information is shown on the screen or shown and saved to a file.	Summary information will be shown on screen and saved to a file.

2.4.54 --WARN=*level*: Set Warning Level

The --WARN option is used to set the compiler warning level. Allowable warning levels range from -9 to 9. The warning level determines how pedantic the compiler is about dubious type conversions and constructs. The default warning level --WARN=0 will allow all normal warning messages. Warning level --WARN=1 will suppress the message Func() declared implicit int. --WARN=3 is recommended for compiling code originally written with other, less strict, compilers. --WARN=9 will suppress all warning messages. Negative warning levels --WARN=-1, --WARN=-2 and --WARN=-3 enable special warning messages including compile-time checking of arguments to printf() against the format string specified.

Use this option with care as some warning messages indicate code that is likely to fail during execution, or compromise portability.

2.4.55 --WARNFORMAT=*format*: Set Warning Message Format

This option sets the format of warning messages produced by the compiler. See Section 2.4.31 for full information.

Chapter 3

C Language Features

HI-TECH PICC-18 STD 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 language features which are specific to the *Microchip* PIC 18 family of processors.

3.1 ANSI Standard Issues

3.1.1 Divergence from the ANSI C Standard

HI-TECH PICC-18 STD diverges from the ANSI C standard in one area: function recursion.

Due to the PIC18's hardware limitations of no easily-usable stack and limited memory, function recursion is unsupported.

3.1.2 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 HI-TECH PICC-18 STD compiler behaves in such situations.

3.2 Processor-related Features

HI-TECH PICC-18 STD has many features which relate directly to the PIC18 family of processors. These are detailed in the following sections.

3.2.1 Processor Support

HI-TECH PICC-18 STD supports the full range of Microchip PIC 18 processors. Additional code-compatible processors may be added by editing the picc-18.ini file in the DAT directory. 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.

3.2.2 Configuration Fuses

The PIC18 processors have several locations which contain the *configuration bits* or *fuses*. These bits may be set using the configuration macro. The macro has the form:

```
__CONFIG(n,x);
```

(there are two leading *underscore* characters) where ***n*** is the configuration register number and ***x*** is the value that is to be the configuration word. The macro is defined in <htc.h> so be sure to include that into each module that uses this macro.

The configuration macro programs the upper and lower half of each register, i.e. it programs 16 bits with each call. Special named quantities are defined in the header file appropriate for the processor you are using to help you enable the required features. To look up the available attributes for any particular chip, consult Appendix **??** of this manual.

For example, to set a PIC18Cxx1 chip to have an RC type oscillator, an 8-bit bus width, the powerup timer disabled, the watchdog timer enabled with a post scale factor of 1:1, and the stack full/underflow reset disabled, the following could be used.

```
#include <htc.h>
__CONFIG(1, RC);
__CONFIG(2, BW8 & PWRTDIS & WDTPS1 & WDTEN);
__CONFIG(4, STVRDIS);
```

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 bits correctly to ensure proper operation of the part when programmed. Consult your PIC18 datasheet for more details.

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

3.2.3 ID Locations

Some PIC18 devices have location 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 <htc.h>
__IDLOC(x);
```

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

```
__IDLOC(15F01);
```

will attempt to fill ID locations which the values: F1H, F5H, FFH, F0H, F1H. The base address of the ID locations is specified by the *idloc* psect which will be automatically assigned as appropriate address based on the type of processor selected.

3.2.4 EEPROM and Flash Runtime Access

The compiler offers several methods of accessing EEPROM and Flash memory. These are described in the following sections.

3.2.4.1 EEPROM Access

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 <htc.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. For example, to write a byte-size value to an address in EEPROM memory using the macro version of EEPROM write:

```
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, **__EEPROMSIZE** predefines the number of bytes of EEPROM available on chip.

3.2.4.2 Flash Access

To copy a block of code/data to an area in flash memory:

```
flash_write(source_pointer, length, dest_pointer);
```

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

```
variable=flash_read(address);
```

3.2.5 Peripheral Memory Ranges

Some devices have sections of memory that are intended for use by its peripherals or the device itself. In these instances it may be beneficial to exclude these ranges from the compiler's available resources. Conversely, if you can be sure that a particular resource will not be used by the device (and it is safe to do so), you may benefit by making the resource available to the compiler. Typical examples of these types of memory resources are on-chip debug facilities, boot sectors in program space, DMA and USB data buffers. These memory ranges can be included or excluded from the memory ranges available to the compiler via the driver's –RAM or –ROM options.

TUTORIAL

Consider this example. A device has RAM in the address range from 0 to 7FFh, however the range 400h to 7FFh is dual-port RAM shared with the device's onboard USB module. This memory range by default is unavailable to the compiler. If a program being built for this device is running short on RAM and doesn't use the USB facility, it would make sense to avail this memory range to the program. One way that this may be done is to define and add the range using the driver option `--RAM=default,+400-7ff`. Alternately, if the driver recognises the resource by name, the option could be modified to say `--RAM=default,+USBRAM`.

Those peripheral ranges that can be referred to by name are detailed in Table 3.1.

Table 3.1: Peripheral memory tags

Peripheral tag	Description	Applicable devices
USBRAM	USB dual-port RAM	Devices that have dual-port USB RAM
BOOTROM	Boot sector in flash memory	Flash memory based devices with a *fixed length* boot sector
BOOT256, BOOT512, BOOT1K, BOOT2K, BOOT4K	Boot sector in flash memory	Flash memory based devices with *variable length* boot sector. Only tags corresponding to valid boot sector sizes for that chip will be available.

3.2.6 Bit Instructions

Wherever possible, HI-TECH PICC-18 STD will attempt to use the PIC18 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.

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

will produce the instruction:

```
bsf _foo,6
```

To set or clear individual bits within integral type, 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);
```

3.2.7 Multi-byte SFRs

Some of the SFRs associated with the PIC18 can be grouped to form multi-byte values, e.g. the TMRxH and TMRxL register together form a 16-bit timer count value. Depending on the device and mode of operation, there may be hardware requirements to read these registers correctly, e.g.

the TMRxL register often must be read before trying to read the TMRxH register to obtain a valid 16-bit result.

Although it is possible to define an absolute non-char C variable to map over such registers, the order in which HI-TECH PICC-18 STD reads the bytes of a multi-byte object varies depending on the context of the variable in an expression, i.e. it may read the most significant byte first, or the least. Thus, it highly recommended that the existing SFR char definitions in the chip header files be used. Each SFR should be accessed directly and in the required order by the programmer's code. This will ensure a much higher degree of portability.

The following code copies the two byte registers into C unsigned variable i for subsequent use.

```
i = TMR0L;
i += TMR0H << 8;
```

3.3 Files

3.3.1 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.

3.3.2 Symbol Files

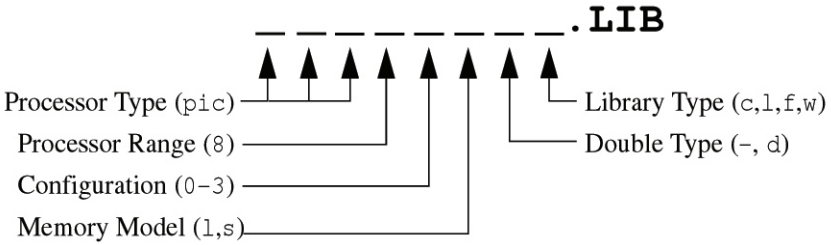
The PICC18 `-G` option tells the compiler to produce several symbol files which can be used by debuggers and simulators to perform symbolic and source-level debugging. Using the `--IDE` option may also enable symbol file generation as well.

The `-G` option produces an absolute symbol files which contain both assembler- and C-level information. This file is produced by the linker after the linking process has ben completed. If no symbol filename is specified, a default filename of *`file`*`.sym` will be used, 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:

```
PICC18 --CHIP=18F242 -Gtest.sym test.c init.c
```

This option will also generate other symbol files for each module compiled. These files are produced by the code generator and do not contain absolute address. These files have the extension `.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`.

Figure 3.1: Library naming convention



3.3.3 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 PICC18 command is to produce *Microchip* COFF and *Intel* HEX output. If no output filename or type is specified, PICC18 will produce a *Microchip* COFF and *Intel* HEX file with the same base name as the first source or object file specified on the command line. Table 2.10 shows the output format options available with PICC18. The *File Type* column lists the filename extension which will be used for the output file.

In addition to the options shown, the `-O` option may be used to request generation of binary or UBROF files. If you use the -O option to specify an output filename with a .bin type, for example `-Otest.bin`, PICC18 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`.

3.3.4 Library files

3.3.4.1 Standard Libraries

PICC-18 includes a number of standard libraries, each with the range of functions described in Appendix A.

Figure 3.1 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 `8` for the PIC18 family.
- Configuration is a digit, bit 0 of which is either `1` for 24-bit wide program space pointers; otherwise `0`. Bit 1 is `0` to disallow the use of the LFSR instruction; `1` to allow this instruction.

Table 3.2: Printf functionality

Library type	Supported formats
c (standard)	%d, %u, %o, %x, %X, %s, %c
l (+long)	c library plus %ld, %lx, %lX, %lo
f (+float)	l library plus %f, %e
w (full featured)	f library plus %E, %g, %G, %i, %p

- Memory Model is either *l* for large or *s* for small model.
- Double Type is – for 24-bit doubles, and d for 32-bit doubles.
- Library Type is c for standard library. Types l, f and w denote enhanced printf libraries which are explained in section 3.3.4.2.

3.3.4.2 Printf Libraries

The standard libraries contain a compact, low-featured implementation of printf (and related functions). The standard printf supports processing of the simplest format types. If an application requires to process more complexed format types, enhanced routines can be linked from specific printf libraries. Note that increasing the functionality of printf will also increase the amount of program space consumed by these library routines. The capabilities of each printf library is explained in Table 3.2. The printf libraries are simply a suppliment to the standard libraries so the naming convention is the same.

To nominate selection of a particular printf library, use the PICC18 option, -Lx as descriped in section 2.4.7. It is not neccessary to nominate selection of the c (standard) library as this file will automatically included by PICC18, unless explicitly excluded by the --RUNTIME option, as described in section 2.4.47. For more details on the printf functions, see the library function documentation in Appendix A.

3.3.4.3 Peripheral Libraries

To accompany the standard libraries, PICC-18 automatically includes an additional set of peripheral dependant libraries. The functions contained in these libraries are those which have particular a reliance on hardware or a device's special function registers, for example accesses to EEPROM memory. Figure 3.2 illustrates the naming convention used for the peripheral libraries.

PICC-18 peripheral libraries are denoted by the name pic8*x*--p.lib, where *x* is a 3-bit bitmask that representing the combinations of peripheral characteristics and errata workarounds applicable to this library. An explanation of the bits within the bitmask is available in Table 3.3.

Figure 3.2: Peripheral Library Nomenclature

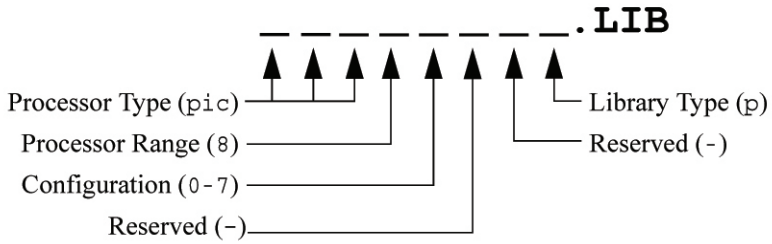


Table 3.3: Peripheral Library Configuration Bitmask

Bit index	Attribute
0	Device implementing EEDATA errata workaround for EEPROM reads
1	Device has more than 256 bytes of EEPROM on board
2	Device implementing 4000h boundary errata workaround

3.3.4.4 Program Memory Libraries

Some devices in the PIC18 architecture have the ability to erase and re-write their own program memory at runtime. There are subtle differences between the mechanisms that a device may use when operating on its own program memory. The block sizes of a program memory erase or write operation may vary from device to device. Silicon errata issues may also dictate minor changes need be applied to code sequences that enact a program memory access operation. For this reason a specific set of libraries are provided to cater for the varying requirements of each device. Figure 3.3 illustrates the naming convention used for these program memory libraries.

These libraries are denoted by the name `picxxypm.lib`, where *xx* is an 8-bit hexadecimal code used to identify the block sizes of a device's program memory write and erase operations.

Figure 3.3: Program Memory Library Nomenclature

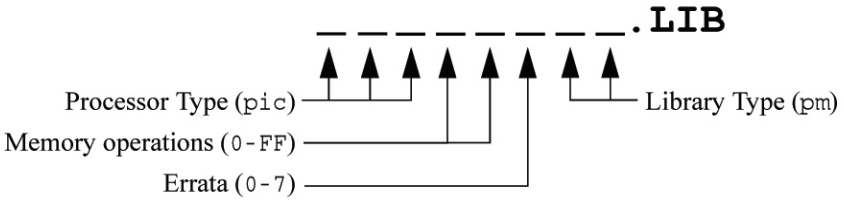


Table 3.4: Program Memory Errata Bitmask

Bit index	Errata and build characteristic
0	Library implements additional NOPs when reading beyond program space at -40C
1	Library implements 4000h boundary errata workaround
2	Library uses 24-bit pointers to code space

The value *y* is a 3-bit bitmask which represents the combinations of errata workarounds and build characteristics applied to this library. An explanation of the bits within the *y* bitmask is available in Table 3.4.

3.3.5 Runtime 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 of its function `main()`. It is the job of the *runtime startup* code to perform these tasks.

Traditionally, runtime startup code is a generic, precompiled routine which is always linked into a user's program. Even if a user's program does not need all aspects of the runtime startup code, redundant code is linked in which, albeit not harmful, takes up memory and slows execution. For example, if a program does not use any uninitialized variables, then no routine is required to clear the bss psects.

HI-TECH PICC-18 STD differs from other compilers by using a novel method to determine exactly what runtime startup code is required and links this into the program automatically. It does this by performing an additional link step which does not produce any usable output, but which can be used to determine the requirements of the program. From this information PICC18 then "writes" the assembler code which will perform the runtime startup. This code is stored into a file which can then be assembled and linked into the remainder of the program in the usual way.

Since the runtime startup code is generated automatically on every compilation, the generated files associated with this process are deleted after they have been used. If required, the assembler file which contains the runtime startup code can be kept after compilation and linking by using the driver option `--RUNTIME=default,+keep`. The residual file will be called `startup.as` and will be located in the current working directory. If you are using an IDE to perform the compilation the destination directory is dictated by the IDE itself, however you may use the `--OUTDIR` option to specify an explicit output directory to the compiler.

This is an automatic process which does not require any user interaction, however some aspects of the runtime code can be controlled, if required, using the `--RUNTIME` option. These are described in the sections below.

3.3.5.1 Initialization of Data psects

One job of the runtime startup code is ensure that any initialized variables contain their initial value before the program begins execution. Initialized variables are those which are not `auto` objects and which are assigned an initial value in their definition, such as `input` in the following example.

```
int input = 88;
void main(void) { ...
```


Since `auto` objects are dynamically created, they require code to be positioned in the function in which they are defined to perform their initialization. It is also possible that their initial value changes on each instance of the function. As a result, initialized `auto` objects do not use the data psects.

Such initialized objects have two components and are placed within the data psects.

The actual initial values are placed in a psect called `idata`. The other component is where the variables will reside, and be accessed, at runtime. Space is reserved for the runtime location of initialized variables in a psect called `rdata`. This psect does not contribute to the output file.

The runtime startup code performs a block copy of the values from the `idata` to the `rdata` psect so that the RAM variables will contain their initial values before `main()` is executed. Each location in the `idata` psect is copied to appropriate place in the `rdata` psect.

The block copy of the data psects may be omitted by disabling the `init` suboption of `--RUNTIME`. For example:

```
--RUNTIME=default,-init
```

With this part of the runtime startup code absent, the contents of initialized variables will be unpredictable when the program begins execution. Code relying on variables containing their initial value will fail.

Variables whose contents should be preserved over a reset, should be qualified with `persistent`, see Section 3.4.10.1. Such variables are linked at a different area of memory and are not altered by the runtime startup code in any way.

3.3.5.2 Clearing the Bss Psects

The ANSI standard dictates that those non-`auto` objects which are not initialized must be cleared before execution of the program begins. The compiler does this by grouping all such uninitialized objects into a bss psect. This psect is then cleared as a block by the runtime startup code.

The abbreviation "bss" stands for Block Started by Symbol and was an 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.

The name of the bss psects are `rbss`, `bss`, `bigbss` and `farbss`.

The block clear of the `bss` psect may be omitted by disabling the `clear` suboption of `--RUNTIME`. For example:

```
--RUNTIME=default,-clear
```

With this part of the runtime startup code absent, the contents of uninitialized variables will be unpredictable when the program begins execution.

As with initialized variables, those whose contents are to be preserved over a reset, should be qualified `persistent`, see Section 3.4.10.1.

3.3.5.3 Linking in the C Libraries

By default, a set of libraries are automatically passed to the linker to be linked in with user's program. The libraries can be omitted by disabling the `clib` suboption of `--RUNTIME`. For example:

```
--RUNTIME=default,-clib
```

With this part of the runtime startup code absent, the user must provide alternative library or source files to allow calls to library routines. This suboption may be useful if alternative library or source files are available and you wish to ensure that no HI-TECH C library routines are present in the final output.

Some C statements produce assembler code that call library routines even though no library function was called by the C code. These calls perform such operations as division or floating-point arithmetic. If the C libraries have been excluded from the code output, these implicit library calls will also require substitutes.

Table 3.5: Basic data types

Type	Size (bits)	Arithmetic Type
`bit`	1	unsigned integer
`char`	8	signed or unsigned integer[1]
`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[2]	real

3.3.5.4 The powerup Routine

Some hardware configurations require special initialization, often within the first few cycles of execution after reset. 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. No special linking options or jumps to the powerup routine are required, the compiler will detect if you are using a powerup routine and will automatically generate code to jump to it after reset. If you use a powerup routine, you will, however, need to add a jump to `start` after your initializations. Refer to comments in the powerup source file for details about this. The `powerup.as` source file can be found in the compiler's `SOURCES` directory.

3.4 Supported Data Types and Variables

The HI-TECH PICC-18 STD compiler supports basic data types with 1, 2, 3 and 4 byte sizes. 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 3.5 shows the data types and their corresponding size and arithmetic type.

Table 3.6: 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

3.4.1 Radix Specifiers and Constants

The format of integral constants specifies their radix. HI-TECH PICC-18 STD supports the ANSI standard radix specifiers as well as ones which enable binary constants to be specified in C code. The format used to specify the radices are given in Table 3.6. The letters used to specify binary or hexadecimal radices are case insensitive, as are the letters used to specify the hexadecimal digits.

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 HI-TECH PICC-18 STD.

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 the program memory. 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, correct
```

Defining and initializing a non-`const` array (i.e. not a pointer definition) with a string, for example:

```
char ca[]= "two";          // "two" different to the above
```

produces an array in data space which is initialised at startup with the string `"two"` (copied from program space), whereas a constant string used in other contexts represents an unnamed `const`-qualified array, accessed directly in program space.

HI-TECH C 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 the data space as shown in the last statement in the previous 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".

3.4.2 Bit Data Types and Variables

HI-TECH PICC-18 STDsupports 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 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. The bit return value will be returned in the carry flag of the STATUS register.

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, however, not possible to declare pointers to bit variables or statically initialize bit variables.

Operations on bit objects are performed using the single bit instructions (bsf and bcf) 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 assignment 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 psect's 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 PICC18 flag `--STRICT` is used, the `bit` keyword becomes unavailable.

3.4.3 Using Bit-Addressable Registers

The bit variable facility may be combined with absolute variable declarations (see Section 3.5.2) 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 FD0h, then declare a `bit` variable at absolute bit address 4050:

```
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 PIC18 registers is <htc.h> - at compile time this will include the appropriate header for the selected chip.

3.4.4 8-Bit Integer Data Types and Variables

HI-TECH PICC-18 STD supports both `signed char` and `unsigned char` 8-bit integral types. If the `signed` or `unsigned` keyword is absent from the variable's definition, the default type is `unsigned char` unless the PICC18 `--CHAR=signed` option is used, in which case the default type 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 HI-TECH PICC-18 STD the `char` types will commonly be used for a number of purposes, such as 8-bit integers, as storage for ASCII characters, and for access to I/O locations.

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.

3.4.5 16-Bit Integer Data Types

HI-TECH PICC-18 STD supports four 16-bit integer types. `short` and `int` are 16-bit two's complement signed integer types, representing integral values from -32,768 to +32,767 inclusive. `Unsigned short` and `unsigned int` 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.

Variables may be declared using the `signed short int` and `unsigned short int` keyword sequences, respectively, to hold values of these types. 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`.

3.4.6 32-Bit Integer Data Types and Variables

HI-TECH PICC-18 STD 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`.

Table 3.7: 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

Table 3.8: Floating-point format example IEEE 754

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

3.4.7 Floating Point Types and Variables

Floating point is implemented using either a IEEE 754 32-bit format or a modified (truncated) 24-bit form of this.

The 24-bit format is used for all `float` values. For `double` values, the 24-bit format is the default, or if the `--double=24` option is used. The 32-bit format is used for `double` values if the `--double=32` option is used.

This format is described in 3.7, where:

- sign is the sign bit
- The exponent is 8-bits 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 $(-1)^{sign}$ x $2^{(exponent-127)}$ x 1.mantissa.

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

Note that the most significant bit of the mantissa column in 3.8 (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 3.8 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:

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

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`.

There is a fast implementation available for 32-bit doubles. Fast 32-bit doubles can be selected via the `--double=fast32` command line option. Although selecting this offers an improvement in speed when doing calculations on 32-bit doubles, this comes at a cost of code size.

3.4.8 Structures and Unions

HI-TECH PICC-18 STD 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.

3.4.8.1 Bit-fields in Structures

HI-TECH PICC-18 STD 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 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 bytes. 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 bit-fields 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};
```

3.4.8.2 Structure and Union Qualifiers

HI-TECH PICC-18 STD 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 the program space and each member will, obviously, be read-only. Remember that all members must be initialized if a structure is `const` as they cannot be initialized at runtime.

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};
```

3.4.9 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. HI-TECH PICC-18 STD supports both ANSI qualifiers and additional special qualifiers which are useful for embedded applications and which take advantage of the PIC18 architecture.

3.4.9.1 Const and Volatile Type Qualifiers

HI-TECH PICC-18 STD supports the use of the ANSI type qualifiers `const` and `volatile`.

The `const` type qualifier is used to tell the compiler that an object is read only 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 special psects in the program space. Obviously, a `const` object must be initialised when it is declared as it cannot be assigned a value at any point at runtime. 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 using different generated code to non-`volatile` objects. For example, when assigning a non-volatile object the value 1, the object may 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.

3.4.10 Special Type Qualifiers

HI-TECH PICC-18 STD supports special type qualifiers, `persistent`, `near` and `far` to allow the user to control placement of `static` and `extern` class variables into particular address spaces. If the PICC18 option, `--STRICT` is used, these type qualifiers are changed to `__persistent`, `__near` and `__far`, respectively. 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) {
   persistent int intvar; /* WRONG! */
   ... other code ...
}
```

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

```
static persistent int intvar;
```

HI-TECH PICC-18 STD also supports the keywords `bank1`, `bank2` and `bank3`. These keywords have been included to allow code to be easily ported from PICC. These keywords are accepted by HI-TECH PICC-18 STD, but have no effect in terms of the object's storage or how they are accessed. These keywords do, however, affect the storage of objects when compiling with the PICC compiler - see your PICC manual for more details.

3.4.10.1 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.

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. `Persistent` objects are placed within one of the non-volatile psects. If the `persistent` object is also qualified `near`, it placed in the `nvrram` psect. `Persistent bit` objects are placed within the `nvbit` psect. All other `persistent` objects are placed in the `nvram` psect.

There are some library routines provided to check and initialise `persistent` data - see Appendix A for more information, and for an example of using `persistent` data.

3.4.10.2 Near Type Qualifier

The `near` type qualifier is used to place static variables in the *access bank* of the PIC18. `Near` objects are represented by 8 bit addresses and the access bank is always accessable regardless of the currently selected RAM bank so accessing `near` objects may be faster than accessing other objects, and typically results in smaller code sizes.

Here is an example of an unsigned char object placed within the access bank:

```
static near unsigned char fred;
```

3.4.10.3 Far Type Qualifier

The `far` type qualifier is used to place variables of permanent duration into *external program space* of the PIC18 for those devices which can support additional memory. Accesses to `far` variables are less efficient than accesses to internal variables and extensive accesses to these variables will result in larger code sizes.

Here is an example of an `unsigned int` object placed into the device's external code space:

```
far unsigned int farvar;
```

Note that not all devices support extending their memory space in this way and the `far` qualifier is not applicable to all PIC18 devices. For those devices that can extend their memory, the address range where the additional memory will be mapped must first be specified with a –RAM option. For example, to map additional data memory from 20000h to 2FFFFh use `--RAM=default,+20000-2FFFF`.

Also consider that if the external memory area uses addresses greater than FFFFh (as in most cases) the `--CP=24` command line option will also be required in order to access these variables correctly.

3.4.11 Bdata Type Qualifier

The `bdata` type qualifier only has significance when compiling in the small memory model. In this model all `static` and `extern` class variables are placed in the access bank, but this qualifier specifies that the object is to be placed outside the access bank in the banked data area of the device. The object then behaves like an unqualified object in large model. This qualifier is useful when the access bank has overflowed by a small amount as it allows some objects to moved back into the banked memory and prevents having to revert to the large memory model.

3.4.12 Pointer Types

There are two basic pointer types supported HI-TECH PICC-18 STD: data pointers and function pointers. Data pointers hold the address of data objects which can be read and/or written by the program. Function pointers hold the address of an executable routine which can be called indirectly via the pointer.

Of the data pointers, RAM pointers are limited to accessing only the data space (RAM) of the PIC18 device, but the const and far pointer types can access the data and program space (typically ROM, although hardware using devices with an external memory interface may implement any type of memory in this space).

3.4.12.1 RAM Pointers

All RAM pointer objects on these PIC18 devices are 16 bits wide, with the exception of pointers to objects qualified as near which are 8 bits wide.

A pointer to RAM, for example:

```
char * cp;
```

is 16 bits wide and can access all of the RAM available on the PIC18 devices.

A pointer to `near` is only 8 bits wide and can access the general-purpose RAM area of the access bank. In other words they can be used to dereference any variable qualified as `near`. The amount of general purpose RAM in the access bank varies from device to device. Being smaller in size, using pointers to `near` result in smaller code sizes. If a pointer only ever accesses `near`-qualified objects, then that pointer should be qualified as a pointer to `near`.

The operation of RAM pointers is unaffected by the `--CP=24`/`--CP=16` switch, nor are they affected by the choice of memory model.

3.4.12.2 Const and Far Pointers

Const and far pointers can either be 16 or 24 bits wide. Their size can be toggled with the `--CP=24` or `--CP=16` command line option. The code used to dereference them also changes with their size. The same pointer size must be used for all modules in a project.

A pointer to `far` is identical to a pointer to `const`, except that pointers to `far` may be used to write to the address they hold. A pointer to `const` objects cannot be used to write as the `const` qualifier imposes that the object is read-only.

`Const` and `far` pointers which are 16 bits wide can access all RAM areas and most of the program space. At runtime when dereferenced, the contents of the pointer are examined. For addresses above the upper limit of RAM the program space is accessed using table read or table write instructions. Addresses below the upper limit of RAM access the data space. Even if the address held by a pointer to `const` is in RAM, the RAM location may not be changed.

The default linker options always place `const` data at addresses above the upper limit of the data space so that the correct memory space is accessed when dereferencing with pointers.

If the target device selected has more than 64k bytes of program space memory, then only the lower 64k bytes may be accessed with 16-bit wide pointers. Provided that all program space objects that need to be dereferenced are in the lower 64k bytes, 16-bit pointers to `const` and `far` objects may still be used. The smaller pointer size results in less RAM required and less code produced and so should be used whenever possible.

`Const` and `far` pointers which are 24 bits wide can access all RAM areas and all of the program space. At runtime when dereferenced, the contents of the pointer are examined. If bit number 21 in the address is set, the address is assumed to be a RAM address. Bit number 21 of the address is

then ignored. If Bit number 21 is clear, then the address is assumed to be of an object in the program space and the access is performed using table read or table write instructions. Again, no writes to objects are permitted using a pointer to `const`.

Note that when dereferencing a 24-bit pointer, the most significant implemented bit (bit number 21) of the `TBLPTRU` register may be overwritten. This bit may be used to enable access to the configuration area of the PIC18 device. If loading the table pointer registers from hand-written assembler code, make no assumptions about the state of bit number 21 prior to executing table read or write instructions.

3.4.12.3 Function Pointers

Function pointers can be defined to indirectly call functions or routines in the program space. The size of these pointers are 16 or 24 bits wide and is controlled by the `--CP=24`/`--CP=16` command line option. When 16-bit wide function pointers are used only routines within the lower 64k bytes can be indirectly called. The larger 24-bit function pointer allows indirect calls to be made to any routine, but at the expense of increased code size and RAM usage.

It should be stressed that the `--CP=16` option affects function pointer sizes and that it does not affect code that calls functions directly, i.e. by their name rather than indirectly via a pointer. Thus you can still directly call functions residing at any location even if you are using the `--CP=16` option, however you can only indirectly call functions that reside in the lower 64k byte area of the program space.

The addresses for all code labels are always shown in the map file as an untruncated byte address regardless of the options used.

3.4.12.4 Combining Type Qualifiers and Pointers

Pointers can be qualified like any other C object, but care must be taken when doing so as there are two quantities associated with pointers. The first is the actual *pointer* itself, which is treated like any ordinary C variable and has memory reserved for it. The second is the *object* that the pointer references, or to which the pointer points. The general form of a pointer definition looks like the following.

“object’s type & qualifiers” * “pointer’s qualifiers” “pointer’s name” ;

TUTORIAL

Here are three examples, highlighting the fields with spacing:

near int * nip ;

int * near inp ;

near int * near ninp ;

The first example is a pointer called `nip`. It contains the address of an `int` object that is qualified `near`. Since a `near` object is in the access bank the pointer is only 8 bits wide as discussed above. The pointer itself (i.e. the 8-bit value the pointer holds) will reside somewhere in the main banked memory.

The second example is a pointer called `inp` which contains the address of an `int` object. Since this object is not qualified `near`, the pointer needs 16 bits to access the object's location. The `near` keyword after the * indicates that the pointer itself has been qualified `near` and so the pointer (i.e. the 16-bit value the pointer holds) will reside in the access bank, but the object whose address the pointer holds is located in the main banked memory.

The last example is of a pointer called `ninp` which is itself qualified `near` and which also holds the address of an object that is also qualified `near`. In this example, both the pointer and the object that the pointer references will be located in the access bank. The pointer will be 8 bits wide.

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.

The `const`, `volatile`, `far` and `persistent` modifiers may also be applied to pointers in the above manner.

To allow portability between PICC and PICC-18 code, the use of the `bank1`, `bank2` and `bank3` keywords is allowed with PICC-18 pointer definitions in the manner described above. These keywords have no effect and do not alter the way in which indirect accesses are made. The use of these keywords with pointers defined in PICC does affect the operation of pointer dereferences - see your PICC manual for more details.

3.5 Storage Class and Object Placement

Objects are positioned in different memory areas dependant on their storage class and declaration. This is discussed in the following sections.

3.5.1 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.

3.5.1.1 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 from 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 function's `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 PIC18. 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.

3.5.1.2 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 sized 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 PIC18.

`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.

3.5.2 Absolute Variables

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

```
volatile unsigned char Portvar @ 0xF80;
```

will declare a variable called `Portvar` located at F80h. 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 F80h
```

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 special function 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 as specified in Section 3.12.3.3.

Absolute variables are accessed using the address specified with their definition, thus there are no symbols associated with them. Because the linker never sees any symbols for these objects it is not aware that they have been allocated space and it cannot make any checks for overlap of absolute variables with other objects. It is entirely the programmer's responsibility to ensure that absolute variables are allocated memory that is not already in use.

3.5.3 Objects in Program Space

`Const` objects are usually placed in program space. On the PIC18 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 3.4.12.

3.6 Functions

3.6.1 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 `PICC18 -S` option, allowing you to examine the assembler code.

3.6.2 Function Return Values

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

3.6.2.1 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
```

3.6.2.2 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
```

3.6.2.3 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_return_struct)
movwf   fsr0l
movlw   high(?a_return_struct)
movwf   fsr0h
movlw   structret
movwf   fsr1l
clrf    fsr1h
movlw   8
global  structcopy
call    structcopy
```

3.6.3 Memory Models and 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 information in the chipinfo file (which defaults to picc-18.ini in the DAT directory). The linker will automatically locate code and `const`-qualified data into all the available memory pages and ensure that psects do not straddle any memory boundary.

Temporary variables created and used by the compiler are placed in the access bank to increase efficiency.

There are two memory models available for HI-TECH PICC-18 STD: small and large. The default memory model is large. The memory model is selected via the -B*model* command line option. 2.4.1.

In large memory model, all objects qualified `near` are placed in a "near" psect (e.g. `rbss`, `rdata`) which are positioned in the access bank. These objects can be efficiently accessed using less generated code than other objects. All other objects are placed in the PIC18's banked memory space in the following manner. Each module can allocate up to one bank of initialized (`data` psect), and one bank of uninitialized (`bss` psect), `global` or `static` local objects. Any object that is larger than one bank in size (e.g. an array) is placed in a separate area (in one of the "big" psects) and this area can grow across bank boundaries to a size limited only by the available space on the device. Any

single byte objects are also placed in one of the "big" psects ("big" refers to the size of the psect, not the size of the objects within the psect). All `auto` and parameter variables from all functions are overlapped by the linker if possible and then placed into an available RAM bank.

In small model, all objects qualified `near` are placed in a "near" psect (e.g. `rbss`, `rdata`) which are positioned in the access bank as per the large model. The `global` and `static` local initialized and uninitialized objects are also placed in the access bank as are all single byte objects. Objects larger than the access bank size are positioned in a separate area (in one of the "big" psects) and this area can grow across bank boundaries to a size limited only by the available space on the device. All `auto` and parameter variables are positioned as per the large model.

3.7 Register Usage

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.

3.8 Operators

HI-TECH PICC-18 STD supports all the ANSI operators. The exact results of some of these are implementation defined. The following sections illustrate code produced by the compiler.

3.8.1 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*. HI-TECH PICC-18 STD 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 a `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 above 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, HI-TECH PICC-18 STD 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 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.

Table 3.9: Integral division

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

3.8.2 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-18 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.

3.8.3 Division and modulus with integral types

The sign of the result of division with integers when either operand is negative is implementation specific. 3.9 shows the expected sign of the result of the division of operand 1 with operand 2 when compiled with PICC-18.

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

3.9 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 `PICC18` 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 Section 4.3.8.3). In C, user-defined psects can be created by using the `#pragma psect` preprocessor directive, see Section 3.12.3.3.

3.9.1 Compiler-generated Psects

The code generator places code and data into psects with standard names which are subsequently 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 These psects contain the ROM image of any initialised variables. These psects are copied into the `data` psects at startup.

irdata These psects contain the ROM image of any initialised `near` variables. These psects are copied into the `rdata` psects at startup.

ibigdata These psects contain the ROM image of initialised objects which at runtime reside in the `bigdata` psect. This includes `global` or `static` local `char` objects or `char` arrays, and arrays whose size exceeds the size of a RAM bank.

ifardata This psect contains the ROM image of initialised objects which at runtime reside in the `fardata` psect.

text Is a `global` psect used for executable code and library functions.

const These psects hold objects that are declared `const` and string literals which are not modifiable.

config Used to store the configuration words.

idloc Used to store the ID location words.

eeprom_data Used to store data to be programmed into the EEPROM data area.

intcode Is the psect which contains the executable code for the entry point to the default or high-priority interrupt service routine. This psect is linked to the interrupt vector at address 08H.

intcodelo Is the psect which contains the executable code the low-priority interrupt service routine. This psect is linked to the interrupt vector at address 018H.

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

end_init Used by initialization code which, for example, clears RAM.

clrtext Used by some startup routines for copying the `data` psects.

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

rbss These psects contain any uninitialized `near` variables. They reside in the access bank.

bigbss These psects contain any uninitialized `global` or `static` local `char` objects or `char` arrays, and arrays whose size exceeds the size of a RAM bank. This psect is linked into a psect class which does not have RAM bank boundaries. Accessing objects in this area may be less efficient that accessing objects in the `data` psect.

farbss This psect contains any uninitialized objects which have been declared as `far` to be positioned in external code space. The location of this psect must be be specified to the compiler with a `--RAM` option which defines an extra address range beginning at an address greater than the top address of program memory.

fardata This psect contains initialized objects which have been declared as `far` to be positioned in external code space. As with farbss, the address range to accomodate this external memory must first be specified in a `--RAM` option.

bss These psects contain any uninitialized variables not contained in the above psects.

rdata These psects contain any initialised `near` variables. They reside in the access bank.

bigdata These psects contain any initialized `global` or `static` local char objects or `char` arrays, and arrays whose size exceeds the size of a RAM bank. This psect is linked into a psect class which does not have RAM bank boundaries. Accessing objects in this area may be less efficient that accessing objects in the data psect.

data This psect contains initialised variables for a program that are not contained in any of the above psects. This psect will be wholly contained within a RAM bank and so that it can be accessed more efficiently.

nvrram This psect holds `near persistent` variables. It is not cleared or otherwise modified by the runtime startup code.

nvbit This psect holds `persistent bit` objects. It is not cleared or otherwise modified at startup.

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

rbit This psect is used to store all `bit` variables. All bit objects are `near` by default and are placed in the access bank.

struct Contains any structure larger than 4 bytes in size which is returned from a function.

intsave_regs Holds the registers (including temporary locations) saved by the interrupt service routine.

temp Is used to store scratch variables used by the compiler. These include function return values larger than a byte and values passed to and returned from library routines. This psect will be positioned in the access bank.

3.10 Interrupt Handling in C

The compiler incorporates features allowing interrupts to be handled from C code. Interrupt functions are often called *interrupt service routines* (ISR). Interrupts are also known as *exceptions*. PIC18 devices have two separate interrupt vectors and a priority scheme to dictate how the interrupt code is called.

3.10.1 Interrupt Functions

The function qualifier `interrupt` may be applied to at most two functions to allow them to be called directly from the hardware interrupts. 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 `PICC18` option `--STRICT` is used, the interrupt keyword becomes `__interrupt`. Wherever this manual refers to the `interrupt` keyword, assume `__interrupt` if you are using `--STRICT`.)

The PIC18 devices have two interrupts, each with their own vector location. These have different priorities and are known as *low-priority* and *high-priority interrupts*. If the PIC18 is placed in compatibility mode, only one interrupt is available and this defaults to being the high-priority interrupt. An `interrupt` function must be declared as type `interrupt void` and may not have parameters. In addition, the keyword `low_priority` may be used to indicate that the `interrupt` function is to be linked with the low-priority vector when not in compatibility mode. `Interrupt` functions may not be called directly from C code, but they may call other functions themselves, subject to certain limitations. Once defined, the `interrupt` function is linked to the corresponding interrupt vector.

An example of a high-priority (default) `interrupt` function is shown here.

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

```
}
```

A `low-priority interrupt` function may be defined as in the following example.

```
void interrupt low_priority tc_clr(void){
   tick_count = 0;
}
```

It is up to the user to determine and set the priority levels associated with each interrupt source on the PIC18 devices. Defining a low-priority `interrupt` function does *not* put the PIC into interrupt-priority mode.

Low- and high-priority `interrupt` functions have their own separate areas of memory in which to save context, thus a high-priority `interrupt` function may interrupt a low-priority `interrupt` function with no loss of data. The high-priority **interrupt** can also employ the devices' shadow registers to enable rapid context switching during the entry and exit of the service routine.

The `interrupt_level` pragma may be used with either or both `interrupt` functions in the usual way.

3.10.2 Context Saving on Interrupts

The PIC18 processor only saves the program counter on its stack whenever an interrupt occurs. Other registers and objects must be saved in software. PICC-18 automatically 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-18 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 if they are not already saved by the interrupt entry routine.

By default, the high-priority interrupt function will utilize a **fast interrupt save/restore** technique where the W, STATUS and BSR registers are saved and restored via the devices' internal shadow registers. This minimizes code size and reduces the instruction cycles to access the high-priority service routine. Note that for some older devices, the compiler will not apply **fast interrupt save/restore** if compiling for the MPLAB ICD2 debugger, as the debugger istself utilizes these shadow registers.

The high-priority or compatibility-mode `interrupt` function places a small routine in a psect called `intcode` which is linked directly to the interrupt vector. This code saves the STATUS (if **fast interrupts** are not used) and PCLATH registers then jumps to code placed in a `text` psect. This code

will save further context if it is necessary and then jump to code directly related to the `interrupt` function. The `interrupt` function code is also placed in a `text` psect.

All objects saved are done so to locations at an offset to a symbol called `saved_regsh`, except for the BSR register. If **fast interrupts** are not used, BSR is saved to a location symbol called `saved_bsrh`.

The low-priority interrupt function places the code to save the STATUS and PCLATH registers in a psect called `intcodelo`, which is directly linked to the low-priority interrupt vector. Operation is then similar to the high-priority interrupt case, only with objects being saved offset to the symbol `saved_regsl` and the BSR register saved to a location symbol called `saved_bsrl`.

3.10.3 Context Retrieval

Any objects saved by the compiler are automatically restored before the interrupt function returns. The restoration code is placed into a `text` psect. The `retfie` instruction placed at the end of the interrupt code will reload the program counter and the program will return to the location at which it was when the interrupt occurred.

3.10.4 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 function 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 to suppress the errors generated.

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. This exclusion must be guaranteed by the user, i.e. the compiler is not able to control interrupt priorities. Each `interrupt` function 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 interrupts of one or more levels. This will prevent the 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 main-line code */
```

```
#pragma interrupt_level 1
void bill(){
   int i;
   i = 23;
}
/* two interrupt functions calling same non-interrupt function */
#pragma interrupt_level 1
void interrupt fred(void){
   bill();
}
#pragma interrupt_level 1
void interrupt joe(void){
   bill();
}
main(){
   bill();
}
```

Both the low- and high-priority `interrupt` functions may use the interrupt level feature.

3.10.5 Interrupt Registers

It is up to the user how they want the interrupt source configured. All the registers and bits associated with interrupts are defined in the specific header file which can be accessed by including `<htc.h>`. The following is an example of setting up the interrupts associated with the change-on-PORTB source. Interrupt priorities are used and the interrupt source is made a low priority. See your PIC18 datasheet for more information.

```
void main(void){
   TRISB = 0x80;   // Only RB7 will interrupt on change
   IPEN  = 1;      // Interrupt priorities enabled
   PEIE  = 1;      // enable peripheral interrupts
   RBIP  = 0;      // make this a low priority interrupt
   RBIE  = 1;      // enable PORTB change interrupt
   RBIF  = 0;      // clear any pending events
   GIEL  = 1;      // enable low-priority interrupts
   while(1)continue; // sit here and wait for interrupt
}
void interrupt low_priority b_change(void){
   if(RBIE && RBIF){
```

```
            PORTB;    // Read PORTB to clear any mismatch
            RBIF = 0; // clear event flag
            // process interrupt here
        }
    }
```

3.11 Mixing C and Assembler Code

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

3.11.1 External Assembly Language Functions

Entire functions may be coded in assembly language as separate `.as` source files, assembled by the assembler (`ASPIC18`) and combined into the binary image using the linker. This technique allows arguments and return values to be passed between C and assembler code.

The following are guidelines that must be adhered to when writing a routine in assembly code that is callable from C code.

- select, or define, a suitable psect for the executable assembly code
- select a name (label) for the routine so that its corresponding C identifier is valid
- ensure that the routine's label is globally accessable from other modules
- select an appropriate equivalent C prototype for the routine on which argument passing can be modelled
- ensure any symbol used to hold arguments to the routine is globally accessable
- ensure any symbol used to hold a return value is globally accessable
- optionally, use a signature value to enable type checking when the function is called
- write the routine ensuring arguments are read from the correct location, the return value is loaded to the correct storage location before returning
- ensure any local variables required by the routine have space reserved by the appropriate directive

A mapping is performed on the names of all C functions and non-`static` global variables. See 3.11.2 for a description of mappings between C and assembly identifiers.

TUTORIAL

An assembly routine is required which can add two 16-bit values together. The routine must be callable from C code. Both the values are passed in as arguments when the routine is called from the C code. The assembly routine should return the result of the addition as a 16-bit quanity.

Most compiler-generated executable code is placed in a psect called `text` (see Section 3.9.1). As we do not need to have this assembly routine linked at any particular location, we can use this psect so the code is bundled with other executbale code and stored somewhere in the program space. This way we do not need to use any additional linker options. So we use an ordinary looking psect that you would see in assembly code produced by the compiler. The psect's name is `text`, will be linked in the CODE class, which will reside in a memory space that has 1 bytes per addressable location:

```
PSECT text,local,class=CODE,delta=1
```

Now we would like to call this routine `add`. However in assembly we must choose the name `_add` as this then maps to the C identifier `add` since the compiler prepends an underscore to all C identifiers when it creates assembly labels. If the name `add` was chosen for the assembler routine the it could never be called from C code. The name of the assembly routine is the label that we will assocaite with the assembly code:

```
_add:
```

We need to be able to call this from other modules, so make this label globally accessable:

```
GLOBAL _add
```

By compiling a dummy C function with a similar prototype to how we will be calling this assembly routine, we can determine the signature value. We add an assembler directive to make this signature value known:

```
SIGNAT _add,8298
```

When writing the function, you can find that the parameters will be loaded into the function's parameter area by the calling function, and the result should be placed in btemp.

To call an assembly routine from C code, a declaration for the routine must be provided. This ensures that the compiler knows how to encode the function call in terms of paramters and return values, however no other code is necessary.

If a signature value is present in the assembly code routine, its value will be checked by the linker when the calling and called routines' signatures can be compared.

TUTORIAL

To continue the previous example, here is a code snippet that declares the operation of the assembler routine, then calls the routine.

```
extern unsigned int add(unsigned a, unsigned b);
void main(void)
{
    int a, result;
    a = read_port();
    result = add(5, a);
}
```

3.11.2 Accessing C objects from within Assembly Code

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 if 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 most PIC18 instructions. The exceptions are the `movff` and `lsfr` instructions. Failure to do this may result in fixup errors issued by the linker. If in doubt as to writing assembler which accesses 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.

C identifiers are assigned different symbols in the output assembly code so that an assembly identifier cannot conflict with an identifier defined in C code. If assembly programmers choose identifier names that do not begin with an *underscore*, these identifiers will never conflict with C identifiers. Importantly, this implies that the assembly identifier, `i`, and the C identifier `i` relate to different objects at different memory locations.

3.11.2.1 Accessing special function register names from assembler

If writing separate assembly modules, SFR definitions will not automatically be present. If writing assembler code from within a C module, SFRs may be accessed by referring to the symbols defined by the chip-specific C header files. Whenever you include `<htc.h>` into a C module, all the available SFRs are defined as absolute C variables. As the contents of this file is C code, it cannot be included into an assembler module, but assembler code can uses these definitions. To use a SFR in in-line assembler code from within the same C module that includes `<htc.h>`, simply use the symbol with an *underscore* character prepended to the name. For example:

```
#include <htc.h>
void main(void)
{
    PORTA = 0x55;
    asm("movlw 0xAA");
    asm("movwf _PORTA");
    ...
}
```

Alternately, an assembler equivalent of htc.h is provided. If an assembler source includes the file aspic18.h, equivalent assembler definitions of the devices' SFRs will be available.

3.11.3 #asm, #endasm and asm()

PIC18 instructions may also be directly embedded "in-line" into C code using the directives `#asm`, `#endasm` or the statement `asm()`.

The `#asm` and `#endasm` directives are used to start and end a block of assembly instructions which are to be embedded into the assembly output of the code generator. 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, however you can easily include multiple instructions with this form of in-line assembly.

The `asm()` statement is used to embed a single assembler instruction. This form looks and behaves like a C statement, however each instruction must be encapsulated within an `asm()` statement.

You should not use a `#asm` block within any C constructs such as `if`, `while`, `do` etc. In these cases, use only the `asm("")` form, which is a C statement and will correctly interact with all C flow-of-control structures.

The following example shows both methods used to rotate a byte left through carry:

```
unsigned char var;
void main(void){
   var = 1;
#asm   // like this...
   movlb (_var) >> 8
   rlcf  (_var)&0ffh,f
#endasm
   asm("movlb (_var)>>8");
   asm("rlcf (_var)&0ffh,f");
}
```

When using in-line assembler code, great care must be taken to avoid interacting with compiler-generated code. If in doubt, compile your program with the `PICC18 -S` option and examine the assembler code generated by the compiler.

3.12 Preprocessing

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

3.12.1 Preprocessor Directives

HI-TECH PICC-18 STD accepts several specialised preprocessor directives in addition to the standard directives. All of these are listed in the table below.

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

Table 3.10: Preprocessor directives

Directive	Meaning	Example
`#`	preprocessor null directive, do nothing	`#`
`#assert`	generate error if condition false	`#assert SIZE > 10`

`#asm`	signifies the beginning of in-line assembly	`#asm`
		`mov r0, r1h`
		`#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 filename for listing	`#line 3 final`
`#`*nn*	(where *nn* is a number) short for `#line` *nn*	`#20`
`#pragma`	compiler specific options	`See section 3.12.3`
`#undef`	undefines preprocessor symbol	`#undef FLAG`
`#warning`	generate a warning message	`#warning Length not set`

3.12.2 Predefined Macros

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

Symbol	When set	Usage
continued...		
HI_TECH_C	Always	To indicate that the compiler in use is HI-TECH C.
_HTC_VER_MAJOR_	Always	To indicate the compiler's major version number.
_HTC_VER_MINOR_	Always	To indicate the compiler's minor version number.
_HTC_VER_PATCH_	Always	To indicate the patch level of the compiler's version number.
_HTC_EDITION_	Always	To indicate which of PRO, STD or Lite compiler is in use. Values of 2, 1 or 0 are assigned respectively.
LARGE_DATA	--CP=24	To indicate that program space pointers are 24 bits in size.
SMALL_DATA	--CP=16	To indicate that program space pointers are 16 bits in size.
LARGE_MODEL	-Bl	To indicate code is compiled in large memory model.
SMALL_MODEL	-Bs	To indicate code is compiled in small memory model.
__PICC18__	Always	To indicate the use of the HI-TECH PICC-18 compiler.
MPC	Always	To indicate the code is compiled for the Microchip PIC family.
_PIC18	Always	To indicate that this is a PIC18 device.
_ROMSIZE	Always	To indicate the number of bytes of program space this device has.
_RAMSIZE	Always	To indicate the number of bytes of data space this device has.
_EEPROMSIZE	If EEPROM is present	To indicate if EEPROM memory is present and how many bytes are available.

Symbol	When set	Usage
_FLASH_ERASE_SIZE	Always	To indicate the number of bytes erased in a single flash-erase operation at runtime.
_FLASH_WRITE_SIZE	Always	To indicate the number of bytes erased in a single flash-write operation at runtime.
__MPLAB_ICD2__	--DEBUGGER=ICD2	To indicate that the code is being generated for the Microchip ICD2 In-Circuit debugger.
MPLAB_ICD	--DEBUGGER=ICD2	As above, value is 2
_ICDROM_START	--DEBUGGER=ICD2	Defined the start address of the ICD2's reserved program space
_ICDROM_END	--DEBUGGER=ICD2	Defined the end address of the ICD2's reserved program space
_ERRATA_TYPES	Always	Defines a bitmask to show which types of silicon errata may be applicable to this build.
_chipname	When chip selected	To indicate the specific chip type selected
__FILE__	Always	To indicate this source file being preprocessed.
__LINE__	Always	To indicate this source line number.
__DATE__	Always	To indicate the current date, e.g. May 21 2004
__TIME__	Always	To indicate the current time, e.g. 08:06:31.

3.12.3 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 3.12. Those keywords not discussed elsewhere are detailed below.

Table 3.12: Pragma directives

Directive	Meaning	Example
`interrupt_level`	Allow interrupt function to be called from main-line code. See Section 3.10.4.	`#pragma inline(fabs)`
`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 fsr0,fsr1`
`switch`	Specify code generation for switch statements	`#pragma switch direct`

3.12.3.1 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 *backslash,* `\`, 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.

3.12.3.2 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 runtime, 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 any visible effect. See Section 2.4.54.

3.12.3.3 The #pragma psect Directive

Normally the object code generated by the compiler is broken into the standard psects as described in Section 3.9.1. 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.

The general form of this pragma looks like:

```
#pragma psect default_psect=new_psect
```

and instructs the code generator that anything that would normally appear in the compiler-generated psect *default_psect*, will now appear in a new psect called *new_psect*. This psect will be identical to *default_psect* in terms of its options, however will have a different name. Thus, this new psect can be explicitly positioned by the linker without affecting the original psect's location.

If the name of the default psect that is being redirected contains a counter, e.g. `absbss0`, `absbss1`, `absbss2`, then the placeholder `%u` should be used in the name of the psect at the position of the counter, e.g. `absbss%u`. Any default psect, regardless of the counter value, will match such a psect name.

This pragma remains in force until the end of the module and any given psect should only be redirected once in a particular module. All psect redirections for a particular module should be placed at the top of the source file, below any `#include` statements and above any other declarations.

TUT•RIAL

A particular function, called `read_port()`, needs to be located at the absolute address 0x400 in a program. Using the `#pragma psect` directive in the source code, and adding a new linker option can do this. First write the function in the usual way. Place the function definition in a separate module. There is obvioulsy something special about this function so a module all to itself is probably a good idea anyway.

```
unsigned char read_port(void)
{
    return PORTA;
}
```

Now, how do we know in which psect the code associated with the function will be placed? Compile you program, inlcuding this new module and generate an assembly list file, see Section 2.4.18.

Look for the definition of the function. A function starts with an assembly label which is the name of the function prepended with an *underscore*. In this example, the label appears on line 37.

```
36              psect text
37 0002 _read_port:
```

Look above this to see the first `PSECT` directive you encounter. This will indicate the name of the psect in which the code is located. In this case it is the psect called `text`.

So let us redirect this psect into one with a unique and more meaningful name. In the C module that contains the definition for `read_port()` place the following pragma:

```
#pragma psect text=readport
```

at the top of the module, before the function definition. With this, the `read_port()` function will be placed in the psect called `readport`. Confirm this in the new assembly list file.

Now we can tell the linker where we would like this psect positioned. Issue an additional option to the command-line driver to place this psect at address 0x400.

```
-L-preadport=0400h
```

Then generate and check the map file, see Section 2.4.9. You should see the additional linker command (minus the leading `-L` part of the option) present in the section after `Linker command line:`. You should also see the remapped psect name appear in the source file list of psects, e.g.:

```
            Name      Link  Load Length Selector Space Scale
/tmp/cgt9e31jr.obj
main.obj    maintext  0     0    2      0        0
            portread  400   400  2      800      0
```

Check the link address to ensure it is that requested, in this case, 0x400.

3.12.3.4 The #pragma regsused Directive

HI-TECH C 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.

Table 3.13: Valid Register Names

Register Name	Description
wreg	W register
status	STATUS register
pclat	PCLATH register
prodl, prodh	product result registers
btemp, btemp+1...btemp+11	btemp temporary registers 0 to 11
xbtemp	btemp temporary registers 12+
fsr0, fsr1, fsr2	indirect data pointers 0, 1 and 2

Table 3.14: switch types

switch type	description
`auto`	use smallest code size method (default)
`direct`	table lookup (fixed delay)

The `#pragma regsused` directive allows the programmer to indicate register usage for functions that will not be "seen" by the code generator, for example if they were written in assembly code.

The general form of the pragma is:

```
#pragma regsused register_list
```

where *register_list* is a space-separated list of registers names. Those registers not listed are assumed to be unused by the function or routine. The code generator may use the unlisted registers to hold values across a function call. Hence, if the routine does in fact use these registers, unreliable program execution may eventuate.

The register names are not case sensitive and a warning will be produced if the register name is not recognised. A blank list indicates that the specified function or routine uses no registers.

3.12.3.5 The #pragma switch Directive

Normally the compiler decides the code generation method for `switch` statements which results in the smallest possible code size. The #pragma switch directive can be used to force the compiler to use one particular method. The general form of the switch pragma is:

```
#pragma switch switch_type
```

where switch_type is one of the available switch methods listed in Table 3.14.

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 timing is an issue for `switch` statements (i.e.: state machines).

This pragma affects all code generated onward. The `auto` option may be used to revert to the default behaviour.

3.13 Linking Programs

The compiler will automatically invoke the linker unless requested to stop after producing assembler code (`PICC18 -S` option) or object code (`PICC18 -C` option).

HI-TECH C, by default, generates Intel HEX output files. Use the `--OUTPUT=` option to specify a different output format.

After linking, the compiler will automatically generate a memory usage map which shows the address used by, and the total sizes of, all the psects which are used by the compiled code.

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.

More detailed memory usage information, listed in ascending order of individual psects, may be obtained by using the `PICC18 --SUMMARY=psect` option. Alternately, generate a map file for the complete memory specification of the program.

3.13.1 Replacing Library Modules

Although HI-TECH C comes with a librarian (`LIBR`) which allows you to unpack library files and replace modules with your own modified versions, you can easily replace a library module that is linked into your program without having to do this. If you have a source file which contains an alternate implementation of a library routine, by adding this source file to the command-line list of source files, the routine provided will be used in preference to the equivalent routine in the library.

This method works due to the way the linker scans source and library files. When trying to resolve a symbol (in this instance a function name) the linker first scans all source modules for the definition. Only if it cannot resolve the symbol in these files does it then search the library files. Even though the symbol may be defined in a source file and a library file, the linker will not search the libraries and no multiply defined symbol error will result. This is not true if a symbol is defined twice in source files.

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.

```
PICC18 --chip=18F242 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.

3.13.2 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. HI-TECH PICC-18 STD is only likely to issue a mismatch error from the linker when the routine is either a precompiled object file or an assembly routine. Other function mismatches are reported by the code generator.

TUTORIAL

It is sometimes necessary to write assembly language routines which are called from C using an `extern` declaration. Such assembly language functions should 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 PICC18 `-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 assembler 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)
{
}
```

Table 3.15: Supported standard I/O functions

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

and compile it to assembler code using

```
PICC18 -S x.c
```

The resultant assembler code includes the following line:

```
SIGNAT _widget,8249
```

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

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

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

3.13.3 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 the load start address is specified as `__B`*`name`*.

3.14 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 3.15. More details of these functions can be found in Appendix A.

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.

3.15 Debugging Information

3.15.1 MPLAB-specific information

Certain options and compiler features are specifically intended to help MPLAB perform symbolic debugging. The `--IDE=MPLAB` switch performs two functions, both specific to MPLAB. Since MPLAB does not read the local symbol information produced by the compiler, this option generates additional global symbols which can be used to represent most local symbols in a program.

The `--IDE=MPLAB` switch also alters the line numbering information produced so that MPLAB can better follow the C source when performing source-level stepping.

This option also adjusts the format for compiler errors so that they can be more readily interpreted by the MPLAB IDE.

Chapter 4

Macro Assembler

The Macro Assembler included with HI-TECH PICC-18 STD assembles source files for PIC18 MCUs. This chapter describes the usage of the assembler and the directives (assembler pseudo-ops and controls) accepted by the assembler in the source files.

The HI-TECH C Macro Assembler package includes a linker, librarian, cross reference generator and an object code converter.

Athough the term "assembler" is almost universally used to decribe the tool which converts human-readable mnemonics into machine code, both "assembler" and "assembly" are used to describe the source code which such a tool reads. The latter is more common and is used in this manual to describe the language. Thus you will see the terms *assembly language* (or just *assembly*), *assembly listing* and etc, but *assembler options*, *assembler directive* and *assembler optimizer*.

4.1 Assembler Usage

The assembler is called `ASPIC18` and is available to run on *Windows* and *UNIX* machines. Note that the assembler will not produce any messages unless there are errors or warnings — there are no "assembly completed" messages.

Typically the command-line driver, PICC18, is used to envoke the assembler as it can be passed assembler source files as input, however the options for the assembler are supplied here for instances

where the assembler is being called directly, or when they are specified using the command-line driver option `--SETOPTION`, see Section 2.4.50.

The usage of the assembler is similar under all of available operating systems. All command-line options are recognised in either upper or lower case. The basic command format is shown:

```
ASPIC18 [ 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 4.1, and a full description of each option follows.

Table 4.1: ASPIC18 command-line options

Option	Meaning	Default
-A	Produce assembler output	Produce object code
-C	Produce cross-reference file	No cross reference
-C*chipinfo*	Define the chipinfo file	dat\picc-18.ini
-E[*file*|*digit*]	Set error destination/format	
-F*length*	Specify listing form length	66
-H	Output hex values for constants	Decimal values
-I	List macro expansions	Don't list macros
-L[*listfile*]	Produce listing	No listing
-O	Perform optimization	No optimization
-O*outfile*	Specify object name	srcfile.obj
-P*processor*	Define the processor	
-R	Specify non-standard ROM	
-T*width*	Specify listing page width	80
-V	Produce line number info	No line numbers
-W*level*	Set warning level threshold	0
-X	No local symbols in OBJ file	

4.2 Assembler Options

The command-line options recognised by ASPIC18 are as follows.

-A An assembler file with an extension `.opt` will be produced if this option is used. This is useful when checking the optimized assembly produced using the `-O` assembler option. Thus if both `-A` and `-O` are used with an assembly source file, the file will be optimized and rewritten, without the usual conversion to an object file.
The output file, when this option is used, is a valid assembly file that can be passed to the assembler. This differs to the assembly list file produced by the assembler when the -L assembler option is used.

-C A cross reference file will be produced when this option is used. This file, called *`srcfile`*`.crf`, where *`srcfile`* is the base portion of the first source file name, will contain raw cross reference information. The cross reference utility `CREF` must then be run to produce the formatted cross reference listing. See Section 4.7 for more information.

-C*chipinfo* Specify the chipinfo file to use. The chipinfo file is called `picc-18.ini` and can be found in the `DAT` directory of the compiler distribution.

-E[*file*|*digit*] 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 `-E` option with no argument will make the assembler use an alternate format for error and warning messages. Use of the option in this form has a similar effect as the same option used with command-line driver. See Section 2.4.31 for more information.
Specifying a digit as argument has a similar effect, only it allows selection of pre-set message formats.
Specifying a filename as argument will force the assembler to direct error and warning messages to a file with the name specified.

-F*length* By default when an assembly list file is requested (see assembler option `-L`), the listing format is pageless, i.e. the assembly listing output is continuous. The output may be formatted into pages of varying lengths. Each page will begin with a header and title, if specified. The `-F` option allows a page length to be specified. A zero value of *`length`* implies pageless output. The length is specified in a number of lines.

-H Particularly useful in conjunction with the `-A` or `-L ASPIC18` options, this option specifies that output constants should be shown 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.

-L*listfile* This option requests the generation of an assembly listing file. If `listfile` is specified then the listing will be written to that file, otherwise it will be written to the standard output. An assembly listing file contains additional fields, such as the address and opcode fields, which are not part of the assembly source syntax, hence these files cannot be passed to the assembler for compilation. See the assembler `-A` option for generating processed assembly source files that can be used as source files in subsequent compilation.

-O This requests the assembler to perform optimization on the assembly code. Note that the use of this option slows the assembly process down, as the assembler must make an additional pass over the input code. Debug information for assembler code generated from C source code may become unreliable.

-O*outfile* 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 filename and appending `.obj`. The `-O` option allows the user to override the default filename and specify a new name for the object file.

-P*processor* This option defines the processor which is being used. The processor type can also be indicated by use of the `PROCESSOR` directive in the assembler source file, see Section 4.3.8.24. You can also add your own processors to the compiler via the compiler's chipinfo file.

-V This option will include line number and filename information in the object file produced by the assembler. Such information may be used by debuggers. Note that the line numbers will correspond with assembler code lines in the assembler file. This option should not be used when assembling an assembler file produced by the code generator from a C source file, i.e. it should only be used with hand-written assembler source files.

-T*width* This option allows specification of the assembly list file width, in characters. `width` should be a decimal number greater than 41. The default width is 80 characters.

-W[!]*warnlevel* This option allow the warning threshold level to be set. This will limit the number of warning messages produce when the assembler is executing. The effect of this option is similar to the command-line driver's `--WARN` option, see Section 2.4.54.

-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` assembler option will prevent the local symbols from being included in the object file, thereby reducing the file size.

4.3 HI-TECH C Assembly Language

The source language accepted by the macro assembler, `ASPIC18`, is described below. All opcode mnemonics and operand syntax are strictly PIC18 assembly language. Additional mnemonics and

Table 4.2: ASPIC18 statement formats

Format 1	*label*:			
Format 2	*label*:	*mnemonic*	*operands*	; *comment*
Format 3	*name*	*pseudo-op*	*operands*	; *comment*
Format 4	; *comment only*			
Format 5	<empty line>			

assembler directives are documented in this section.

4.3.1 Statement Formats

Legal statement formats are shown in Table 4.2.

The `label` field is optional and, if present, should contain one identifier. A label may appear on a line of its own, or precede a mnemonic as shown in the second format.

The third format is only legal with certain assembler directives, such as `MACRO`, `SET` and `EQU`. The `name` field is mandatory and should also contain one identifier.

If the assembly file is first processed by the C preprocessor, see Section 2.4.12, then it may also contain lines that form valid preprocessor directives. See Section 3.12.1 for more information on the format for these directives.

There is no limitation on what column or part of the line in which any part of the statement should appear.

4.3.2 Characters

The character set used is standard 7 bit ASCII. Alphabetic case is significant for identifiers, but not mnemonics and reserved words. *Tabs* are treated as equivalent to *spaces*.

4.3.2.1 Delimiters

All numbers and identifiers must be delimited by *white space*, non-alphanumeric characters or the end of a line.

4.3.2.2 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.

Table 4.3: ASPIC18 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 `0x` or followed by `H` *or* `h`

4.3.3 Comments

An assembly comment is initiated with a *semicolon* that is not part of a string or character constant.

If the assembly file is first processed by the C preprocessor, see Section 2.4.12, then it may also contain C or C++ style comments using the standard `/* ... */` and `//` syntax.

4.3.3.1 Special Comment Strings

Several comment strings are appended to assembler instructions by the code generator. These are typically used by the assembler optimizer.

The comment string `;volatile` is used to indicate that the memory location being accessed in the commented instruction is associated with a variable that was declared as `volatile` in the C source code. Accesses to this location which appear to be redundant will not be removed by the assembler optimizer if this string is present.

This comment string may also be used in assembler source to achive the same effect for locations defined and accessed in assembly code.

4.3.4 Constants

4.3.4.1 Numeric Constants

The assembler performs all arithmetic with signed 32-bit precision.

The default radix for all numbers is 10. Other radices may be specified by a trailing base specifier as given in Table 4.3.

Hexadecimal numbers must have a leading digit (e.g. 0ffffh) to differentiate them from identifiers. Hexadecimal digits 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.

4.3.4.2 Character Constants and Strings

A character constant is a single character enclosed in *single quotes* `'` .

Multi-character constants, or strings, are a sequence of characters, not including *carriage return* or *newline* characters, enclosed within matching quotes. Either *single quotes* `'` or *double quotes* `"` maybe used, but the opening and closing quotes must be the same.

4.3.5 Identifiers

Assembly 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 alphabetics 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
```

4.3.5.1 Significance of Identifiers

Users of other assemblers that attempt to implement forms of data typing for identifiers should note that this assembler attaches no significance to any symbol, and places no restrictions or expectations on the usage of a symbol.

The names of *psects* (program sections) and ordinary symbols occupy separate, overlapping name spaces, but other than this, the assembler does not care whether a symbol is used to represent bytes, words or sports cars. 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.

4.3.5.2 Assembler-Generated Identifiers

Where a LOCAL directive is used in a macro block, the assembler will generate a unique symbol to replace each specified identifier in each expansion of that macro. These unique symbols will have the form `??`*nnnn* where *nnnn* is a 4 digit number. The user should avoid defining symbols with the same form.

4.3.5.3 Location Counter

The current location within the active program section is accessible via the symbol `$`. This symbol expands to the address of the currently executing instruction. Thus:

```
goto $
```

will represent code that will jump to itself and form an endless loop. By using this symbol and an offset, a relative jump destination to be specified.

The address represented by `$` is a word address and thus any offset to this symbol represents a number of instructions. For example:

```
goto $+1
movlw 8
movwf _foo
```

will skip one instruction.

4.3.5.4 Register Symbols

Code in assembly modules may gain access to the special function registers by including pre-defined assembly header files. The appropriate file can be included by add the line:

```
#include <aspic18.h>
```

to the assembler source file. Note that the file must be included using a C pre-processor directive and hence the option to pre-process assembly files must be enabled when compiling, see Section 2.4.12. This header file contains appropriate commands to ensure that the header file specific for the target device is included into the souce file.

These header files contain `EQU` declarations for all byte or multi-byte sized registers and `#define` macros for named bits within byte registers.

4.3.5.5 Symbolic Labels

A label is symbolic alias which is assigned a value equal to its offset within the current psect.

A label definition consists of any valid assembly identifier followed by a *colon*, `:`. The definition may appear on a line by itself or be positioned before a statement. Here are two examples of legitimate labels interspersed with assembly code.

```
frank:
            movlw 1
            goto fin
simon44:    clrf _input
```

Here, the label `frank` will ultimately be assigned the address of the `mov` instruction, and `simon44` the address of the `clrf` instruction. Regardless of how they are defined, the assembler list file produced by the assembler will always show labels on a line by themselves.

Labels may be used (and are prefered) in assembly code rather than using an absolute address. Thus they can be used as the target location for jump-type instructions or to load an address into a register.

Like variables, labels have scope. By default, they may be used anywhere in the module in which they are defined. They may be used by code above their definition. To make a label accessable in other modules, use the GLOBAL directive. See Section 4.3.8.1 for more information.

4.3.6 Expressions

The operands to instructions and directives are comprised of expressions. Expressions can be made up of numbers, identifiers, 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 4.4. 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.

4.3.7 Program Sections

Program sections, or *psects*, are simply a section of code or data. They are a way of grouping together parts of a program (via the psect's name) even though the source code may not be physically adjacent in the source file, or even where spread over several source files.

The concept of a program section is not a HI-TECH-only feature. Often referred to as blocks or segments in other compilers, these grouping of code and data have long used the names `text`, `bss` and `data`.

A psect is identified by a name and has several attributes. The `PSECT` assembler directive is used to define a psect. It takes as arguments a name and an optional comma-separated list of flags. See Section 4.3.8.3 for full information on psect definitions. Chapter 5 has more information on the operation of the linker and on optins that can be used to control psect placement in memory.

The assembler associates no significance to the name of a psect and the linker is also not aware of which are compiler-generated or user-defined psects. Unless defined as `abs` (absolute), psects are relocatable.

Table 4.4: ASPIC18 operators

Operator	Purpose	Example
*	Multiplication	`movlw 4*33`
+	Addition	`bra $+1`
-	Subtraction	`DB 5-2`
/	Division	`movlw 100/4`
= or eq	Equality	`IF inp eq 66`
> or gt	Signed greater than	`IF inp > 40`
>= or ge	Signed greater than or equal to	`IF inp ge 66`
< or lt	Signed less than	`IF inp < 40`
<= or le	Signed less than or equal to	`IF inp le 66`
<> or ne	Signed not equal to	`IF inp <> 40`
low	Low byte of operand	`movlw low(inp)`
high	High byte of operand	`movlw high(1008h)`
highword	High 16 bits of operand	`DW highword(inp)`
mod	Modulus	`movlw 77 mod 4`
&	Bitwise AND	`clrf inp&0ffh`
^	Bitwise XOR (exclusive or)	`movlw inp^80`
|	Bitwise OR	`movlw inp|1`
not	Bitwise complement	`movlw not 055h`
< < or shl	Shift left	`DB inp><8`
> > or shr	Shift right	`movlw inp shr 2`
rol	Rotate left	`DB inp rol 1`
ror	Rotate right	`DB inp ror 1`
float24	24-bit version of real operand	`DW float24(3.3)`
nul	Tests if macro argument is null	

The following is an example showing some executable instructions being placed in the `text` psect, and some data being placed in the `rbss` psect.

```
PSECT text,class=CODE
adjust:
   goto  clear_fred
increment:
   incf _fred
PSECT bss,class=BANK0,space=1
fred:
   DS 2
PSECT text,class=CODE
clear_fred:
   clrf _fred
   return
```

Note that even though the two blocks of code in the `text` psect are separated by a block in the `bss` psect, the two `text` psect blocks will be contiguous when loaded by the linker. In other words, the `incf _fred` instruction will be followed by the `clrf` instruction in the final ouptut. The actual location in memory of the `text` and `bss` psects will be determined by the linker.

Code or data that is not explicitly placed into a psect will become part of the default (unnamed) psect.

4.3.8 Assembler Directives

Assembler *directives*, or *pseudo-ops*, are used in a similar way to instruction mnemonics, but either do not generate code, or generate non-executable code, i.e. data bytes. The directives are listed in Table 4.5, and are detailed below.

4.3.8.1 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
```

Table 4.5: ASPIC18 assembler directives

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
SET	Define or re-define symbol value
DB	Define constant byte(s)
DW	Define constant word(s)
DS	Reserve storage
IF	Conditional assembly
ELSIF	Alternate conditional assembly
ELSE	Alternate conditional assembly
ENDIF	End conditional assembly
FNADDR	Inform the linker that a function may be indirectly called
FNARG	Inform the linker that evaluation of arguments for one function requires calling another
FNBREAK	Break call graph links
FNCALL	Inform the linker that one function calls another
FNCONF	Supply call graph configuration information for the linker
FNINDIR	Inform the linker that all functions with a particular signature may be indirectly called
FNROOT	Inform the linker that a function is the "root" of a call grpah
FNSIZE	Inform the linker of argument and local variable 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 assembled.
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

Table 4.6: PSECT flags

Flag	Meaning
abs	Psect is absolute
bit	Psect holds bit objects
class=*name*	Specify class name for psect
delta=*size*	Size of an addressing unit
global	Psect is global (default)
limit=*address*	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=*boundary*	Start psect on specified boundary
size=*max*	Maximum size of psect
space=*area*	Represents area in which psect will reside
with=*psect*	Place psect in the same page as specified psect

4.3.8.2 END

END is optional, but if present should be at the very end of the program. It will terminate the assembly and not even blank lines should follow this directive. 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
```

4.3.8.3 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 4.6, below.

Once a psect has been declared it may be resumed later by another PSECT directive, however the flags need not be repeated.

- 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.

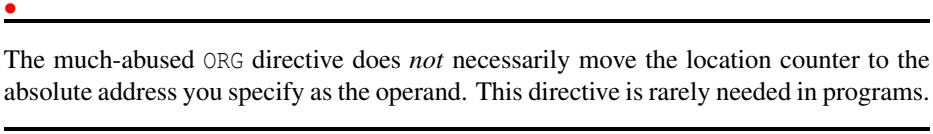
- The `class` flag specifies a class name for this psect. Class names are used to allow local psects to be referred to by a class name at link time, since they cannot be referred to by their own name. Class names are also useful where psects need only be positioned anywhere within a range of addresses rather than at one specific address.
- The `delta` flag defines the size of an addressing unit. In other words, the number of bytes covered for an increment in the address.
- A psect defined as `global` will be combined with other `global` psects of the same name from other modules at link time. This is the default behaviour for psects, unless the `local` flag is used.
- 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. Where there are two `local` psects in the one module, they reference the same psect. 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 runtime. `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 runtime 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 program memory and data memory may have a different `space` value to indicate that the program space address zero, for example, is a different location to the data memory address zero. Devices which use banked RAM data memory typically 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=text` will specify that this psect should be placed in the same page as the `text` 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
```

4.3.8.4 ORG

The ORG directive changes the value of the location counter within the current psect. This means that the addresses set with ORG are relative to the base address of the psect, which is not determined until link time.



The much-abused ORG directive does *not* necessarily move the location counter to the absolute address you specify as the operand. This directive is rarely needed in programs.

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 determined by the argument. It is not possible to move the location counter backward. For example:

```
ORG 100h
```

will move the location counter to the beginning of the current psect plus 100h. The actual location will not be known until link time.

In order to use the ORG directive to set the location counter to an absolute value, the directive must be used from within an absolute, overlaid psect. For example:

```
PSECT absdata,abs,ovrld
    ORG 50h
```

4.3.8.5 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 Section 4.3.8.6.

4.3.8.6 SET

This pseudo-op is equivalent to EQU except that allows a symbol to be re-defined. For example

```
thomas SET 0h
```

4.3.8.7 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. 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.

4.3.8.8 DW

DW operates in a similar fashion to DB, except that it assembles expressions into words. Example:

```
DW -1, 3664h, 'A', 3777Q
```

4.3.8.9 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
```

4.3.8.10 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.

4.3.8.11 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().

4.3.8.12 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.

4.3.8.13 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
```

4.3.8.14 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 startup code used by C compiler generated code. For the HI-TECH PICC-18 Compiler the startup routine will 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 ?.

4.3.8.15 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
```

4.3.8.16 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.

4.3.8.17 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
```

4.3.8.18 IF, ELSIF, ELSE and ENDIF

These directives implement conditional assembly. The argument to `IF` and `ELSIF` should be an absolute expression. If it is non-zero, then the code following it up to the next matching `ELSE`, `ELSIF` or `ENDIF` will be assembled. If the expression is zero then the code up to the next matching `ELSE` or `ENDIF` 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
   goto aardvark
ELSIF DEF
   goto denver
ELSE
   goto grapes
ENDIF
```

In this example, if `ABC` is non-zero, the first `jmp` instruction will be assembled but not the second or third. If `ABC` is zero and `DEF` is non-zero, the second `jmp` will be assembled but the first and third will not. If both `ABC` and `DEF` are zero, the third `jmp` will be assembled. Conditional assembly blocks may be nested.

4.3.8.19 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: storem
;args:  arg1 - the NAME of the source variable
;       arg2 - the literal value to load
```

```
;descr: Loads two registers with the value in the variable:
ldtwo   MACRO   arg1,arg2
   movlw &arg2
   movwf &arg1
ENDM
```

When used, this macro will expand to the 2 instructions in the body of the macro, with the formal parameters substituted by the arguments. Thus:

```
storem tempvar,2
```

expands to:

```
movlw 2
movwf tempvar
```

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.

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*, `;;`.

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.

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
```

By default, the assembly list file will show macro in an unexpanded format, i.e. as the macro was invoked. Expansion of the macro in the listing file can be shown by using the `EXPAND` assembler control, see Section 4.3.9.2,

4.3.8.20 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
   more: decfsz count
   goto more
ENDM
```

when expanded will include a unique assembler generated label in place of more. For example:

```
down foobar
```

expands to:

```
??0001 decfsz foobar
       goto ??0001
```

if invoked a second time, the label more would expand to ??0002.

4.3.8.21 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 it specifies 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.

4.3.8.22 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,w
ENDM
```

will expand to

```
addwf fred,w
addwf fred,w
addwf fred,w
```

4.3.8.23 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 romdata,class=CODE,delta=2
    IRPC char,ABC
    DB 'char'
ENDM
PSECT text
```

will expand to:

```
PSECT romdata,class=CODE,delta=2
    DB 'A'
    DB 'B'
    DB 'C'
PSECT text
```

4.3.8.24 PROCESSOR

The output of the assembler may vary depending on the target device. The device name is typically set using the `--CHIP` option to the command-line driver PICC18, see Section 2.4.22, or using the assembler `-P` option, see Table 4.1, but can also be set with this directive, e.g.

```
PROCESSOR 16F877
```

4.3.8.25 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 C 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 the symbol `_fred`. If a different signature value for `_fred` is present in any object file, the linker will report an error.

4.3.9 Assembler Controls

Assembler controls may be included in the assembler source to control assembler operation such as listing format. These keywords have no significance anywhere else in the program. The control is invoked by the directive `OPT` followed by the control name. Some keywords are followed by one or more parameters. For example:

```
OPT EXPAND
```

A list of keywords is given in Table 4.7, and each is described further below.

Table 4.7: DSPIC assembler controls

Control[1]	Meaning	Format
COND*	Include conditional code in the listing	COND
EXPAND	Expand macros in the listing output	EXPAND
INCLUDE	Textually include another source file	INCLUDE <*pathname*>
LIST*	Define options for listing output	LIST [<*listopt*>, ..., <*listopt*>]
NOCOND	Leave conditional code out of the listing	NOCOND
NOEXPAND*	Disable macro expansion	NOEXPAND
NOLIST	Disable listing output	NOLIST
PAGE	Start a new page in the listing output	PAGE
SUBTITLE	Specify the subtitle of the program	SUBTITLE "<*subtitle*>"
TITLE	Specify the title of the program	TITLE "<*title*>"

4.3.9.1 COND

Any conditional code will be included in the listing output. See also the NOCOND control in Section 4.3.9.5.

4.3.9.2 EXPAND

When EXPAND is in effect, the code generated by macro expansions will appear in the listing output. See also the NOEXPAND control in Section 4.3.9.6.

4.3.9.3 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, for example:

```
OPT INCLUDE "options.h"
```

The driver does not pass any search paths to the assembler, so if the include file is not located in the working directory, the pathname must specify the exact location.

See also the driver option -P in Section 2.4.12 which forces the C preprocessor to preprocess assembly file, thus allowing use of preprocessor directives, such as #include (see Section 3.12.1).

4.3.9.4 LIST

If the listing was previously turned off using the NOLIST control, the LIST control on its own will turn the listing on.

Table 4.8: 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.

Alternatively, the LIST control may includes options to control the assembly and the listing. The options are listed in Table 4.8.

See also the NOLIST control in Section 4.3.9.7.

4.3.9.5 NOCOND

Using this control will prevent conditional code from being included in the listing output. See also the COND control in Section 4.3.9.1.

4.3.9.6 NOEXPAND

NOEXPAND disables macro expansion in the listing file. The macro call will be listed instead. See also the EXPAND control in Section 4.3.9.2. Assembly macro are discussed in Section 4.3.8.19.

4.3.9.7 NOLIST

This control turns the listing output off from this point onward. See also the LIST control in Section 4.3.9.4.

4.3.9.8 NOXREF

NOXREF will disable generation of the *raw* cross reference file. See also the XREF control in Section 4.3.9.13.

4.3.9.9 PAGE

PAGE causes a new page to be started in the listing output. A *Control-L* (*form feed*) character will also cause a new page when encountered in the source.

4.3.9.10 SPACE

The `SPACE` control will place a number of blank lines in the listing output as specified by its parameter.

4.3.9.11 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 in Section 4.3.9.12.

4.3.9.12 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 in Section 4.3.9.11.

4.3.9.13 XREF

`XREF` is equivalent to the driver command line option `--CR` (see Section 2.4.26). It causes the assembler to produce a raw cross reference file. The utility `CREF` should be used to actually generate the formatted cross-reference listing.

Chapter 5

Linker and Utilities

5.1 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 driver 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.

5.2 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.

5.3 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.

5.4 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 Section 4.3.8.3 for more information on `PSECT` options.

5.5 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.

5.6 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.

5.7 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 5.1 and discussed in the following paragraphs.

Table 5.1: Linker command-line options

Option	Effect
`-8`	Use 8086 style segment:offset address form
`-A`*`class=low-high,...`*	Specify address ranges for a class
`-C`*`x`*	Call graph options
continued...	

[1] In earlier versions of HI-TECH C the linker was called `LINK.EXE`

Table 5.1: Linker command-line options

Option	Effect
-C*psect=class*	Specify a class name for a global psect
-C*baseaddr*	Produce binary output file based at *baseaddr*
-D*class=delta*	Specify a class delta value
-D*symfile*	Produce old-style symbol file
-E*errfile*	Write error messages to *errfile*
-F	Produce `.obj` file with only symbol records
-G*spec*	Specify calculation for segment selectors
-H*symfile*	Generate symbol file
-H+*symfile*	Generate enhanced symbol file
-I	Ignore undefined symbols
-J*num*	Set maximum number of errors before aborting
-K	Prevent overlaying function parameter and auto areas
-L	Preserve relocation items in `.obj` file
-LM	Preserve segment relocation items in `.obj` file
-N	Sort symbol table in map file by address order
-Nc	Sort symbol table in map file by class address order
-Ns	Sort symbol table in map file by space address order
-M*mapfile*	Generate a link map in the named file
-O*outfile*	Specify name of output file
-P*spec*	Specify psect addresses and ordering
-Q*processor*	Specify the processor type (for cosmetic reasons only)
-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 (-9 to 9)
-W*width*	Set map file width (>=10)
-X	Remove any local symbols from the symbol file
-Z	Remove trivial local symbols from the symbol file

5.7.1 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.

5.7.2 -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.

5.7.3 -C*x*

These options allow control over the call graph information which may be included in the map file produced by the linker. There are four varients of this option:

Fully expanded callgraph The `-Cf` option displays the full callgraph information.

Short form callgraph The `-Cs` option is the default callgraph option which removes some redundant information from the callgraph display. In the case where there are parameters to a function that involve function calls, the callgraph information associated with the "ARG function" is only shown the first time it is encountered in the callgraph. See Sections 5.9.1 and 5.10.2.2 for more information on these functions.

Critical path callgraph 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. See Section 5.10.2.2 for more information on critcal paths.

No callgraph The `-Cn` option removes the call graph information from the map file.

5.7.4 -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.

5.7.5 -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.

5.7.6 -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.

5.7.7 -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.

5.7.8 -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.

5.7.9 -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=` flag 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.

5.7.10 -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`.

5.7.11 -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`.

5.7.12 -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.

5.7.13 -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.

5.7.14 -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.

5.7.15 -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.

5.7.16 -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.

5.7.17 -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 5.10.

5.7.18 -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.

5.7.19 -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.

5.7.20 -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 ascending 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.

5.7.21 -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.

5.7.22 -S

This option prevents symbol information relating from being included in the symbol file produced by the linker. Segment information is still included.

5.7.23 -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
```

5.7.24 -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.

5.7.25 -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.

5.7.26 -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.

5.7.27 -X

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

5.7.28 -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.

5.8 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
```

5.9 Compiled Stack Operation

A compiler can either take advantage of the hardware stack contained on a device, or produce code which uses a *compiled stack* for parameter passing between functions and auto variables. Temporary variables used by a function may also be allocated space in the auto area. (Temporary variables with names like btemp, wtemp or ltemp are *not* examples of such variables. These variables are treated more like registers, although they may be allocated memory.) A compiled stack consists of fixed memory areas that are usable by each function's auto and parameter variables. When a compiled stack is used, functions are not re-entrant since local variables in each function will use the same fixed area of memory every time the function is invoked.

Fundamental to the compiled stack is the call graph which defines a tree-like hierarchy indicating the structure of function calls. The call graph consists of one or more *call trees* which are defined by the program. Each tree has a *root function*, which is typically not called by the program, but which is executed via other means. The function main is an example of a root function. Interrupt functions are another. The term *main-line code* means any code that is executed, or may be executed, by a function that appears under the main root in the call graph. See Section 5.10.2.2 for detailed information on the call graph which is displayed in the map file.

Each function in the call graph is allocated an *auto/parameter block* (APB) for its parameter, auto and temporary variables. Temporary variables act just like auto variables. Local variables which are qualified static are not part of this block. For situations where a compiled stack is used,

the linker performs additional operations to minimise the memory consumed by the program by overlaying each function's APB where possible.

In assembly code variables within a function's APB are referenced via special symbols, which marks the start of the auto or parameter area in the block, and an offset. The symbol used to represent the base address of the parameter area within the function's APB is the concatenation of `?` and the assembler name of the function. The symbol used to represent the base address of the auto area within the function's APB is the concatenation of `?a`, in the case of Standard version compilers, or `??`, in the case of PRO version compilers, and the assembler name of the function.

For example, a function called `foo`, for example, will use the assembly symbol `?_foo` as the base address for all its parameters variables that have been allocated memory, and either `?a_foo` (Standard) or `??_foo` (PRO) as the base address for `auto` variables which the function defines. So the first two-byte `auto` variable might be referenced in PRO version compiler assembly code as `??_foo`; the second `auto` variable as `??_foo+2`, etc. Note that some parameters may be passed in registers, and may not have memory allocated to them in the parameter area of the APB.

The linker allocates memory for each function's APB, based on how that function is used in a program. In particular, the linker determines which functions are, or may be, active at the same time. If one function calls another, then both are active at the same time. To this end, a call graph is created from information in the object files being linker. See Section 5.10.2.2 for information on reading the call graph displayed in the map file. This information is directly related to the `FNCALL` assembler directive (see Section 4.3.8.13 for more information) which the code generator places in the assembler output whenever a C function calls another. Hand-written assembler code should also contain these directives, if required. Information regarding the size of the auto and parameter areas within in function's APB is specified by the `FNSIZE` assembler directive (see Section 4.3.8.16).

5.9.1 Parameters involving Function Calls

The linker must take special note of the results of function calls used in expressions that are themselves parameters to another function. For example, if `input` and `output` are both functions that accept two `int` parameters and and both return an `int`, the following:

```
result = output(out_selector, input(int_selector, 10));
```

shows that the function `input` is called to determine the second parameter to the function `output`. This information is very important as it indicates areas of the code that must be considered carefully, lest the code fail due to re-entrancy related issues.

A re-entrant call is typically considered to be the situation in which a function is called and executed while another instance of the same function is also actively executing. For a compiled stack program, a function must be considered active as soon as its parameter area has been modified in preparation for a call, even though code in that function is not yet being executed and a call to that function has not been made. This is particularly import with functions that accept more than

one parameter as the ANSI standard does not dictate the order in which function parameters must be evaluated.

Such a condition is best illustrated by an example, which is shown in the following tutorial.

TUTORIAL

PARAMETERS IMPLEMENTED AS FUNCTION CALLS Consider the following code.

```
int B(int x, int y) {
  return x - y;
}
int A(int a, int b) {
  return a+B(9, b);
}
void main(void) {
  B(5, A(6, 7));  // consider this statement
}
```

For the highlighted statement, the compiler *might* evaluate and load the first parameter to the function `B`, which is the literal, `5`. To do this, the value of `5` is loaded to the locations `?_B` and `?_B+1`. Now to evaluate the second parameter value to the function `B`, the compiler must first call the function `A`. So `A`'s parameters are loaded and the call to function `A` is made. Code inside the function `A`, calls the function `B`. This involves loading the parameters to `B`: the contents of the variable `b` are loaded to `?_B+2` and `?_B+3`, and the value `9` is loaded to `?_B` and `?_B+1`, which corrupts the contents of these locations which were loaded earlier for the still pending call to function `B`. Function `A` eventually returns normally and the the return value is the loaded to the second parameter locations for the still pending call to function `B`, back at the highlighted line of source. However, the value of `5` previously loaded as the first parameter to `B` has been lost. When the call to function `B` is now made, the parameters will not be correct.
Note that the function `B` is not actively executing code in more than one instance of the function at the same time, however the code that loads the parameters to function B is.

The linker indicates in the call graph those functions that may have been called to determine parameter values to other functions. See Section 5.10.2.2 for information on how this is displayed in the map file.

5.10 Map Files

The map file contains information relating to the relocation of psects and the addresses assigned to symbols within those psects.

5.10.1 Generation

If compilation is being performed via HI-TIDE™ a map file is generated by default without you having to adjust the compiler options. If you are using the driver from the command line then you'll need to use the `-M` option, see Section 2.4.9.

Map files are produced by the linker. If the compilation process is stopped before the linker is executed, then no map file is produced. The linker will still produce a map file even if it encounters errors, which will allow you to use this file to track down the cause of the errors. However, if the linker ultimately reports `too many errors` then it did not run to completion, and the map file will be either not created or not complete. You can use the `--ERRORS` option on the command line, or as an alternate MPLAB IDE setting, to increase the number of errors before the compiler applications give up. See Section 2.4.32 for more information on this option.

5.10.2 Contents

The sections in the map file, in order of appearance, are as follows:

- The compiler name and version number;
- A copy of the command line used to invoke the linker;
- The version number of the object code in the first file linked;
- The machine type;
- Optionally (dependent on the processor and compiler options selected), the call graph information;
- A psect summary sorted by the psect's parent object file;
- A psect summary sorted by the psect's CLASS;
- A segment summary;
- Unused address ranges summary; and
- The symbol table

Portions of an example map file, along with explanatory text, are shown in the following sections.

5.10.2.1 General Information

At the top of the map file is general information relating to the execution of the linker.

When analysing a program, always confirm the compiler version number shown in the map file if you have more than one compiler version installed to ensure the desired compiler is being executed.

The chip selected with the `--CHIP` option should appear after the *Machine type* entry.

The *Object code version* relates to the file format used by relocatable object files produced by the assembler. Unless either the assembler or linker have been updated independently, this should not be of concern.

A typical map file may begin something like the following. This example has been cut down for clarity and brevity, and should not be used for reference.

```
HI-TECH Software PICC Compiler std#V9.60
Linker command line:
--edf=C:\Program Files\HI-TECH Software\pic\std\9.60\dat\en_msgs.txt \
-h+conv.sym -z -Q16F73 -ol.obj -Mconv.map -ver=PICC#std#V9.60 \
-ACODE=00h-07FFhx2 -ACONST=00h-0FFhx16 -ASTRING=00h-0FFhx16 \
-ABANK0=020h-07Fh -ABANK1=0A0h-0FFh \
-preset_vec=00h,intentry,intcode -ppowerup=CODE -pintsave_0=07Fh \
-prbit_0=BANK0,rbss_0=BANK0,rdata_0=BANK0,idata_0=CODE \
C:\DOCUME~1\user\LOCALS~1\Temp\cgta5eHNF.obj conv.obj \
C:\Program Files\HI-TECH Software\pic\std\9.60\lib\pic412-c.lib \
C:\Program Files\HI-TECH Software\pic\std\9.60\lib\pic20--u.lib
Object code version is 3.9
Machine type is 16F73
```

The *Linker command line* shown is the entire list of options and files that were passed to the linker for the build recorded by this map file. Remember, these are linker options and not command-line driver options. Typically the first options relate to general execution of the linker: path and file names for various input and output support files; and the chip type etc. These are followed by the memory allocation options, e.g. `-A` and `-p`. Last are the input object and library files that will be linked to form the output.

The linker command line should be used to confirm that driver options that control the link step have been specified correctly, and at the correct time. It is particularly useful when using the driver `-L-` option, see Section 2.4.8.

TUTORIAL

CONFIRMING LINKER OPERATION A project requires that a number of memory locations be reserved. For the compiler and target device used by the project, the `--ROM` driver option is suitable for this task. How can the operation of this option be confirmed?

First the program is compiled without using this option and the following linker class definition is noted in the linker command line:

```
-ACODE=0-03FFFhx2
```

The class name may vary between compilers and the selected target device, however there is typically a class that is defined to cover the entire memory space used by the device.
The driver option `--ROM=default,-4000-400F` is then used and the map file resulting from the subsequent build shows the following change:

```
-ACODE=0-03FFFh,04010h-07FFFh
```

which confirms that the memory option was seen by the linker and that the memory requested was reserved.

5.10.2.2 Call Graph Information

A *call graph* is produced and displayed in the map file for target devices and memory models that use a compiled stack to facilitate parameter passing between functions and `auto` variables. See Section 5.9 for more detailed information on compiled stack operation.

The call graph in the map file shows the information collated and interpreted by the linker, which is primarily used to allow overlapping of functions' APBs. The following information can be obtained from studying the call graph:

- The functions in the program that are "root" nodes marking the top of a call tree, and which are not directly called;
- The functions that the linker deemed were called, or may have been called, during program execution;
- The program's hierarchy of function calls;
- The size of the auto and parameter areas within each function's APB;
- The offset of each function's APB within the program's auto/parameter psect;
- Which functions' APBs are consuming memory not overlapped by the APB of any other function (on the critical path);
- Which functions are called indirectly;
- Which functions are called as part of a parameter expression for another function; and

- The estimated call tree depth.

These features are discussed below.

The call graph produced by PRO versions compilers is very similar to that produced by Standard version compilers, however there are differences. A typical PRO compiler call graph may look something like:

```
Call graph:
*_main size 0,4 offset 0
*    _byteconv size 0,17 offset 4
         float size 3,7 offset 21
         ldiv size 8,6 offset 21
         _crv ARG size 0 offset 21
             _crv size 1 offset 21
             ldiv size 8,6 offset 21
             _convert size 4,0 offset 33
             _srv size 2,10 offset 21
                 _convert size 4,0 offset 33
*        _srv size 2,10 offset 21
*            _convert size 4,0 offset 33
     _init size 0,4 offset 4
     indir_func size 0,0 offset 4
Estimated maximum call depth: 3
*intlevel1 size 0,0 offset 37
*    _isr size 0,2 offset 37
*        i1ldiv size 8,6 offset 44
Estimated maximum call depth: 2
```

Each line basically consists of the name of the function in question, and its APB size and offset. The general form of most entries look like:

name size *p*,*a* offset *n*

Note that the function *name* will always be the assembly name, thus the function `main` appears as `_main`.

A function printed with no indent is a *root function* in a call tree. These functions are typically not called by the C program. Examples include the function `main`, any any `interrupt` functions the program defines. The programmer may also define additional functions that are root functions in the call tree by using the `FNROOT` assembler directive, see Section 4.3.8.17 for more information. The code generator issues an `FNROOT` directive for each `interrupt` function encountered, and the runtime startup code contains the `FNROOT` directive for the function `main`.

The functions that the root function calls, or *may* call, are indented one level and listed below the root node. If any of these functions call (or might call) other functions, these called functions are indented and listed below the calling functions. And so the process continues for entire program. A function's inclusion into the call graph does not imply the function was called, but there is a possibility that the function was called. For example, code such as:

```
int test(int a) {
  if(a)
    foo();
  else
    bar();
}
```

will list `foo` and `bar` under `test`, as either may be called. If `a` is always true, then clearly the function `bar` will never be called. If a function does not appear in the call graph, the linker has determined that the function cannot possibly be called, and that it is not a root function. For code like:

```
int test(void) {
  int a = 0;
  if(a)
    foo();
  else
    bar();
}
```

the function `foo` will never appear in the call graph.

The inclusion of a function into the call graph is controlled by the `FNCALL` assembler directive, see Section 4.3.8.13 for more information. These directives are placed in the assembler output by the code generator. For the above code, the code generator optimiser will remove the redundant call to `bar` before the C source code conversion is performed, as so the `FNCALL` directive will not be present in the output file, hence not detectable by the linker. When writing assembler source code, the `FNCALL` assembler directive should always be used, particularly if the assembler routines define local `auto`-like variables using the `FNSIZE` directive, see below, and also Section 4.3.8.16 for more information.

If printed, the two components to the *size* are the size of that function's parameter area, and the size of the function's auto area, respectively. The parameter size only includes those parameters which are allocated memory locations, and which are not passed via a register. The auto size does not include any `auto` variables which are allocated registers by the code generator's (global) optimizer for the entire duration of the function. The auto size does, however, include any values which must be stored temporarily in the functions scratch area. Variables which are passed via a register may

need to be saved into the function's temporary variable if that register is required for code generation purposes, in which case they do not contribute to the function's parameter size, but increase the size of the auto area.

The total parameter and auto area for each function is grouped to form an APB. This is then allocated an address within the program's auto/parameter psect. The *offset* value indicates the offset within the psect for that block. Thus, two APBs with the same offset are mapped over one another.

If a star, `*`, appears on the very left line of a call tree, this implies that the memory consumed by the function represented by that line does not fully overlap with that of other functions, and thus this functions APB directly influences the size of the auto/parameter psect, and hence the total RAM usage of the program. Such functions are said to be on the critical path. If the RAM usage of a program needs to be reduced and the number or size of the parameters or auto variables defined by the starred functions can be reduced, the program's RAM usage will also be reduced. Reducing the number or size of the parameters or auto variables defined by the functions that are not starred will have no effect on the program's total RAM usage.

PRO compilers track the values assigned to function pointers and maintains a list of all functions that could be called via the function pointer. Functions called indirectly are listed in the call graph along with those functions which are directly called.

If the *ARG* flag appears after a function's name, this implies that the call to this "ARG function" involves other function calls to determine the parameter values for this function. For example, if `input` and `output` are both functions that take two `int` parameters and and both return an `int`, the following:

```
result = output(out_selector, input(in_selector, 10));
```

shows that the function `input` is called to determine the second parameter to the function `output`.

The ARG function's name is listed again under the line which actually shows the ARG flag, and any functions this function calls appear here, indented in the usual way. Under this is listed *every* function (regardless of its depth in the call tree) that *could* be called to determine a parameter value to the ARG function throughout the program. If any of these functions call other functions, they also list called functions below, indented in the usual way. For example the following annotated call graph snippet illustrates the ARG function one.

```
_one ARG size 0 offset 21        ; _one is the ARG function
    _one size 0 offset 21        ; ** here is _one's call tree:
        _two size 2,2 offset 21  ; ** _one may call _two
    _prep1 size 1,1 offset 45    ; # _prep1, _get & _prep2 may
    _get  size 0,0 offset 47     ; # ultimately be called to
    _prep2 size 1,1 offset 47    ; # obtain parameters for _one
        _get  size 0,0 offset 47 ; _prep2 may call by _get
```

After each tree in call tree, there is an indication of the maximum call depth that might be realised by that tree. This may be used as a guide to the stack usage of the program. No definitive value can be given for the program's total stack usage for several reasons:

- Certain parts of the call tree may never be reached, reducing that tree's stack usage;
- The contribution of interrupt (or other) trees to the tree associated with the `main` function cannot be determined as the point in `main`'s call tree at which the interrupt (or other function invocation) will occur cannot be known;
- Any additional stack usage by functions, particularly interrupt functions, cannot be known; and
- The assembler optimizer may have replaced function calls with jumps to functions, reducing that tree's stack usage.

The code generator also produces a warning if the maximum stack depth appears to have been exceeded. For the above reasons, this warning, too, is intended to be a guide to potential stack problems.

The above call graph example is analysed in the following tutorial.

TUTORIAL

INTERPRETING A PRO COMPILER CALL GRAPH The graph graph shown above indicates that the program compiled consists of two call trees, rooted at the functions `main`, which can have up 3 levels of stack used, and `intlevel1`, which can use up to two levels of stack. In the example above, the symbol `_main` is associated with the function `main`, and `intlevel1` associated with an `interrupt` function (with an interrupt level of 1).

Here, the function `main` takes no parameters and defines 4 bytes of `auto` variables. The total size of the APB for main is 4, and this was placed at an offset of 0 in the program's auto/parameter psect. The function `main` may call a function called `init`. This function also uses a total of 4 bytes of `auto` variables. The function `main` is still active when `init` is active so their APBs must occupy distinct memory. (NB `main` will always be active during program execution, by definition.) The block for `init` follows immediately after that of `main`'s at address offset 4. The function `init` does not call any other functions.

The `main` function may also call the function `byteconv`. This function defines a total of 17 bytes of `auto` variables. It is called when main is still active, but it is never active at the same time as `init` is active, so its APB can overlap with that of `init` and is placed at offset 4 within the auto/parameter psect.

The function byteconv may call several functions: float, ldiv, crv and srv. (Any function name that does not start with an underscore must be an assembly routine. The routine float and ldiv in this case relating to floating point and long division library routines.) All these functions have their APB placed at the same offset in the auto/parameter psect. Of these functions, srv also may call convert.

The call to crv from byteconv indicates that other functions might be called to obtain crv's parameter values. Those other functions are listed in a "flattened" call list below the ARG function line which shows every possible function that might be called, regardless of call depth. The functions which might be called are: ldiv, convert and srv. The function srv, which also calls convert still indicates this fact by also listing convert below and indented in the more conventional call graph format. The two lines of C code that produced this outcome were:

```
if(crv((my_long%10)) != 5) // ...
if(crv(srv(8)) != 6) // ...
```

where crv accepts one char parameter and returns a char. The call to srv is obvious; the other call come from the modulus operator, calling ldiv.

The other call tree rooted at intlevel1 relates to the interrupt function. intlevel1 is not a real function, but is used to represent the interrupt level associated with the interrupt function. There is no call from intlevel1 to the function isr and no stack usage. Note that an additional level of call depth is indicated for interrupt functions. This is used to mark the place of the return address of the stack. The selected device may use a differing number of stack locations when interrupts occur and this needs to be factored into any stack calculations.

Notice that the interrupt function isr calls a function called i1ldiv. This is a duplicate of the ldiv routine that is callable by functions under the intlevel1 call tree. Having duplicate routines means that these implicitly called assembly library routines can safely be called from both code under the main call tree and code under the interrupt tree. PRO compilers will have as many duplicates of these routines as there are interrupt levels.

The call graph shows that the functions: main, byteconv, srv, convert, isr and i1ldiv are all consuming APB memory that does not fully overlap with that of other functions. Reducing the auto/parameter memory requirements for these functions will reduce the program's memory requirements. The call graph reveals that 82 bytes of memory are required by the program for autos and parameters, but that only 58 are reserved and used by the program. The difference shows the amount of memory saved by overlapping of these blocks by the linker.

5.10.2.3 Psect Information listed by Module

The next section in the map file lists those modules that made a contribution to the output, and information regarding the psects these modules defined.

This section is heralded by the line that contains the headings:

```
Name   Link  Load  Length  Selector  Space  Scale
```

Under this on the far left is a list of object files. These object files include both files generated from source modules and those that were extracted from object library files. In the case of those from library files, the name of the library file is printed before the object file list.

This section shows all the psects (under the *Name* column) that were linked into the program from each object file, and information regarding that psect. This only deals with object files linked by the linker. P-code modules derived from p-code library files are handled by the code generator, and do not appear in the map file.

The *Link* address indicates the address at which this psect will be located when the program is running. (The *Load* address is also shown for those psects that may reside in the HEX file at a different location and which are mapped before program execution.) The *Length* of the psect is shown (in units suitable for that psect). The *Selector* is less commonly used, but the *Space* field is important as it indicates the memory space in which the psect was placed. For Harvard architecture machines, with separate memory spaces, this field must be used in conjunction with the address to specify an exact storage location. The *Scale* of a psect indicates the number of address units per byte — this is left blank if the scale is 1 — and typically this will show 8 for psects that hold bit objects. The *Load* address of psects that hold bits is used to display the link address converted into units of bytes, rather than the load address.

TUTORIAL

INTERPRETING THE PSECT LIST The following appears in a map file.

```
            Name  Link  Load  Length Selector  Space  Scale
    ext.obj text    3A    3A      22       30      0
            bss     4B    4B      10       4B      1
            rbit    50     A       2        0      1      8
```

This indicates that one of the files that the linker processed was called `ext.obj`. (This may have been derived from `ext.c` or `ext.as`.) This object file contained a `text` psect, as well as psects called `bss` and `rbit`. The psect `text` was linked at address 3A and `bss` at address 4B. At first glance, this seems to be a problem given that text is 22 words long, however note that they are in different memory areas, as indicated by the *Space* flag (0 for `text` and 1 for `bss`), and so do not occupy the same memory. The psect

`rbit` contains bit objects, as indicated by its *Scale* value (its name is a bit of a giveaway too). Again, at first glance there seems there could be an issue with `rbit` linked over the top of `bss`. Their *Space* flags are the same, but since `rbit` contains bit objects, all the addresses shown are bit addresses, as indicated by the *Scale* value of 8. Note that the *Load* address field of `rbit` psect displays the *Link* address converted to byte units, i.e. 50h/8 => Ah.

The list of files, that make up the program, indicated in this section of the map file will typically consist of one or more object files derived from input source code. The map file produced by PRO compilers will show one object file derived from all C source modules, however Standard version compilers will show one object file per C source module.

In addition, there will typically be the runtime startup module. The runtime startup code is precompiled into an object file, in the case of Standard version compilers, or is a compiler-written assembler source file, which is then compiled along with the remainder of the program. In either case, an object file module will be listed in this section, along with those psects which it defines. If the startup module is not being deleted after compilation (see the `--RUNTIME` option in Section 2.4.47) then the module name will be `startup.obj`, otherwise this module will have a system-dependent temporary file name, stored in a system-dependent location.

Modules derived from library files area also shown in this list. The name of the library file is printed as a header, followed by a list of the modules that contributed to the output. Only modules that define symbols that are referenced are included in the program output. For example, the following:

```
C:\program files\HI-TECH Software\PICC-18\9.50\lib\pic86l-c.lib
i1aldiv.obj   text 174 174  3C  C  0
aldiv.obj     text  90  90  3C  C  0
```

indicates that both the `i1aldiv.obj` and `aldiv.obj` modules were linked in from the library file `pic86l-c.lib`.

Underneath the library file contributions, there may be a label `COMMON`. This shows the contribution to the program from program-wide psects, in particular that used by the compiled stack auto/parameter area.

This information in this section of the map file can be used to observe several details;

- To confirm that a module is making a contribution to the output file by ensuring that the module appears in the module list;
- To determine the exact psects that each module defines;

- For cases where a user-defined routine, with the same name as a library routine, is present in the programs source file list, to confirm that the user-defined routine was linked in preference to the library routine.

5.10.2.4 Psect Information listed by Class

The next section in the map file is the same psect information listed by module, but this time grouped into the psects' class.

This section is heralded by the line that contains the headings:

```
TOTAL   Name  Link  Load  Length
```

Under this are the class names followed by those psects which belong to this class. These psects are the same as those listed by module in the above section; there is no new information contained in this section.

5.10.2.5 Segment Listing

The class listing in the map file is followed by a listing of segments. A segment is conceptual grouping of contiguous psects, and are used by the linker as an aid in psect placement. There is no segment assembler directive and segments cannot be controlled in any way.

This section is heralded by the line that contains the headings:

```
SEGMENTS   Name  Load  Length  Top  Selector  Space  Class
```

The name of a segment is derived from the psect in the contiguous group with the lowest link address. This can lead to confusion with the psect with the same name. Do not read psect information from this section of the map file.

Typically this section of the map file can be ignored by the user.

5.10.2.6 Unused Address Ranges

The last of the memory summaries Just before the symbol table in the map file is a list of memory which was not allocated by the linker. This memory is thus unused. The linker is aware of any memory allocated by the code generator (for absolute variables), and so this free space is accurate.

This section follows the heading:

```
UNUSED ADDRESS RANGES
```

and is followed by a list of classes and the memory still available in each class defined in the program. If there is more than one range in a class, each range is printed on a separate line. Any paging boundaries within a class are ignored and not displayed in any way.

Note that classes often define memory that is also covered by other classes, thus the total free space in a memory area is not simply the addition of the size of all the ranges indicated. For example if there are two classes the cover the RAM memory — RAM and BANKRAM — and the first 100h out of 500h bytes are used, then both will indicate 000100-0004FF as the unused memory.

5.10.2.7 Symbol Table

The final section in the map file list global symbols that the program defines. This section has a heading:

```
Symbol Table
```

and is followed by two columns in which the symbols are alphabetically listed. As always with the linker, any C derived symbol is shown with its assembler equivalent symbol name. The symbols listed in this table are:

- Global assembly labels;
- Global `EQU`/`SET` assembler directive labels; and
- Linker-defined symbols.

Assembly symbols are made global via the `GLOBAL` assembler directive, see Section 4.3.8.1 for more information. linker-defined symbols act like `EQU` directives, however they are defined by the linker during the link process, and no definition for them will appear in any source or intermediate file.

Non-`static` C functions, and non-`auto` and non-`static` C variables directly map to assembly labels. The name of the label will be the C identifier with a leading *underscore* character. The linker-defined symbols include symbols used to mark the bounds of psects. See Section 3.13.3. The symbols used to mark the base address of each functions' auto and parameter block are also shown. Although these symbols are used to represent the local autos and parameters of a function, they themselves must be globally accessible to allow each calling function to load their contents. The C auto and parameter variable identifiers are local symbols that only have scope in the function in which they are defined.

Each symbol is shown with the psect in which they are placed, and the address which the symbol has been assigned. There is no information encoded into a symbol to indicate whether it represents code or variables, nor in which memory space it resides.

If the psect of a symbol is shown as `(abs)`, this implies that the symbol is not directly associated with a psect as is the case with absolute C variables. Linker-defined symbols showing this as the

psect name may be symbols that have never been used throughout the program, or relate to symbols that are not directly associated with a psect.

Note that a symbol table is also shown in each assembler list file. (See Section 2.4.18 for information on generating these files.) These differ to that shown in the map file in that they list all symbols, whether they be of global or local scope, and they only list the symbols used in the module(s) associated with that list file.

5.11 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.

5.11.1 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 can perform faster searches 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.

5.11.2 Using the Librarian

The librarian program is called LIBR, and the format of commands to it is as follows:

Table 5.2: Librarian command-line options

Option	Effect
`-P`*`width`*	specify page width
`-W`	Suppress non-fatal errors

Table 5.3: Librarian key letter commands

Key	Meaning
`r`	Replace modules
`d`	Delete modules
`x`	Extract modules
`m`	List modules
`s`	List modules with symbols

```
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 5.2.

The key letters are listed in Table 5.3.

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.

5.11.3 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 c.obj from the library file.lib:

```
LIBR d file.lib a.obj b.obj c.obj
```

5.11.4 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.

5.11.5 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.

5.11.6 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.

5.11.7 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.

5.12 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 5.4. Where an address is required, the format is the same as for HLINK.

5.12.1 Checksum Specifications

If you are generating a HEX file output, please refer to the hexmate section 5.15 for calculating checksums. For OBJTOHEX, the checksum specification allows automated checksum calculation and 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

Table 5.4: OBJTOHEX command-line options

Option	Meaning
-8	Produce a CP/M-86 output file
-A	Produce an ATDOS .atx output file
-B*base*	Produce a binary file with offset of *base*. Default file name is l.obj
-C*ckfile*	Read a list of checksum specifications from *ckfile* or standard input
-D	Produce a COD file
-E	Produce an MS-DOS .exe file
-F*fill*	Fill unused memory with 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	Produce 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
-*n*,*m*	Format either Motorola or Intel HEX file, where *n* is the maximum number of bytes per record and *m* specifies the record size rounding. Non-rounded records are zero padded to a multiple of *m*. *m* itself must be a multiple of 2.

and the sum placed in the locations *where1* through *where2* inclusive. For an 8 bit checksum these two addresses should be the same. For a checksum stored low byte first, where1 should be less than where2, and vice versa. The *+offset* is optional, but if supplied, the value offset will be used to initialise the checksum. Otherwise it is initialised to zero. For example:

```
0005-1FFF 3-4 +1FFF
```

This will sum the bytes in 5 through 1FFFH inclusive, then add 1FFFH to the sum. The 16 bit checksum will be placed in locations 3 and 4, low byte in 3. The checksum is initialised with 1FFFH to provide protection against an all zero ROM, or a ROM misplaced in memory. A run time check of this checksum would add the last address of the ROM being checksummed into the checksum. For the ROM in question, this should be 1FFFH. The initialization value may, however, be used in any desired fashion.

5.13 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 5.5.

Each option is described in more detail in the following paragraphs.

5.13.1 -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 `\`.

5.13.2 -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.

Table 5.5: CREF command-line 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 and symbols starting with *prefix*

5.13.3 -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.

5.13.4 -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.

5.13.5 -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.

5.13.6 -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.

5.13.7 -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

Table 5.6: CROMWELL 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
mcoff	Microchip COFF file format
omf51	OMF-51 file format
pe	P&E file format
s19	*Motorola* HEX file format

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.

5.14 Cromwell

The CROMWELL utility converts code and symbol files into different formats. The formats available are shown in Table 5.6.

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 5.7. *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.

5.14.1 -Pname[,architecture]

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. Note that to produce output in COFF format an additional

Table 5.7: CROMWELL command-line options

Option	Description
-P*name[,architecture]*	Processor name and architecture
-N	Identify code classes
-D	Dump input file
-C	Identify input files only
-F	Fake local symbols as global
-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

argument to this option which also specifies the processor architecture is required. Hence for this format the usage of this option must take the form: `-Pname,architecture`. Table 5.8 enumerates the architectures supported for producing COFF files.

5.14.2 -N

To produce some output file formats (e.g. COFF), Cromwell requires that the names of the program memory space psect classes be provided. The names of the classes are given as a comma separated list. For example, in the DSPIC C compiler these classes are typically "CODE" and "NEARCODE", i.e. `-NCODE,NEARCODE`.

5.14.3 -D

The `-D` option is used to display to the screen details about the named input file in a readable format. The input file can be one of the file types as shown in Table 5.6.

5.14.4 -C

This option will attempt to identify if the specified input files are one of the formats as shown in Table 5.6. If the file is recognised, a confirmation of its type will be displayed.

Table 5.8: `-P` option architecture arguments for COFF file output.

Architecture	Description
68K	Motorola 68000 series chips
H8/300	Hitachi 8 bit H8/300 chips
H8/300H	Hitachi 16 bit H8/300H chips
SH	Hitachi 32 bit SuperH RISC chips
PIC12	Microchip base-line PIC chips
PIC14	Microchip mid-range PIC chips
PIC16	Microchip high-end (17Cxxx) PIC chips
PIC18	Microchip PIC18 chips
PIC24	Microchip PIC24F and PIC24H chips
PIC30	Microchip dsPIC30 and dsPIC33 chips

5.14.5 -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.

5.14.6 -Okey

This option specifies the format of the output file. The *key* can be any of the types listed in Table 5.6.

5.14.7 -Ikey

This option can be used to specify the default input file format. The *key* can be any of the types listed in Table 5.6.

5.14.8 -L

Use this option to show what file format types are supported. A list similar to that given in Table 5.6 will be shown.

5.14.9 -E

Use this option to tell `CROMWELL` to ignore any filename extensions that were given. The default extension will be used instead.

5.14.10 -B

In formats that support different endian types, use this option to specify big-endian byte ordering.

5.14.11 -M

When generating COD files this option will remove the preceding *underscore* character from symbols.

5.14.12 -V

Turns on verbose mode which will display information about operations `CROMWELL` is performing.

5.15 Hexmate

The Hexmate utility is a program designed to manipulate Intel HEX files. Hexmate is a post-link stage utility that provides the facility to:

- Calculate and store variable-length checksum values
- Fill unused memory locations with known data sequences
- Merge multiple Intel hex files into one output file
- Convert INHX32 files to other INHX formats (e.g. INHX8M)
- Detect specific or partial opcode sequences within a hex file
- Find/replace specific or partial opcode sequences
- Provide a map of addresses used in a hex file
- Change or fix the length of data records in a hex file.
- Validate checksums within Intel hex files.

Typical applications for hexmate might include:

- Merging a bootloader or debug module into a main application at build time
- Calculating a checksum over a range of program memory and storing its value in program memory or EEPROM

- Filling unused memory locations with an instruction to send the PC to a known location if it gets lost.
- Storage of a serial number at a fixed address.
- Storage of a string (e.g. time stamp) at a fixed address.
- Store initial values at a particular memory address (e.g. initialise EEPROM)
- Detecting usage of a buggy/restricted instruction
- Adjusting hex file to meet requirements of particular bootloaders

5.15.1 Hexmate Command Line Options

Some of these hexmate operations may be possible from the compiler's command line driver. However, if hexmate is to be run directly, its usage is:

```
hexmate <file1.hex ... fileN.hex> <options>
```

Where *file1.hex* through to *fileN.hex* are a list of input Intel hex files to merge using hexmate. Additional options can be provided to further customize this process. Table 5.9 lists the command line options that hexmate accepts.

The input parameters to hexmate are now discussed in greater detail. Note that any integral values supplied to the hexmate options should be entered as hexadecimal values without leading `0x` or trailing `h` characters. Note also that any address fields specified in these options are to be entered as byte addresses, unless specified otherwise in the `-ADDRESSING` option.

5.15.1.1 specifications,filename.hex

Intel hex files that can be processed by hexmate should be in either INHX32 or INHX8M format. Additional specifications can be applied to each hex file to put restrictions or conditions on how this file should be processed. If any specifications are used they must precede the filename. The list of specifications will then be separated from the filename by a comma.

A *range restriction* can be applied with the specification `rStart-End`. A range restriction will cause only the address data falling within this range to be used. For example:

```
r100-1FF,myfile.hex
```

will use *myfile.hex* as input, but only process data which is addressed within the range *100h-1FFh* (inclusive) to be read from *myfile.hex*.

An *address shift* can be applied with the specification `sOffset` . If an address shift is used, data read from this hex file will be shifted (by the *Offset*) to a new address when generating the output. The offset can be either positive or negative. For example:

Table 5.9: Hexmate command-line options

Option	Effect
-ADDRESSING	Set address fields in all hexmate options to use word addressing or other
-BREAK	Break continuous data so that a new record begins at a set address
-CK	Calculate and store a checksum value
-FILL	Program unused locations with a known value
-FIND	Search and notify if a particular code sequence is detected
-FIND...,DELETE	Remove the code sequence if it is detected (use with caution)
-FIND...,REPLACE	Replace the code sequence with a new code sequence
-FORMAT	Specify maximum data record length or select INHX variant
-HELP	Show all options or display help message for specific option
-LOGFILE	Save hexmate analysis of output and various results to a file
-O*file*	Specify the name of the output file
-SERIAL	Store a serial number or code sequence at a fixed address
-SIZE	Report the number of bytes of data contained in the resultant hex image.
-STRING	Store an ASCII string at a fixed address
-STRPACK	Store an ASCII string at a fixed address using string packing
-W	Adjust warning sensitivity
+	Prefix to any option to overwrite other data in its address range if necessary

```
r100-1FFs2000,myfile.hex
```

will shift the block of data from 100h-1FFh to the new address range *2100h-21FFh*.

Be careful when shifting sections of executable code. Program code shouldn't be shifted unless it can be guaranteed that no part of the program relies upon the absolute location of this code segment.

5.15.1.2 + Prefix

When the + operator precedes a parameter or input file, the data obtained from that parameter will be forced into the output file and will overwrite other data existing within its address range. For example:

```
+input.hex +-STRING@1000="My string"
```

Ordinarily, hexmate will issue an error if two sources try to store differing data at the same location. Using the + operator informs hexmate that if more than one data source tries to store data to the same address, the one specified with a '+' will take priority.

5.15.1.3 -ADDRESSING

By default, all address parameters in hexmate options expect that values will be entered as byte addresses. In some device architectures the native addressing format may be something other than byte addressing. In these cases it would be much simpler to be able to enter address-components in the device's native format. To facilitate this, the `-ADDRESSING` option is used. This option takes exactly one parameter which configures the number of bytes contained per address location. If for example a device's program memory naturally used a 16-bit (2 byte) word-addressing format, the option `-ADDRESSING=2` will configure hexmate to interpret all command line address fields as word addresses. The affect of this setting is global and all hexmate options will now interpret addresses according to this setting. This option will allow specification of addressing modes from one byte-per-address to four bytes-per-address.

5.15.1.4 -BREAK

This option takes a comma separated list of addresses. If any of these addresses are encountered in the hex file, the current data record will conclude and a new data record will recommence from the nominated address. This can be useful to use new data records to force a distinction between functionally different areas of program space. Some hex file readers depend on this.

5.15.1.5 -CK

The -CK option is for calculating a checksum. The usage of this option is:

```
-CK=start-end@destination[+offset][wWidth][tCode][gAlogithm]
```

where:

- *Start* and *End* specify the address range that the checksum will be calculated over.
- *Destination* is the address where to store the checksum result. This value cannot be within the range of calculation.
- *Offset* is an optional initial value to add to the checksum result. *Width* is optional and specifies the byte-width of the checksum result. Results can be calculated for byte-widths of 1 to 4 bytes. If a positive width is requested, the result will be stored in big-endian byte order. A negative width will cause the result to be stored in little-endian byte order. If the width is left unspecified, the result will be 2 bytes wide and stored in little-endian byte order.
- *Code* is a hexadecimal code that will trail each byte in the checksum result. This can allow each byte of the checksum result to be embedded within an instruction.
- *Algorithm* is an integer to select which hexmate algorithm to use to calculate the checksum result. A list of selectable algorithms are given in Table 5.10. If unspecified, the default checksum algorithm used is 8 bit addition.

A typical example of the use of the checksum option is:

```
-CK=0-1FFF@2FFE+2100w2
```

This will calculate a checksum over the range `0-1FFFh` and program the checksum result at address `2FFEh`, checksum value will apply an initial offset of `2100h`. The result will be two bytes wide.

5.15.1.6 -FILL

The -FILL option is used for filling unused memory locations with a known value. The usage of this option is:

```
-FILL=Code@Start-End[,data]
```

where:

- *Code* is the opcode that will be programmed to unused locations in memory. Multi-byte codes should be entered in little endian order.

Table 5.10: Hexmate Checksum Algorithm Selection

Selector	Algorithm description
-4	Subtraction of 32 bit values from initial value
-3	Subtraction of 24 bit values from initial value
-2	Subtraction of 16 bit values from initial value
-1	Subtraction of 8 bit values from initial value
1	Addition of 8 bit values from initial value
2	Addition of 16 bit values from initial value
3	Addition of 24 bit values from initial value
4	Addition of 32 bit values from initial value

- *Start* and *End* specify the address range that this fill will apply to.

For example:

```
-FILL=3412@0-1FFF,data
```

will program opcode `1234h` in all unused addresses from program memory address `0` to `1FFFh` (Note the endianism). `-FILL` accepts whole bytes of hexadecimal data from 1 to 8 bytes in length.

Adding the `,data` flag to this option is not required. If the data flag has been specified, hexmate will only perform ROM filling to records that actually contain data. This means that these records will be padded out to the default data record length or the width specified in the -FORMAT option. Records will also begin on addresses which are multiples of the data record length used. The default data record length is 16 bytes. This facility is particularly useful or is a requirement for some bootloaders that expect that all data records will be of a particular length and address alignment.

5.15.1.7 -FIND

This option is used to detect and log occurrences of an opcode or partial code sequence. The usage of this option is:

```
-FIND=Findcode[mMask]@Start-End[/Align][w][t"Title"]
```

where:

- *Findcode* is the hexadecimal code sequence to search for and is entered in little endian byte order.
- *Mask* is optional. It allows a bit mask over the Findcode value and is entered in little endian byte order.

- *Start* and *End* limit the address range to search through.
- *Align* is optional. It specifies that a code sequence can only match if it begins on an address which is a multiple of this value. w, if present will cause hexmate to issue a warning whenever the code sequence is detected.
- *Title* is optional. It allows a title to be given to this code sequence. Defining a title will make log-reports and messages more descriptive and more readable. A title will not affect the actual search results.

TUTORIAL

Let's look at some examples. The option `-FIND=3412@0-7FFF/2w` will detect the code sequence `1234h` when aligned on a `2` (two) byte address boundary, between `0h` and `7FFFh`. `w` indicates that a warning will be issued each time this sequence is found. Another example, `-FIND=3412M0F00@0-7FFF/2wt"ADDXY"` is same as last example but the code sequence being matched is masked with `000Fh`, so hexmate will search for `123xh`. If a byte-mask is used, is must be of equal byte-width to the opcode it is applied to. Any messaging or reports generated by hexmate will refer to this opcode by the name, *ADDXY* as this was the title defined for this search.

If hexmate is generating a log file, it will contain the results of all searches. `-FIND` accepts whole bytes of hex data from 1 to 8 bytes in length. Optionally, `-FIND` can be used in conjunction with `,REPLACE` or ,DELETE (as described below).

5.15.1.8 -FIND...,DELETE

If DELETE is used in conjunction with a -FIND option and a sequence is found that matches the -FIND criteria, it will be removed. This function should be used with extreme caution and is not recommended for removal of executable code.

5.15.1.9 -FIND...,REPLACE

REPLACE Can only be used in conjunction with a -FIND option. Code sequences that matched the -FIND criteria can be replaced or partially replaced with new codes. The usage for this sub-option is:

`-FIND...,REPLACE=`*`Code`*`[m`*`Mask`*`]`

where:

- *Code* is a little endian hexadecimal code to replace the sequences that match the -FIND criteria.
- *Mask* is an optional bit mask to specify which bits within *Code* will replace the code sequence that has been matched. This may be useful if, for example, it is only necessary to modify 4 bits within a 16-bit instruction. The remaining 12 bits can masked and be left unchanged.

5.15.1.10 -FORMAT

The -FORMAT option can be used to specify a particular variant of INHX format or adjust maximum record length. The usage of this option is:

`-FORMAT=`*`Type`*`[,`*`Length`*`]`

where:

- *Type* specifies a particular INHX format to generate.
- *Length* is optional and sets the maximum number of bytes per data record. A valid length is between 1 and 16, with 16 being the default.

TUTORIAL

Consider this case. A bootloader trying to download an INHX32 file fails succeed because it cannot process the extended address records which are part of the INHX32 standard. You know that this bootloader can only program data addressed within the range 0 to 64k, and that any data in the hex file outside of this range can be safely disregarded. In this case, by generating the hex file in INHX8M format the operation might succeed. The hexmate option to do this would be `-FORMAT=INHX8M`.
Now consider this. What if the same bootloader also required every data record to contain eight bytes of data, no more, no less? This is possible by combining `-FORMAT` with `-FILL`. Appropriate use of `-FILL` can ensure that there are no gaps in the data for the address range being programmed. This will satisfy the minimum data length requirement. To set the maximum length of data records to eight bytes, just modify the previous option to become `-FORMAT=INHX8M,8`.

The possible types that are supported by this option are listed in Table 5.11. Note that INHX032 is not an actual INHX format. Selection of this type generates an INHX32 file but will also initialize the upper address information to zero. This is a requirement of some device programmers.

Table 5.11: INHX types used in -FORMAT option

Type	Description
INHX8M	Cannot program addresses beyond 64K.
INHX32	Can program addresses beyond 64K with extended linear address records.
INHX032	INHX32 with initialization of upper address to zero.

5.15.1.11 -HELP

Using -HELP will list all hexmate options. By entering another hexmate option as a parameter of -HELP will show a detailed help message for the given option. For example:

```
-HELP=string
```

will show additional help for the -STRING hexmate option.

5.15.1.12 -LOGFILE

The -LOGFILE option saves hex file statistics to the named file. For example:

```
-LOGFILE=output.log
```

will analyse the hex file that hexmate is generating and save a report to a file named *output.log*.

5.15.1.13 -O*file*

The generated Intel hex output will be created in this file. For example:

```
-Oprogram.hex
```

will save the resultant output to *program.hex*. The output file can take the same name as one of its input files, but by doing so, it will replace the input file entirely.

5.15.1.14 -SERIAL

This option will store a particular hex value at a fixed address. The usage of this option is:

-SERIAL=*Code*[+/-*Increment*]@*Address*[+/-*Interval*][r*Repetitions*]

where:

- *Code* is a hexadecimal value to store and is entered in little endian byte order.

- *Increment* is optional and allows the value of *Code* to change by this value with each repetition (if requested).
- *Address* is the location to store this code, or the first repetition thereof.
- *Interval* is optional and specifies the address shift per repetition of this code.
- *Repetitions* is optional and specifies the number of times to repeat this code.

For example:

```
-SERIAL=000001@EFFE
```

will store hex code `00001h` to address `EFFEh`.

Another example:

```
-SERIAL=0000+2@1000+10r5
```

will store 5 codes, beginning with value `0000` at address `1000h`. Subsequent codes will appear at address intervals of `+10h` and the code value will change in increments of `+2h`.

5.15.1.15 -SIZE

Using the -SIZE option will report the number of bytes of data within the resultant hex image to standard output. The size will also be recorded in the log file if one has been requested.

5.15.1.16 -STRING

The -STRING option will embed an ASCII string at a fixed address. The usage of this option is:

```
-STRING@Address[tCode]="Text"
```

where:

- *Address* is the location to store this string.
- *Code* is optional and allows a byte sequence to trail each byte in the string. This can allow the bytes of the string to be encoded within an instruction.
- *Text* is the string to convert to ASCII and embed.

For example:

```
-STRING@1000="My favourite string"
```

will store the ASCII data for the string, `My favourite string` (including null terminator) at address `1000h`.

Another example:

```
-STRING@1000t34="My favourite string"
```

will store the same string with every byte in the string being trailed with the hex code `34h`.

5.15.1.17 -STRPACK

This option performs the same function as -STRING but with two important differences. Firstly, only the lower seven bits from each character are stored. Pairs of 7 bit characters are then concatenated and stored as a 14 bit word rather than in separate bytes. This is usually only useful for devices where program space is addressed as 14 bit words. The second difference is that -STRING's `t` specifier is not applicable with -STRPACK.

Appendix A

Library Functions

The functions within the standard 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.

__checksum_failed

Synopsis

```
void __checksum_failed(void)
```

Description

This routine will be called during the execution of compiler-generated startup code if it has been requested that a built-in checksum and verification routine be included, and the result of the verification does not match the expected result. The default implementation of this function contains a busy loop which serves no purpose other than to prevent the program from proceeding to `main`. It is implement a customised response to a checksum validation failure according to your system's ability. Some of the possible actions that could be included in a customised routine could be to blink an error LED, re-flash the program image or call for help. If this routine is allowed to return, it will resume execution of the compiler-generated startup code and proceed to `main`. If action was taken to correct the problem, it may be preferred to call for a system reset rather than proceeding through to main as this will cause the checksum to be verified again prior to entering `main`.

Notes

Be aware that this routine will have been entered prior to execution of startup code that would normally clear and initialize data memory. If your customised routine uses variables, their initial states should be assigned by your routine prior to their use.

See also

ram_test_failed()

__CONFIG

Synopsis

```
#include <htc.h>

__CONFIG(n, data)
```

Description

This macro is used to program the configuration fuses that set the device into various modes of operation.

The macro accepts the number corresponding to the configuration register it is to program, then the 16-Bit value it is to update it with.

16-Bit masks have been defined to describe each programmable attribute available on each device. These attribute masks can be found tabulated in this manual in the Features and Runtime Environment section.

Multiple attributes can be selected by ANDing them together.

Example

```
#include <htc.h>

__CONFIG(1,RC & OSCEN)
__CONFIG(2,WDTPS16 & BORV45)
__CONFIG(4, DEBUGEN)

void
main (void)
{
}
```

See also

__EEPROM_DATA(), __IDLOC()

__delay_400_cycles

Synopsis

```
void __delay_ms(int)
```

Description

This routine when called will invoke a fixed delay of 400 instruction cycles multiplied by the value passed to the function as a parameter. Also consider an additional 10 to 12 instruction cycle overhead in the call/return mechanism to this function. Alternately, the macro `__delay_ms()` is defined as an interface to this routine where a delay period can be specified as a number of milliseconds rather than as 400 cycle multiples.

See also

__delay_ms()

Table A.1: Maximum delay versus system frequency

System frequency	Maximum delay
40 MHz	2621 ms
20 MHz	5242 ms
10 MHz	10484 ms

__delay_ms

Synopsis

```
__delay_ms(x)
```

Description

This routine is defined as a preprocessor macro. The macro interfaces with library routine *__delay_400_cycles()*. The parameter in this macro, *x*, defines the number of milliseconds that the program will be delayed by calling this macro. This macro requires the definition of the preprocessor symbol _XTAL_FREQ. This value that this symbol should be defined with is the oscillator frequency (in Hertz) used in your system. The maximum delay period that can be requested varies linearly according to the _XTAL_FREQ setting. Some limits of *x* that *__delay_ms*(x) can use for various frequencies are given in table A.1.

Should these limits be exceeded the actual period of delay will be significantly less than requested.

See also

__delay_400_cycles()

__EEPROM_DATA

Synopsis

```
#include <htc.h>

__EEPROM_DATA(a,b,c,d,e,f,g,h)
```

Description

This macro is used to store initial values into the device's EEPROM registers at the time of programming.

The macro must be given blocks of 8 bytes to write each time it is called, and can be called repeatedly to store multiple blocks.

__EEPROM_DATA() will begin writing to EEPROM address zero, and will auto-increment the address written to by 8, each time it is used.

Example

```
#include <htc.h>

__EEPROM_DATA(0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07)
__EEPROM_DATA(0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F)

void
main (void)
{
}
```

See also

__CONFIG()

__IDLOC

Synopsis

```
#include <htc.h>

__IDLOC(x)
```

Description

This macro places data into the device's special locations outside of addressable memory reserved for ID. This would be useful for storage of serial numbers etc.

The macro will attempt to write 5 nibbles of data to the 5 locations reserved for ID purposes.

Example

```
#include <htc.h>

__IDLOC(15F01);
/* will store 1, 5, F, 0 and 1 in the ID registers*/

void
main (void)
{
}
```

See also

__EEPROM_DATA(), __CONFIG()

__RAM_CELL_TEST

Synopsis

```
void _ram_cell_test(void)
```

Description

Should not be called from user code. This routine is called from a loop within the generated runtime startup code if the system requires a RAM integrity test before program execution. Upon entry to this routine, the FSR0 register has been loaded with the RAM cell to test. The value 0x55 will be assigned to the cell and verified, followed by 0xAA. If either of these values fail verification, the routine will call `ram_test_failed()` with an error code in the working register and FSR0 still containing the address that was in error.

This routine is called repeatedly during startup, with each subsequent call testing the next address in sequence. If the location being tested contains non volatile data, the value will be backed up in the PRODL register before the test routine is called and restored upon return from the routine.

This library routine can be overriden by a user's implementation if the standard cell tests are insufficient for a particular system's verification.

See also

ram_test_failed()

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));
}
```

See Also

labs(), fabs()

Return Value

The absolute value of **j**.

ACOS

Synopsis

```
#include <math.h>

double acos (double f)
```

Description

The **acos()** function implements the inverse 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

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.

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 - π

ASSERT

Synopsis

```
#include <assert.h>

void assert (int e)
```

Description

This macro is used for debugging purposes; the basic method of usage is to place assertions liberally throughout your code at points where correct operation of the code depends upon certain conditions being true initially. An **assert()** routine may be used to ensure at run time that an assumption holds true. For example, the following statement asserts that the pointer tp is not equal to NULL:

assert(tp);

If at run time the expression evaluates to false, the program will abort with a message identifying the source file and line number of the assertion, and the expression used as an argument to it. A fuller discussion of the uses of **assert()** is impossible in limited space, but it is closely linked to methods of proving program correctness.

Example

```
void
ptrfunc (struct xyz * tp)
{
    assert(tp != 0);
}
```

Note

When required for ROM based systems, the underlying routine _fassert(...) will need to be implemented by the user.

ATAN

Synopsis

```
#include <math.h>

double atan (double x)
```

Description

This function returns the arc tangent of its argument, i.e. it returns an angle e in the range - π

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 x, double x)
```

Description

This function returns the arc tangent of `y/x`.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f\n", atan2(10.0, -10.0));
}
```

See Also

sin(), cos(), tan(), asin(), acos(), atan()

Return Value

The arc tangent of `y/x`.

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(), strtod()

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.

BSEARCH

Synopsis

```
#include <stdlib.h>

void * bsearch (const void * key, void * base, size_t n_memb,
size_t size, int (*compar)(const void *, const void *))
```

Description

The **bsearch()** function searches a sorted array for an element matching a particular key. It uses a binary search algorithm, calling the function pointed to by **compar** to compare elements in the array.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

struct value {
    char name[40];
    int value;
} values[100];

int
val_cmp (const void * p1, const void * p2)
{
    return strcmp(((const struct value *)p1)->name,
                  ((const struct value *)p2)->name);
}

void
main (void)
{
    char inbuf[80];
    int i;
    struct value * vp;
```

```
        i = 0;
        while(gets(inbuf)) {
            sscanf(inbuf,"%s %d", values[i].name, &values[i].value);
            i++;
        }
        qsort(values, i, sizeof values[0], val_cmp);
        vp = bsearch("fred", values, i, sizeof values[0], val_cmp);
        if(!vp)
            printf("Item 'fred' was not found\n");
        else
            printf("Item 'fred' has value %d\n", vp->value);
    }
```

See Also

qsort()

Return Value

A pointer to the matched array element (if there is more than one matching element, any of these may be returned). If no match is found, a null pointer is returned.

Note

The comparison function must have the correct prototype.

CEIL

Synopsis

```
#include <math.h>

double ceil (double f)
```

Description

This routine returns the smallest whole number not less than **f**.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    double j;

    scanf("%lf", &j);
    printf("The ceiling of %lf is %lf\n", j, ceil(j));
}
```

CGETS

Synopsis

```
#include <conio.h>

char * cgets (char * s)
```

Description

The **cgets()** function will read one line of input from the console into the buffer passed as an argument. It does so by repeated calls to getche(). As characters are read, they are buffered, with *backspace* deleting the previously typed character, and *ctrl-U* deleting the entire line typed so far. Other characters are placed in the buffer, with a *carriage return* or *line feed (newline)* terminating the function. The collected string is null terminated.

Example

```
#include <conio.h>
#include <string.h>

char buffer[80];

void
main (void)
{
    for(;;) {
        cgets(buffer);
        if(strcmp(buffer, "exit") == 0)
            break;
        cputs("Type 'exit' to finish\n");
    }
}
```

See Also

getch(), getche(), putch(), cputs()

Return Value

The return value is the character pointer passed as the sole argument.

CLRWDT

Synopsis

```
#include <htc.h>

CLRWDT();
```

Description

This macro is used to clear the device's internal watchdog timer.

Example

```
#include <htc.h>

void
main (void)
{
    WDTCON=1;
      /* enable the WDT */

    CLRWDT();
}
```

config_read(), config_write()

Synopsis

```
#include <htc.h>

unsigned int config_read(void);

void config_write(unsigned char, unsigned int);
```

Description

These functions allow access to the device configuration registers which determine many of the behavioural aspects of the device itself.

config_read() accepts a single parameter to determine which config word will be read. The 16-Bit value contained in the register is returned.

config_write() doesn't return any value. It accepts a second parameter which is a 16-Bit value to be written to the selected register.

Example

```
#include <htc.h>

void
main (void)
{
    unsigned int   value;

    value = config_read(2);  // read register 2
    value |= WDTEN;  // modify value
    config_write(2, value);   // update config register

}
```

See Also

device_id_read(),idloc_read(),idloc_write()

Return Value

config_read() returns the 16-Bit value contained in the nominated configuration register.

Note

The functions **config_read() config_write()** are only applicable to such devices that support this feature.

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 implement hyperbolic equivalents of the trigonometric functions; cos(), sin() and tan().

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f\n", cosh(1.5));
    printf("%f\n", sinh(1.5));
    printf("%f\n", tanh(1.5));
}
```

Return Value

The function **cosh()** returns the hyperbolic cosine value.
The function **sinh()** returns the hyperbolic sine value.
The function **tanh()** returns the hyperbolic tangent value.

CPUTS

Synopsis

```
#include <conio.h>

void cputs (const char * s)
```

Description

The **cputs()** function writes its argument string to the console, outputting *carriage returns* before each *newline* in the string. It calls putch() repeatedly. On a hosted system **cputs()** differs from puts() in that it writes to the console directly, rather than using file I/O. In an embedded system **cputs()** and puts() are equivalent.

Example

```
#include <conio.h>
#include <string.h>

char buffer[80];

void
main (void)
{
    for(;;) {
        cgets(buffer);
        if(strcmp(buffer, "exit") == 0)
            break;
        cputs("Type 'exit' to finish\n");
    }
}
```

See Also

cputs(), puts(), putch()

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.

device_id_read()

Synopsis

```
#include <htc.h>

unsigned int device_id_read(void);
```

Description

This function returns the device ID code that is factory-programmed into the chip. This code can be used to identify the device and its revision number.

Example

```
#include <htc.h>

void
main (void)
{
    unsigned int   id_value;
    unsigned int   device_code;
    unsigned char revision_no;

    id_value = device_id_read();
        /* lower 5 bits represent revision number
         * upper 11 bits identify device */
    device_code = (id_value >> 5);
    revision_no = (unsigned char)(id_value & 0x1F);

}
```

See Also

flash_read(), config_read()

Return Value

device_id_read() returns the 16-Bit factory-programmed device id code used to identify the device type and its revision number.

Note

The **device_id_read()** is applicable only to those devices which are capable of reading their own program memory.

DI, EI

Synopsis

```
#include <htc.h>

void ei (void)
void di (void)
```

Description

The **di()** macro disables all interrupts globally (regardless of priority settings), **ei()** re-enables interrupts globally. These are implemented as macros defined in **PIC18.h**. 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 <htc.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;
```

```
}
```

Note

As these macros act on the global interrupt enable bit of the PIC18 processor, ei() will only restore those interrupt sources that were previously enabled.

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.

EEPROM_READ, EEPROM_WRITE

Synopsis

```
#include <htc.h>

unsigned char eeprom_read (unsigned int address);
void eeprom_write (unsigned int address, unsigned char value);
```

Description

These functions allow access to the on-chip 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 this memory. 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 datum.

Reading data is completed in the one cycle and no polling is necessary to check for a read completion.

Example

```
#include <htc.h>

void
main (void)
{
    unsigned char data;
    unsigned int address = 0x0010;

    data=eeprom_read(address);
    eeprom_write(address, data);
}
```

See Also

flash_erase, flash_read, flash_write

Note

The high and low priority interrupt are disabled during sensitive sequences required to access EEPROM. Interrupts are restored after the sequence has completed. eeprom_write() will clear the EEIF hardware flag before returning.

Both eeprom_read() and eeprom_write() are available in a similar macro form. The essential difference between the macro and function implementations is that `EEPROM_READ()`, the macro, does not test nor wait for any prior write operations to complete.

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(), labs()

FLASH_ERASE, FLASH_READ, FLASH_WRITE

Synopsis

```
#include <htc.h>

void flash_erase (unsigned long addr);
unsigned char flash_read (unsigned long addr);
void flash_write(const unsigned char * source, unsigned int length,
     far unsigned char * dest_addr);
```

Description

These functions allow access to the flash memory of the microcontroller (when present).

Reading from the flash memory can be done one byte at a time with use of the **flash_read()** function. **flash_read** returns the data value found at the specified address in flash memory.

Entire sectors of flash memory can be restored to an unprogrammed state (value=**FFFFh**) with use of the **flash_erase()** function. The number of bytes erased per flash erase action is specific to each device's flash-erase block size. Specifying an address to the **flash_erase** function, will erase the entire block that contains the given address.

flash_write() copies blocks of data/code from RAM/flash to a new destination in flash memory. **flash_write** requires a pointer to the data that will be copied, the length of data to copy (in bytes) and a pointer to the destination address in flash memory. This function can be used to update values of variables declared as const. Any erasures of flash memory required in order write to the flash are performed within **flash_write** and pre-erasure is not required.

Example

```
#include <htc.h>
const unsigned char old_text[]="insert text here";
unsigned char new_text[]="HI-TECH Software";
void
main (void)
{
    const unsigned char * source = new_text;
    far unsigned char * dest = (far unsigned char *) old_text;
    unsigned char data;
    unsigned int length = sizeof(new_text);
```

```
    // Read a byte of data from flash address 1000h
    data = flash_read(0x1000);

    // Copy data from source to destination in flash.
    // source does not have be in flash.
    // Any required flash erasures done internally.
    flash_write(source, length, dest);

    // Erase block containing the address 4000h
    flash_erase(0x4000);
}
```

Return Value

flash_read() returns the data found at the given address, as an unsigned char.

Note

The **flash_write()** function can be used to update anywhere from 1 to 65535 bytes of data at a time, however it is more efficient to write in data lengths that are multiples of that device's erase block size.
Ensure that the function does not attempt to overwrite the section of program memory from which it is currently executing and extreme caution must be exercised if modifying code at the device's reset or interrupt vectors. A reset or interrupt must not be triggered while this sector is in erasure. For PIC18FxxJxx parts, care must also be taken to ensure that a reset does not occur during a write targetting the last 1024 bytes of program space as this region also contains the device's configuration values.

FMOD

Synopsis

```
#include <math.h>

double fmod (double x, double y)
```

Description

The function **fmod** returns the remainder of **x/y** as a floating point quantity.

Example

```
#include <math.h>

void
main (void)
{
    double rem, x;

    x = 12.34;
    rem = fmod(x, 2.1);
}
```

Return Value

The floating-point remainder of x/y.

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()

FTOA

Synopsis

```
#include <stdlib.h>

char * ftoa (float f, int * status)
```

Description

The function **ftoa** converts the contents of **f** into a string which is stored into a buffer which is then return.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char * buf;
    float input = 12.34;
    int status;
    buf = ftoa(input, &status);
    printf("The buffer holds %s\n", buf);
}
```

See Also

strtol(), itoa(), utoa(), ultoa()

Return Value

This routine returns a reference to the buffer into which the result is written.

GETCH, GETCHE

Synopsis

```
#include <conio.h>

char getch (void)
char getche (void)
```

Description

The **getch()** function reads a single character from the console keyboard and returns it without echoing. The **getche()** function is similar but does echo the character typed.

In an embedded system, the source of characters is defined by the particular routines supplied. By default, the library contains a version of **getch()** that will interface to the Lucifer Debugger. The user should supply an appropriate routine if another source is desired, e.g. a serial port.

The module *getch.c* in the SOURCES directory contains model versions of all the console I/O routines. Other modules may also be supplied, e.g. *ser180.c* has routines for the serial port in a Z180.

Example

```
#include <conio.h>

void
main (void)
{
    char c;

    while((c = getche()) != '\n')
        continue;
}
```

See Also

cgets(), cputs(), ungetch()

GETCHAR

Synopsis

```
#include <stdio.h>

int getchar (void)
```

Description

The **getchar()** routine is a getc(stdin) operation. It is a macro defined in **stdio.h**. Note that under normal circumstances **getchar()** will NOT return unless a *carriage return* has been typed on the console. To get a single character immediately from the console, use the function getch().

Example

```
#include <stdio.h>

void
main (void)
{
    int c;

    while((c = getchar()) != EOF)
        putchar(c);
}
```

See Also

getc(), fgetc(), freopen(), fclose()

Note

This routine is not usable in a ROM based system.

GETS

Synopsis

```
#include <stdio.h>

char * gets (char * s)
```

Description

The **gets()** function reads a line from standard input into the buffer at **s**, deleting the *newline* (cf. fgets()). The buffer is null terminated. In an embedded system, **gets()** is equivalent to cgets(), and results in getche() being called repeatedly to get characters. Editing (with *backspace*) is available.

Example

```
#include <stdio.h>

void
main (void)
{
    char buf[80];

    printf("Type a line: ");
    if(gets(buf))
        puts(buf);
}
```

See Also

fgets(), freopen(), puts()

Return Value

It returns its argument, or NULL on end-of-file.

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.

idloc_read(),idloc_write()

Synopsis

```
#include <htc.h>

unsigned char idloc_read(void);

void idloc_write(unsigned char, unsigned char);
```

Description

These functions allow access to the user ID register which can be used to store small amounts of information such as serial numbers, checksums etc.

idloc_read() accepts a single parameter to determine which user ID register to read. The value contained in the register is returned.

idloc_write() doesn't return any value. It accepts a second parameter which is a value to be written to the selected register. Note that only the lower nibble is significant. The upper nibble of the value written will always be 0xF as per Microchip's documentation.

Example

```
#include <htc.h>

void
main (void)
{
    unsigned char   value;

    value = idloc_read(2);  // read register 2
    value++;                // modify value
    idloc_write(2, value);  // update user ID register

}
```

See Also

device_id_read(),config_read(),config_write()

Return Value

idloc_read() returns the value contained in the nominated user ID register.

Note

The functions **idloc_read() idloc_write()** are only applicable to such devices that support this feature.

Note also that ICD2 breakpoints should not be set within the **idloc_write()** function. Doing so can result in disrupting the operation of the debugger.

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()

ISDIG

Synopsis

```
#include <ctype.h>

int isdig (int c)
```

Description

The **isdig**() function tests the input character `c` to see if is a decimal digit (0 – 9) and returns true is this is the case; false otherwise.

Example

```
#include <ctype.h>

void
main (void)
{
    char buf[] = "1998a";
    if(isdig(buf[0]))
        printf("valid type detected\n");
}
```

See Also

isdigit() (listed un isalnum())

Return Value

Zero if the character is a decimal digit; a non-zero value otherwise.

ITOA

Synopsis

```
#include <stdlib.h>

char * itoa (char * buf, int val, int base)
```

Description

The function **itoa** converts the contents of **val** into a string which is stored into **buf**. The conversion is performed according to the radix specified in **base**. **buf** is assumed to reference a buffer which has sufficent space allocated to it.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[10];
    itoa(buf, 1234, 16);
    printf("The buffer holds %s\n", buf);
}
```

See Also

strtol(), utoa(), ltoa(), ultoa()

Return Value

This routine returns a copy of the buffer into which the result is written.

LABS

Synopsis

```
#include <stdlib.h>

int labs (long int j)
```

Description

The **labs()** function returns the absolute value of long value **j**.

Example

```
#include <stdio.h>
#include <stdlib.h>

void
main (void)
{
    long int a = -5;

    printf("The absolute value of %ld is %ld\n", a, labs(a));
}
```

See Also

abs()

Return Value

The absolute value of **j**.

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**

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. On systems where it is not possible to predetermine this value, **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.

LOG, LOG10

Synopsis

```
#include <math.h>

double log (double f)
double log10 (double f)
```

Description

The **log()** function returns the natural logarithm of **f**. The function **log10()** returns the logarithm to base 10 of **f**.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    for(f = 1.0 ; f <= 10.0 ; f += 1.0)
        printf("log(%1.0f) = %f\n", f, log(f));
}
```

See Also

exp(), pow()

Return Value

Zero if the argument is negative.

LONGJMP

Synopsis

```
#include <setjmp.h>

void longjmp (jmp_buf buf, int val)
```

Description

The **longjmp()** function, in conjunction with setjmp(), provides a mechanism for non-local goto's. To use this facility, setjmp() should be called with a **jmp_buf** argument in some outer level function. The call from setjmp() will return 0.

To return to this level of execution, **longjmp()** may be called with the same **jmp_buf** argument from an inner level of execution. ***Note*** however that the function which called setjmp() must still be active when **longjmp()** is called. Breach of this rule will cause disaster, due to the use of a stack containing invalid data. The **val** argument to **longjmp()** will be the value apparently returned from the setjmp(). This should normally be non-zero, to distinguish it from the genuine setjmp() call.

Example

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf jb;

void
inner (void)
{
    longjmp(jb, 5);
}

void
main (void)
{
    int i;
```

```
        if(i = setjmp(jb)) {
            printf("setjmp returned %d\n", i);
            exit(0);
        }
        printf("setjmp returned 0 - good\n");
        printf("calling inner...\n");
        inner();
        printf("inner returned - bad!\n");
    }
```

See Also

setjmp()

Return Value

The **longjmp()** routine never returns.

Note

The function which called setjmp() must still be active when **longjmp()** is called. Breach of this rule will cause disaster, due to the use of a stack containing invalid data.

LTOA

Synopsis

```
#include <stdlib.h>

char * ltoa (char * buf, long val, int base)
```

Description

The function **itoa** converts the contents of **val** into a string which is stored into **buf**. The conversion is performed according to the radix specified in **base**. **buf** is assumed to reference a buffer which has sufficent space allocated to it.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[10];
    utoi(buf, 12345678L, 16);
    printf("The buffer holds %s\n", buf);
}
```

See Also

strtol(), itoa(), utoa(), ultoa()

Return Value

This routine returns a copy of the buffer into which the result is written.

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.

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.

MEMMOVE

Synopsis

```
#include <string.h>

void * memmove (void * s1, const void * s2, size_t n)
```

Description

The **memmove()** function is similar to the function memcpy() except copying of overlapping blocks is handled correctly. That is, it will copy forwards or backwards as appropriate to correctly copy one block to another that overlaps it.

See Also

strncpy(), strncmp(), strchr(), memcpy()

Return Value

The function **memmove()** returns its first argument.

MKTIME

Synopsis

```
#include <time.h>

time_t mktime (struct tm * tmptr)
```

Description

The **mktime()** function converts the local calendar time referenced by the tm structure pointer **tmptr** into a time being the number of seconds passed since Jan 1st 1970, or -1 if the time cannot be represented.

Example

```
#include <time.h>
#include <stdio.h>

void
main (void)
{
    struct tm birthday;

    birthday.tm_year = 1955;
    birthday.tm_mon = 2;
    birthday.tm_mday = 24;
    birthday.tm_hour = birthday.tm_min = birthday.tm_sec = 0;
    printf("you have been alive approximately %ld seconds\n", mktime(&birthday));
}
```

See Also

ctime(), asctime()

Return Value

The time contained in the **tm** structure represented as the number of seconds since the 1970 Epoch, or -1 if this time cannot be represented.

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 integral 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**.

NOP

Synopsis

```
#include <htc.h>

NOP();
```

Description

Execute NOP instruction here. This is often useful to finetune delays or create a handle for breakpoints. The NOP instruction is sometimes required during some sensitive sequences in hardware.

Example

```
#include <htc.h>

void
crude_delay(unsigned char x) {
   while(x--){
        NOP();  /* Do nothing for 3 cycles */
        NOP();
        NOP();
   }
}
```

OS_TSLEEP

Synopsis

```
#include <task.h>

void os_tsleep(unsigned short tcks)
```

Description

This routine causes the current task to be removed from the run queue for **tcks** clock ticks.

Example

```
#include <task.h>

void
task(void)
{
    while(1) {
        /* sleep for 100 ticks */
        os_tsleep(100);
    }
}
```

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 NVRAM 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.

If the character ***** is used in place of a decimal constant, e.g. in the format **%*d**, then one integer argument will be taken from the list to provide that value. The types of conversion are:

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.

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.

printf("Name = %.8s", "a1234567890")
       yields 'Name = a1234567'

printf("xx%*d", 3, 4)
       yields 'xx  4'

/* vprintf example */

#include <stdio.h>

int
error (char * s, ...)
{
    va_list ap;

    va_start(ap, s);
    printf("Error: ");
    vprintf(s, ap);
    putchar('\n');
    va_end(ap);
}

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 when linking in alternative libraries. 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 PICC18 -L library scan option for more details.

PRINTF, VPRINTF

Synopsis

```
#include <stdio.h>

int printf (const char * fmt, ...)

#include <stdio.h>
#include <stdarg.h>

int vprintf (const char * fmt, va_list va_arg)
```

Description

The **printf()** function is a formatted output routine, operating on stdout. There are corresponding routines operating on a given stream (fprintf()) or into a string buffer (sprintf()). The **printf()** routine is passed a format string, followed by a list of zero or more arguments. In the format string are conversion specifications, each of which is used to print out one of the argument list values.

Each conversion specification is of the form **%m.nc** where the percent symbol **%** introduces a conversion, followed by an optional width specification **m**. The **n** specification is an optional precision specification (introduced by the dot) and **c** is a letter specifying the type of the conversion.

A minus sign ('-') preceding **m** indicates left rather than right adjustment of the converted value in the field. Where the field width is larger than required for the conversion, blank padding is performed at the left or right as specified. Where right adjustment of a numeric conversion is specified, and the first digit of **m** is 0, then padding will be performed with zeroes rather than blanks. For integer formats, the precision indicates a minimum number of digits to be output, with leading zeros inserted to make up this number if required.

A hash character (**#**) preceding the width indicates that an alternate format is to be used. The nature of the alternate format is discussed below. Not all formats have alternates. In those cases, the presence of the hash character has no effect.

The floating point formats require that the appropriate floating point library is linked. From within HPD this can be forced by selecting the "Float formats in printf" selection in the options menu. From the command line driver, use the option **-LF**.

If the character ***** is used in place of a decimal constant, e.g. in the format **%*d**, then one integer argument will be taken from the list to provide that value. The types of conversion are:

f

Floating point - **m** is the total width and **n** is the number of digits after the decimal point. If **n** is

omitted it defaults to 6. If the precision is zero, the decimal point will be omitted unless the alternate format is specified.

e

Print the corresponding argument in scientific notation. Otherwise similar to **f**.

g

Use **e** or **f** format, whichever gives maximum precision in minimum width. Any trailing zeros after the decimal point will be removed, and if no digits remain after the decimal point, it will also be removed.

o x X u d

Integer conversion - in radices 8, 16, 16, 10 and 10 respectively. The conversion is signed in the case of **d**, unsigned otherwise. The precision value is the total number of digits to print, and may be used to force leading zeroes. E.g. **%8.4x** will print at least 4 hex digits in an 8 wide field. Preceding the key letter with an **l** indicates that the value argument is a long integer. The letter **X** prints out hexadecimal numbers using the upper case letters *A-F* rather than *a*-f as would be printed when using **x**. When the alternate format is specified, a leading zero will be supplied for the octal format, and a leading 0x or 0X for the hex format.

s

Print a string - the value argument is assumed to be a character pointer. At most **n** characters from the string will be printed, in a field **m** characters wide.

c

The argument is assumed to be a single character and is printed literally.

Any other characters used as conversion specifications will be printed. Thus **%** will produce a single percent sign.

The **vprintf()** function is similar to **printf()** but takes a variable argument list pointer rather than a list of arguments. See the description of va_start() for more information on variable argument lists. An example of using **vprintf()** is given below.

Example

```
printf("Total = %4d%", 23)
       yields 'Total =   23%'

printf("Size is %lx" , size)
       where size is a long, prints size
       as hexadecimal.

printf("Name = %.8s", "a1234567890")
       yields 'Name = a1234567'
```

```
printf("xx%*d", 3, 4)
       yields 'xx  4'


/* vprintf example */

#include        <stdio.h>

int
error (char * s, ...)
{
    va_list ap;

    va_start(ap, s);
    printf("Error: ");
    vprintf(s, ap);
    putchar('\n');
    va_end(ap);
}

void
main (void)
{
    int i;

    i = 3;
    error("testing 1 2 %d", i);
}
```

See Also

fprintf(), sprintf()

Return Value

The **printf()** and **vprintf()** functions return the number of characters written to stdout.

PUTCH

Synopsis

```
#include <conio.h>

void putch (char c)
```

Description

The **putch()** function outputs the character **c** to the console screen, prepending a *carriage return* if the character is a *newline*. In a CP/M or MS-DOS system this will use one of the system I/O calls. In an embedded system this routine, and associated others, will be defined in a hardware dependent way. The standard **putch()** routines in the embedded library interface either to a serial port or to the Lucifer Debugger.

Example

```
#include <conio.h>

char * x = "This is a string";

void
main (void)
{
    char * cp;

    cp = x;
    while(*x)
        putch(*x++);
    putch('\n');
}
```

See Also

cgets(), cputs(), getch(), getche()

PUTCHAR

Synopsis

```
#include <stdio.h>

int putchar (int c)
```

Description

The **putchar()** function is a putc() operation on stdout, defined in **stdio.h**.

Example

```
#include <stdio.h>

char * x = "This is a string";

void
main (void)
{
    char * cp;

    cp = x;
    while(*x)
        putchar(*x++);
    putchar('\n');
}
```

See Also

putc(), getc(), freopen(), fclose()

Return Value

The character passed as argument, or EOF if an error occurred.

Note

This routine is not usable in a ROM based system.

PUTS

Synopsis

```
#include <stdio.h>

int puts (const char * s)
```

Description

The **puts()** function writes the string **s** to the *stdout stream*, appending a *newline*. The null character terminating the string is not copied.

Example

```
#include <stdio.h>

void
main (void)
{
    puts("Hello, world!");
}
```

See Also

fputs(), gets(), freopen(), fclose()

Return Value

EOF is returned on error; zero otherwise.

QSORT

Synopsis

```
#include <stdlib.h>

void qsort (void * base, size_t nel, size_t width,
int (*func)(const void *, const void *))
```

Description

The **qsort()** function is an implementation of the quicksort algorithm. It sorts an array of **nel** items, each of length **width** bytes, located contiguously in memory at **base**. The argument **func** is a pointer to a function used by **qsort()** to compare items. It calls **func** with pointers to two items to be compared. If the first item is considered to be greater than, equal to or less than the second then **func** should return a value greater than zero, equal to zero or less than zero respectively.

Example

```
#include <stdio.h>
#include <stdlib.h>

int aray[] = {
    567, 23, 456, 1024, 17, 567, 66
};

int
sortem (const void * p1, const void * p2)
{
    return *(int *)p1 - *(int *)p2;
}

void
main (void)
{
    register int i;
```

```
    qsort(aray, sizeof aray/sizeof aray[0], sizeof aray[0], sortem);
    for(i = 0 ; i != sizeof aray/sizeof aray[0] ; i++)
        printf("%d\t", aray[i]);
    putchar('\n');
}
```

Note

The function parameter must be a pointer to a function of type similar to:

```
int func (const void *, const void *)
```

i.e. it must accept two const void * parameters, and must be prototyped.

RAM_TEST_FAILED

Synopsis

```
void ram_test_failed (unsigned char errcode)
```

Description

The **ram_test_failed()** function is not intended to be called from within the general execution of the program. This routine is called during execution of the generated runtime startup code if the program is using a compiler generated RAM integrity test and the integrity test detects a bad cell.

This function is passed a single byte error code as a parameter. The address that failed this test will be loaded in the FSR0 registers. The failed value will still be accessable through the INDF0 register. The default operation of this routine will halt program execution if a bad cell is detected, however the user is free to enhance this functionality if required.

See Also

__ram_cell_test

Note

This routine is intended to be replaced by an equivalent routine to suit the user's implementation. Possible enhancements include logging the location of the dead cell and continuing to test if there are any more more dead cells, or alerting the outside world that the device has a memory problem.

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.

READTIMER*x*

Synopsis

```
#include <htc.h>

READTIMER0();
READTIMER1();
READTIMER2();
```

Description

The macros **READTIMER0()**, **READTIMER1()** and **READTIMER2()** will return the 16-Bit value presently held in the device's corresponding TMR*x*L and TMR*x*H register pair. Use of this macro ensures that the registers are read in the correct order.

Example

```
#include <htc.h>

void
main (void)
{
    unsigned int timer1value;

    timer1value = READTIMER1();
}
```

See Also

WRITETIMER*x*()

Return Value

An unsigned integer which is the value held in a 16-Bit timer.

RESET

Synopsis

```
#include <htc.h>

RESET();
```

Description

Execute RESET instruction here.

Example

```
#include <htc.h>

void
test_result(unsigned int error_count) {
   if(error_count != 0){
        printf("An error has been detected - Rebooting...\n");
        RESET();  /* Perform software reset */
   }
}
```

ROUND

Synopsis

```
#include <math.h>

double round (double x)
```

Description

The **round** function round the argument to the nearest integer value, but in floating-point format. Values midway between integer values are rounded up.

Example

```
#include <math.h>

void
main (void)
{
    double input, rounded;
    input = 1234.5678;
    rounded = round(input);
}
```

See Also

trunc()

SCANF, VSCANF

Synopsis

```
#include <stdio.h>

int scanf (const char * fmt, ...)

#include <stdio.h>
#include <stdarg.h>

int vscanf (const char *, va_list ap)
```

Description

The **scanf()** function performs formatted input ("de-editing") from the *stdin stream*. Similar functions are available for streams in general, and for strings. The function **vscanf()** is similar, but takes a pointer to an argument list rather than a series of additional arguments. This pointer should have been initialised with va_start().

The input conversions are performed according to the **fmt** string; in general a character in the format string must match a character in the input; however a space character in the format string will match zero or more "white space" characters in the input, i.e. *spaces, tabs or newlines*.

A conversion specification takes the form of the character **%**, optionally followed by an assignment suppression character ('*'), optionally followed by a numerical maximum field width, followed by a conversion specification character. Each conversion specification, unless it incorporates the assignment suppression character, will assign a value to the variable pointed at by the next argument. Thus if there are two conversion specifications in the **fmt** string, there should be two additional pointer arguments.

The conversion characters are as follows:

o x d

Skip white space, then convert a number in base 8, 16 or 10 radix respectively. If a field width was supplied, take at most that many characters from the input. A leading minus sign will be recognized.

f

Skip white space, then convert a floating number in either conventional or scientific notation. The field width applies as above.

s

Skip white space, then copy a maximal length sequence of non-white-space characters. The pointer

argument must be a pointer to char. The field width will limit the number of characters copied. The resultant string will be null terminated.

c

Copy the next character from the input. The pointer argument is assumed to be a pointer to char. If a field width is specified, then copy that many characters. This differs from the **s** format in that white space does not terminate the character sequence.

The conversion characters **o**, **x**, **u**, **d** and **f** may be preceded by an **l** to indicate that the corresponding pointer argument is a pointer to long or double as appropriate. A preceding **h** will indicate that the pointer argument is a pointer to short rather than int.

Example

```
scanf("%d %s", &a, &c)
     with input " 12s"
     will assign 12 to a, and "s" to s.

scanf("%3cd %lf", &c, &f)
     with input " abcd -3.5"
     will assign " abc" to c, and -3.5 to f.
```

See Also

fscanf(), sscanf(), printf(), va_arg()

Return Value

The **scanf()** function returns the number of successful conversions; EOF is returned if end-of-file was seen before any conversions were performed.

SETJMP

Synopsis

```
#include <setjmp.h>

int setjmp (jmp_buf buf)
```

Description

The **setjmp()** function is used with longjmp() for non-local goto's. See longjmp() for further information.

Example

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf jb;

void
inner (void)
{
    longjmp(jb, 5);
}

void
main (void)
{
    int i;

    if(i = setjmp(jb)) {
        printf("setjmp returned %d\n", i);
        exit(0);
    }
    printf("setjmp returned 0 - good\n");
    printf("calling inner...\n");
```

```
        inner();
        printf("inner returned - bad!\n");
    }
```

See Also

longjmp()

Return Value

The **setjmp()** function returns zero after the real call, and non-zero if it apparently returns after a call to longjmp().

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**.

SLEEP

Synopsis

```
#include <htc.h>

SLEEP();
```

Description

This macro is used to put the device into a low-power standby mode.

Example

```
#include <htc.h>
extern void init(void);

void
main (void)
{
    init(); /* enable peripherals/interrupts */

    while(1)
        SLEEP();  /* save power while nothing happening */
}
```

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>

char * strcat (char * s1, const char * s2)
```

Description

This function appends (concatenates) 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>

const char * strchr (const char * s, int c)
const 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...";
    const char c = 's';

    if(strchr(temp, c))
        printf("Character %c was found in string\n", c);
    else
        printf("No character was found in string");
}
```

See Also

strrchr(), strlen(), strcmp()

Return Value

A pointer to the first match found, or NULL if the character does not exist in the string.

Note

Although the function takes an integer argument for the character, only the lower 8 bits of the value are used.

STRCMP, STRICMP

Synopsis

```
#include <string.h>

int strcmp (const char * s1, const char * s2)
int stricmp (const char * s1, const char * s2)
```

Description

The **strcmp()** function compares its two, null terminated, string arguments and returns a signed integer to indicate whether **s1** is less than, equal to or greater than **s2**. The comparison is done with the standard collating sequence, which is that of the ASCII character set.

The **stricmp()** function is the case-insensitive version of this function.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    int i;

    if((i = strcmp("ABC", "ABc")) < 0)
        printf("ABC is less than ABc\n");
    else if(i > 0)
        printf("ABC is greater than ABc\n");
    else
        printf("ABC is equal to ABc\n");
}
```

See Also

strlen(), strncmp(), strcpy(), strcat()

Return Value

A signed integer less than, equal to or greater than zero.

Note

Other C implementations may use a different collating sequence; the return value is negative, zero or positive, i.e. do not test explicitly for negative one (-1) or one (1).

STRCPY

Synopsis

```
#include <string.h>

char * strcpy (char * s1, const char * s2)
```

Description

This function copies a null terminated string **s2** to a character array pointed to by **s1**. The destination array must be large enough to hold the entire string, including the null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = " ... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

See Also

strncpy(), strlen(), strcat(), strlen()

Return Value

The destination buffer pointer **s1** is returned.

STRCSPN

Synopsis

```
#include <string.h>

size_t strcspn (const char * s1, const char * s2)
```

Description

The **strcspn()** function returns the length of the initial segment of the string pointed to by **s1** which consists of characters NOT from the string pointed to by **s2**.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    static char set[] = "xyz";

    printf("%d\n", strcspn( "abcdevwxyz", set));
    printf("%d\n", strcspn( "xxxbcadefs", set));
    printf("%d\n", strcspn( "1234567890", set));
}
```

See Also

strspn()

Return Value

Returns the length of the segment.

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>

char * strncat (char * s1, const char * s2, size_t n)
```

Description

This function appends (concatenates) 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 **strncmp()** 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 **strnicmp()** function is the case-insensitive version of this function.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    int i;

    i = strncmp("abcxyz", "abcxyz",6);
    if(i == 0)
        printf("Both strings are equal\n");
    else if(i > 0)
        printf("String 2 less than string 1\n");
    else
        printf("String 2 is greater than string 1\n");
}
```

See Also

strlen(), strcmp(), strcpy(), strcat()

Return Value

A signed integer less than, equal to or greater than zero.

Note

Other C implementations may use a different collating sequence; the return value is negative, zero or positive, i.e. do not test explicitly for negative one (-1) or one (1).

STRNCPY

Synopsis

```
#include <string.h>

char * strncpy (char * s1, const char * s2, size_t n)
```

Description

This function copies a null terminated string **s2** to a character array pointed to by **s1**. At most **n** characters are copied. If string **s2** is longer than **n** then the destination string will not be null terminated. The destination array must be large enough to hold the entire string, including the null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strncpy(buffer, "Start of line", 6);
    s1 = buffer;
    s2 = " ... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

See Also

strcpy(), strcat(), strlen(), strcmp()

Return Value

The destination buffer pointer **s1** is returned.

STRPBRK

Synopsis

```
#include <string.h>

char * strpbrk (const char * s1, const char * s2)
```

Description

The **strpbrk()** function returns a pointer to the first occurrence in string **s1** of any character from string **s2**, or a null pointer if no character from **s2** exists in **s1**.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * str = "This is a string.";

    while(str != NULL) {
        printf( "%s\n", str );
        str = strpbrk( str+1, "aeiou" );
    }
}
```

Return Value

Pointer to the first matching character, or NULL if no character found.

STRRCHR, STRRICHR

Synopsis

```
#include <string.h>

char * strrchr (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>

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.

STRTOD

Synopsis

```
#include <stdlib.h>

double strtok (const char * s, const char ** res)
```

Description

Parse the string `s` converting it to a double floating point type. This function converts the first occurence of a substring of the input that is made up of characters of the expected form after skipping leading white-space characters. If `res` is not `NULL`, it will be made to point to the first character after the converted sub-string.

Example

```
#include <stdio.h>
#include <strlib.h>

void
main (void)
{
    char buf[] = " 35.7  23.27 ";
    char * end;
    double in1, in2;

    in1 = strtod(buf, &end);
    in2 = strtod(end, NULL);
    printf("in comps: %f, %f\n", in1, in2);
}
```

See Also

atof()

Return Value

Returns a double representing the floating-point value of the converted input string.

STRTOL

Synopsis

```
#include <stdlib.h>

double strtol (const char * s, const char ** res, int base)
```

Description

Parse the string `s` converting it to a long integer type. This function converts the first occurence of a substring of the input that is made up of characters of the expected form after skipping leading white-space characters. The radix of the input is determined from **base**. If this is zero, then the radix defaults to base 10. If **res** is not `NULL`, it will be made to point to the first character after the converted sub-string.

Example

```
#include <stdio.h>
#include <strlib.h>

void
main (void)
{
    char buf[] = " 0X299 0x792 ";
    char * end;
    long in1, in2;

    in1 = strtol(buf, &end, 16);
    in2 = strtol(end, NULL, 16);
    printf("in (decimal): %ld, %ld\n", in1, in2);
}
```

See Also

strtod()

Return Value

Returns a long int representing the value of the converted input string using the specified base.

STRTOK

Synopsis

```
#include <string.h>

char * strtok (char * s1, const char * s2)
```

Description

A number of calls to **strtok()** breaks the string **s1** (which consists of a sequence of zero or more text tokens separated by one or more characters from the separator string **s2**) into its separate tokens.

The first call must have the string **s1**. This call returns a pointer to the first character of the first token, or NULL if no tokens were found. The inter-token separator character is overwritten by a null character, which terminates the current token.

For subsequent calls to **strtok()**, **s1** should be set to a null pointer. These calls start searching from the end of the last token found, and again return a pointer to the first character of the next token, or NULL if no further tokens were found.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * ptr;
    char buf[] = "This is a string of words.";
    char * sep_tok = ".,?! ";

    ptr = strtok(buf, sep_tok);
    while(ptr != NULL) {
        printf("%s\n", ptr);
        ptr = strtok(NULL, sep_tok);
    }
}
```

Return Value

Returns a pointer to the first character of a token, or a null pointer if no token was found.

Note

The separator string **s2** may be different from call to call.

TAN

Synopsis

```
#include <math.h>

double tan (double f)
```

Description

The **tan()** function calculates the tangent of **f**.

Example

```
#include <math.h>
#include <stdio.h>

#define C 3.141592/180.0

void
main (void)
{
    double i;

    for(i = 0 ; i <= 180.0 ; i += 10)
        printf("tan(%3.0f) = %f\n", i, tan(i*C));
}
```

See Also

sin(), cos(), asin(), acos(), atan(), atan2()

Return Value

The tangent of **f**.

TIME

Synopsis

```
#include <time.h>

time_t time (time_t * t)
```

Description

This function is not provided as it is dependant on the target system supplying the current time. This function will be user implemented. When implemented, this function should return the current time in seconds since 00:00:00 on Jan 1, 1970. If the argument **t** is not equal to NULL, the same value is stored into the object pointed to by **t**.

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;

    time(&clock);
    printf("%s", ctime(&clock));
}
```

See Also

ctime(), gmtime(), localtime(), asctime()

Return Value

This routine when implemented will return the current time in seconds since 00:00:00 on Jan 1, 1970.

Note

The **time()** routine is not supplied, if required the user will have to implement this routine to the specifications outlined above.

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.

TRUNC

Synopsis

```
#include <math.h>

double trunc (double x)
```

Description

The **trunc** function rounds the argument to the nearest integer value, in floating-point format, that is not larger in magniture than the argument.

Example

```
#include <math.h>

void
main (void)
{
    double input, rounded;
    input = 1234.5678;
    rounded = trunc(input);
}
```

See Also

round()

ULDIV

Synopsis

```
#include <stdlib.h>

int uldiv (unsigned long num, unsigned long demon)
```

Description

The **uldiv()** function calculate the quotient and remainder of the division of `number` and `denom`, storing the results into a uldiv_t structure which is returned.

Example

```
#include <stdlib.h>

void
main (void)
{
    uldiv_t result;
    unsigned long num = 1234, den = 7;

    result = uldiv(num, den);
}
```

See Also

ldiv(), div()

Return Value

Returns the the quotient and remainder as a uldiv_t structure.

UNGETCH

Synopsis

```
#include <conio.h>

void ungetch (char c)
```

Description

The **ungetch()** function will push back the character **c** onto the console stream, such that a subsequent getch() operation will return the character. At most one level of push back will be allowed.

See Also

getch(), getche()

UTOA

Synopsis

```
#include <stdlib.h>

char * utoa (char * buf, unsigned val, int base)
```

Description

The function **itoa** converts the unsigned contents of **val** into a string which is stored into **buf**. The conversion is performed according to the radix specified in **base**. **buf** is assumed to reference a buffer which has sufficent space allocated to it.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[10];
    utoi(buf, 1234, 16);
    printf("The buffer holds %s\n", buf);
}
```

See Also

strtol(), itoa(), ltoa(), ultoa()

Return Value

This routine returns a copy of the buffer into which the result is written.

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");
}
```

WRITETIMER*x*

Synopsis

```
#include <htc.h>

WRITETIMER0(unsigned int);
WRITETIMER1(unsigned int);
WRITETIMER2(unsigned int);
```

Description

The **WRITETIMER0**, **WRITETIMER1()** and **WRITETIMER2()** macros will assign a 16-Bit value to the TMRxL and TMRxH register pair of the corresponding device timer. Using this macro will ensure that the bytes are written in the correct order.

Example

```
#include <htc.h>

void
main (void)
{
    WRITETIMER1(0xF500);
}
```

See Also

READTIMER*x*()

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.

Appendix B

Error and Warning Messages

This chapter lists most error, warning and advisory messages from all HI-TECH C compilers, with an explanation of each message. Most messages have been assigned a unique number which appears in brackets before each message in this chapter, and which is also printed by the compiler when the message is issued. The messages shown here are sorted by their number. Un-numbered messages appear toward the end and are sorted alphabetically.

The name of the application(s) that could have produced the messages are listed in brackets opposite the error message. In some cases examples of code or options that could trigger the error are given. The use of * in the error message is used to represent a string that the compiler will substitute that is specific to that particular error.

Note that one problem in your C or assembler source code may trigger more than one error message.

(100) unterminated #if[n][def] block from line * ***(Preprocessor)***

A `#if` or similar block was not terminated with a matching `#endif`, e.g.:

```
#if INPUT        /* error flagged here */
void main(void)
{
  run();
}                /* no #endif was found in this module */
```

(101) #* may not follow #else ***(Preprocessor)***

A `#else` or `#elif` has been used in the same conditional block as a `#else`. These can only follow a `#if`, e.g.:

```
#ifdef FOO
  result = foo;
#else
  result = bar;
#elif defined(NEXT)   /* the #else above terminated the #if */
  result = next(0);
#endif
```

(102) #* must be in an #if ***(Preprocessor)***

The `#elif`, `#else` or `#endif` directive 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, e.g.:

```
#ifdef FOO
  result = foo;
#endif
  result = bar;
#elif defined(NEXT)   /* the #endif above terminated the #if */
  result = next(0);
#endif
```

(103) #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. Remove the directive to remove the error, but first check as to why the directive is there.

(104) preprocessor assertion failure ***(Preprocessor)***

The argument to a preprocessor `#assert` directive has evaluated to zero. This is a programmer induced error.

```
#assert SIZE == 4  /* size should never be 4 */
```

(105) no #asm before #endasm *(Preprocessor)*

A `#endasm` operator has been encountered, but there was no previous matching `#asm`, e.g.:

```
void cleardog(void)
{
  clrwdt
#endasm  /* this ends the in-line assembler, only where did it begin? */
}
```

(106) nested #asm directive *(Preprocessor)*

It is not legal to nest `#asm` directives. Check for a missing or misspelt `#endasm` directive, e.g.:

```
#asm
  move r0, #0aah
#asm              ; the previous #asm must be closed before opening another
  sleep
#endasm
```

(107) illegal # directive "*" *(Preprocessor, Parser)*

The compiler does not understand the `#` directive. It is probably a misspelling of a pre-processor `#` directive, e.g.:

```
#indef DEBUG  /* woops -- that should be #undef DEBUG */
```

(108) #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, e.g.:

```
#if            /* woops -- no argument to check */
  output = 10;
#else
  output = 20;
#endif
```

(109) #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 `< >`. Spaces should not be included, and the closing quote or bracket must be present. There should be nothing else on the line other than comments, e.g.:

```
#include stdio.h  /* woops -- should be: #include <stdio.h> */
```

(110) too many file arguments; usage: cpp [input [output]] ***(Preprocessor)***

`CPP` should be invoked with at most two file arguments. Contact HI-TECH Support if the preprocessor is being executed by a compiler driver.

(111) 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, e.g.:

```
#define ONE 1
/* elsewhere: */
#define ONE one  /* Is this correct? It will overwrite the first definition. */
```

(112) #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* , `)`, e.g.:

```
#define FOO(a, 2b)  bar(a, 2b)  /* 2b is not to be! */
```

(113) unterminated string in macro body ***(Preprocessor, Assembler)***

A macro definition contains a string that lacks a closing quote.

(114) illegal #undef argument ***(Preprocessor)***

The argument to `#undef` must be a valid name. It must start with a letter, e.g.:

```
#undef 6YYY  /* this isn't a valid symbol name */
```

(115) recursive macro definition of "*" defined by "*" ***(Preprocessor)***

The named macro has been defined in such a manner that expanding it causes a recursive expansion of itself!

(116) 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, e.g.:

```
#define FUNC(a, b) func(a+b)
FUNC(5, 6;    /* woops -- where is the closing bracket? */
```

(117) 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, e.g.:

```
#if FOO BAR  /* woops -- did you mean: #if FOO == BAR ? */
```

(118) #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.

(119) illegal #if line ***(Preprocessor)***

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

(120) 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, e.g.:

```
#if FOO * % BAR == 4  /* what is "* %" ? */
  #define BIG
#endif
```

(121) 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.

(122) unbalanced paren's, op is "*" ***(Preprocessor)***

The evaluation of a `#if` expression found mismatched parentheses. Check the expression for correct parenthesisation, e.g.:

```
#if ((A) + (B)  /* woops -- a missing ), I think */
  #define ADDED
#endif
```

(123) 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, e.g.:

```
#if XXX : YYY  /* did you mean:  #if COND ? XXX : YYY */
```

(124) 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, e.g.:

```
#if `YYY`  /* what are these characters doing here? */
  int m;
#endif
```

(125) illegal character (* decimal) in #if ***(Preprocessor)***

There is a non-printable character in a `#if` expression that has no business being there. Valid characters are the letters, digits and those comprising the acceptable operators, e.g.:

```
#if ^SYYY  /* what is this control characters doing here? */
  int m;
#endif
```

(126) can't use a string in an #if *(Preprocessor)*

The preprocessor does not allow the use of strings in `#if` expressions, e.g.:

```
#if MESSAGE > "hello"  /* no string operations allowed by the preprocessor */
#define DEBUG
#endif
```

(127) 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 and should be enclosed in parentheses, e.g.:

```
#if defined(a&b)   /* woops -- defined expects a name, not an expression */
  input = read();
#endif
```

(128) illegal operator in #if *(Preprocessor)*

A `#if` expression has an illegal operator. Check for correct syntax, e.g.:

```
#if FOO = 6  /* woops -- should that be: #if FOO == 5 ? */
```

(129) unexpected "\" in #if *(Preprocessor)*

The *backslash* is incorrect in the `#if` statement, e.g.:

```
#if FOO == \34
  #define BIG
#endif
```

(130) #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, e.g.:

```
#if sizeof(unt) == 2  /* woops -- should be: #if sizeof(int) == 2 */
  i = 0xFFFF;
#endif
```

(131) #if ... sizeof: illegal type combination *(Preprocessor)*

The preprocessor found an illegal type combination in the argument to `sizeof()` in a `#if` expression, e.g.

```
#if sizeof(short long int) == 2  /* short or long? make up your mind */
  i = 0xFFFF;
#endif
```

(132) #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, e.g.:

```
#if sizeof()  /* woops -- size of what? */
  i = 0;
#endif
```

(133) #if ... sizeof: bug, unknown type code 0x* *(Preprocessor)*

The preprocessor has made an internal error in evaluating a `sizeof()` expression. Check for a malformed type specifier. This is an internal error. Contact HI-TECH Software technical support with details.

(134) #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, e.g.:

```
#if sizeof(int == 2)  /* woops -- should be: #if sizeof(int) == 2 */
  i = 0xFFFF;
#endif
```

(135) #if bug, operand = * *(Preprocessor)*

The preprocessor has tried to evaluate an expression with an operator it does not understand. This is an internal error. Contact HI-TECH Software technical support with details.

(137) 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, e.g.:

```
#define cc(a, b) a ## 'b  /* the ' character will not lead to a valid token */
```

(138) strange character (*) after ## ***(Preprocessor)***

An unprintable 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, e.g.:

```
#define cc(a, b) a ## 'b  /* the ' character will not lead to a valid token */
```

(139) EOF in comment ***(Preprocessor)***

End of file was encountered inside a comment. Check for a missing closing comment flag, e.g.:

```
/* Here is the start of a comment. I'm not sure where I end, though
}
```

(140) can't open command file * ***(Driver, Preprocessor, Assembler, Linker)***

The command file specified could not be opened for reading. Confirm the spelling and path of the file specified on the command line, e.g.:

```
picc @communds
```

should that be:

```
picc @commands
```

(141) can't open output file * ***(Preprocessor, Assembler)***

An output file could not be created. Confirm the spelling and path of the file specified on the command line.

(142) can't open input file * ***(Preprocessor, Assembler)***

An input file could not be opened. Confirm the spelling and path of the file specified on the command line.

(144) too many nested #if statements ***(Preprocessor)***

`#if`, `#ifdef` etc. blocks may only be nested to a maximum of 32.

(145) cannot open include file "*" ***(Preprocessor)***

The named preprocessor 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. For files not in the current working directory or the standard compiler include directory, you may need to specify an additional include file path to the command-line driver, see Section 2.4.6.

(146) 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.

(147) too many include directories ***(Preprocessor)***

A maximum of 7 directories may be specified for the preprocessor to search for include files. The number of directories specified with the driver is too great.

(148) too many arguments for macro ***(Preprocessor)***

A macro may only have up to 31 parameters, as per the C Standard.

(149) 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.

(150) bug: illegal __ macro "*" ***(Preprocessor)***

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

(151) too many arguments in macro expansion ***(Preprocessor)***

There were too many arguments supplied in a macro invocation. The maximum number allowed is 31.

(152) bad dp/nargs in openpar: c = * ***(Preprocessor)***

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

(153) 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.

(155) work buffer overflow doing * ## ***(Preprocessor)***

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

(156) work buffer overflow: * ***(Preprocessor)***

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

(157) out of memory ***(Code Generator, Assembler, Optimiser)***

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

(158) invalid disable: * ***(Preprocessor)***

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

(159) too much pushback ***(Preprocessor)***

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

(160) 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.

(161) 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.

(163) unexpected text in #control line ignored *(Preprocessor)*

This warning occurs when extra characters appear on the end of a control line, e.g. The extra text will be ignored, but a warning is issued. It is preferable (and in accordance with Standard C) to enclose the text as a comment, e.g.:

```
#if defined(END)
  #define NEXT
#endif END     /* END would be better in a comment here */
```

(164) 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, e.g.:

```
#include "STDIO.H"  /* is this meant to be stdio.h ? */
```

(164) included file * was converted to lower case *(Preprocessor)*

The `#include` file name had to be converted to lowercase before it could be opened.

```
#include <STDIO.H>  /* woops -- should be: #include <stdio.h> */
```

(166) -S, too few values specified in * *(Preprocessor)*

The list of values to the preprocessor (CPP) `-S` option is incomplete. This should not happen if the preprocessor is being invoked by the compiler driver. The values passes to this option represent the sizes of `char`, `short`, `int`, `long`, `float` and `double` types.

(167) -S, too many values, "*" unused *(Preprocessor)*

There were too many values supplied to the -S preprocessor option. See the Error Message `-s, too few values specified in *` on page 310.

(168) unknown option "*" *(Hexmate, Preprocessor)*

This option to the preprocessor/hexmate is not recognized.

(169) strange character after # (*) ***(Preprocessor)***

There is an unexpected character after `#`.

(170) symbol "*" not defined in #undef ***(Preprocessor)***

The symbol supplied as argument to `#undef` was not already defined. This warning may be disabled with some compilers. This warning can be avoided with code like:

```
#ifdef SYM
  #undef SYM  /* only undefine if defined */
#endif
```

(171) wrong number of macro arguments for "*" - * instead of * ***(Preprocessor)***

A macro has been invoked with the wrong number of arguments, e.g.:

```
#define ADD(a, b) (a+b)
ADD(1, 2, 3)            /* woops -- only two arguments required */
```

(172) 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, e.g.:

```
#define str(x) #y  /* woops -- did you mean x instead of y? */
```

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.

(173) 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. This warning may be disabled with some compilers. Example:

```
#if FOO+BAR    /* e.g. FOO was never #defined */
  #define GOOD
#endif
```

(174) 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, e.g.:

```
#if CHAR == 'ab'
  #define MULTI
#endif
```

(175) division 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, e.g.:

```
#if foo/0  /* divide by 0: was this what you were intending? */
  int a;
#endif
```

(176) 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.

(177) 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.

(179) 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, e.g.:

```
output = 0;  /* a comment that was left unterminated
flag = TRUE; /* another comment: hey, where did this line go? */
```

(180) unterminated comment in included file ***(Preprocessor)***

Comments begun inside an included file must end inside the included file.

(181) non-scalar types can't be converted *(Parser)*

You can't convert a structure, union or array to another type, e.g.:

```
struct TEST test;
struct TEST * sp;
sp = test;       /* woops -- did you mean: sp = &test; ? */
```

(182) illegal conversion *(Parser)*

This expression implies a conversion between incompatible types, e.g. a conversion of a structure type into an integer, e.g.:

```
struct LAYOUT layout;
int i;
layout = i;          /* an int cannot be converted into a struct */
```

Note that even if a structure only contains an `int`, for example, it cannot be assigned to an `int` variable, and vice versa.

(183) function or function pointer required *(Parser)*

Only a function or function pointer can be the subject of a function call, e.g.:

```
int a, b, c, d;
a = b(c+d);     /* b is not a function -- did you mean a = b*(c+d) ? */
```

(184) 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.

(185) function does not take arguments *(Parser, Code Generator)*

This function has no parameters, but it is called here with one or more arguments, e.g.:

```
int get_value(void);
void main(void)
{
```

```
  int input;
  input = get_value(6);  /* woops -- the parameter should not be here */
}
```

(186) too many arguments ***(Parser)***

This function does not accept as many arguments as there are here.

```
void add(int a, int b);
add(5, 7, input);              /* this call has too many arguments */
```

(187) too few arguments ***(Parser)***

This function requires more arguments than are provided in this call, e.g.:

```
void add(int a, int b);
add(5);                  /* this call needs more arguments */
```

(188) constant expression required ***(Parser)***

In this context an expression is required that can be evaluated to a constant at compile time, e.g.:

```
int a;
switch(input) {
  case a:       /* woops -- you cannot use a variable as part of a case label */
    input++;
}
```

(189) illegal type for array dimension ***(Parser)***

An array dimension must be either an integral type or an enumerated value.

```
int array[12.5];  /* woops -- twelve and a half elements, eh? */
```

(190) illegal type for index expression ***(Parser)***

An index expression must be either integral or an enumerated value, e.g.:

```
int i, array[10];
i = array[3.5];   /* woops -- exactly which element do you mean? */
```

(191) 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`, e.g.:

```
lip = (long [])input;  /* woops -- maybe: lip = (long *)input */
```

(192) undefined identifier: * ***(Parser)***

This symbol has been used in the program, but has not been defined or declared. Check for spelling errors if you think it has been defined.

(193) not a variable identifier: * ***(Parser)***

This identifier is not a variable; it may be some other kind of object, e.g. a label.

(194)) expected ***(Parser)***

A *closing parenthesis*, `)`, was expected here. This may indicate you have left out this character in an expression, or you have some other syntax error. The error is flagged on the line at which the code first starts to make no sense. This may be a statement following the incomplete expression, e.g.:

```
if(a == b  /* the closing parenthesis is missing here */
  b = 0;   /* the error is flagged here */
```

(195) expression syntax ***(Parser)***

This expression is badly formed and cannot be parsed by the compiler, e.g.:

```
a /=% b;  /* woops -- maybe that should be: a /= b; */
```

(196) struct/union required ***(Parser)***

A structure or union identifier is required before a dot `.`, e.g.:

```
int a;
a.b = 9;  /* woops -- a is not a structure */
```

(197) struct/union member expected ***(Parser)***

A structure or union member name must follow a dot (".") or arrow ("->").

(198) undefined struct/union: * ***(Parser)***

The specified structure or union tag is undefined, e.g.

```
struct WHAT what;  /* a definition for WHAT was never seen */
```

(199) 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, e.g.:

```
struct FORMAT format;
if(format)            /* this operand must be a scaler type */
  format.a = 0;
```

(200) 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, e.g.:

```
int * proc(register int in)
{
  int * ip = &in;      /* woops -- in may not have an address to take */
  return ip;
}
```

(201) can't take this address ***(Parser)***

The expression which was the operand of the `&` operator is not one that denotes memory storage ("an lvalue") and therefore its address can not be defined, e.g.:

```
ip = &8;  /* woops -- you can't take the address of a literal */
```

(202) 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, e.g.:

```
int array[10];
int * ip;
char c;
array = ip;    /* array is not a variable, it cannot be written to */
```

A typecast does not yield an lvalue, e.g.:

```
(int)c = 1;    /* the contents of c cast to int is only a intermediate value */
```

However you can write this using pointers:

```
*(int *)&c = 1
```

(203) illegal operation on a bit variable ***(Parser)***

Not all operations on `bit` variables are supported. This operation is one of those, e.g.:

```
bit   b;
int * ip;
ip = &b;  /* woops -- cannot take the address of a bit object */
```

(204) void function cannot return value ***(Parser)***

A void function cannot return a value. Any `return` statement should not be followed by an expression, e.g.:

```
void run(void)
{
  step();
  return 1;     /* either run should not be void, or remove the 1 */
}
```

(205) integral type required ***(Parser)***

This operator requires operands that are of integral type only.

(206) 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.

(207) simple type required for * ***(Parser)***

A simple type (i.e. not an array or structure) is required as an operand to this operator.

(208) operands of * not same type ***(Parser)***

The operands of this operator are of different pointer, e.g.:

```
int * ip;
char * cp, * cp2;
cp = flag ? ip : cp2;  /* result of ? : will either be int * or char * */
```

Maybe you meant something like:

```
cp = flag ? (char *)ip : cp2;
```

(209) type conflict ***(Parser)***

The operands of this operator are of incompatible types.

(210) bad size list ***(Parser)***

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

(212) missing number after pragma "pack" ***(Parser)***

The pragma `pack` requires a decimal number as argument. This specifies the alignment of each member within the structure. Use this with caution as some processors enforce alignment and will not operate correctly if word fetches are made on odd boundaries, e.g.:

```
#pragma pack  /* what is the alignment value */
```

Maybe you meant something like:

```
#pragma pack 2
```

(214) missing number after pragma "interrupt_level" ***(Parser)***

The pragma `interrupt_level` requires an argument from 0 to 7.

(215) missing argument to "pragma switch" ***(Parser)***

The pragma switch requires an argument of `auto`, `direct` or `simple`, e.g.:

```
#pragma switch  /* woops -- this requires a switch mode */
```

maybe you meant something like:

```
#pragma switch simple
```

(216) 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, e.g.:

```
#pragma psect  /* woops -- this requires an psect to redirect */
```

maybe you meant something like:

```
#pragma psect text=specialtext
```

(218) missing name after pragma "inline" ***(Parser)***

The `inline` pragma expects the name of a function to follow. The function name must be recognized by the code generator for it to be expanded; other functions are not altered, e.g.:

```
#pragma inline  /* what is the function name? */
```

maybe you meant something like:

```
#pragma inline memcpy
```

(219) missing name after pragma "printf_check" ***(Parser)***

The `printf_check` pragma expects the name of a function to follow. This specifies printf-style format string checking for the function, e.g.

```
#pragma printf_check  /* what function is to be checked? */
```

Maybe you meant something like:

```
#pragma printf_check sprintf
```

Pragmas for all the standard printf-like function are already contained in `<stdio.h>`.

(220) exponent expected ***(Parser)***

A floating point constant must have at least one digit after the `e` or `E`., e.g.:

```
float f;
f = 1.234e;  /* woops -- what is the exponent? */
```

(221) hex digit expected ***(Parser)***

After `0x` should follow at least one of the hex digits `0-9` and `A-F` or `a-f`, e.g.:

```
a = 0xg6;  /* woops -- was that meant to be a = 0xf6 ? */
```

(222) binary digit expected ***(Parser)***

A binary digit was expected following the `0b` format specifier, e.g.

```
i = 0bf000;  /* wooops -- f000 is not a base two value */
```

(223) 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". For example:

```
int a = 058;  /* a leading 0 implies octal which has digits 0 thru 7 */
```

(225) missing character in character constant ***(Parser)***

The character inside the single quotes is missing, e.g.:

```
char c = ’;  /* the character value of what? */
```

(226) char const too long ***(Parser)***

A character constant enclosed in single quotes may not contain more than one character, e.g.:

```
c = ’12’;  /* woops -- only one character may be specified */
```

(227) "." 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. (An *ellipsis* is used in function prototypes to indicate a variable number of parameters.)

Either `..` was meant to be an *ellipsis* symbol which would require you to add an extra dot, or it was meant to be a *structure member operator* which would require you remove one dot.

(228) illegal character (*) ***(Parser)***

This character is illegal in the C code. Valid characters are the letters, digits and those comprising the acceptable operators, e.g.:

```
c = `a`;  /* woops -- did you mean c = 'a'; ? */
```

(229) unknown qualifier "*" given to -A ***(Parser)***

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

(230) missing arg to -A ***(Parser)***

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

(231) unknown qualifier "*" given to -I ***(Parser)***

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

(232) missing arg to -I ***(Parser)***

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

(233) bad -Q option * ***(Parser)***

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

(234) close error (disk space?) ***(Parser)***

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

(236) simple integer expression required ***(Parser)***

A simple integral expression is required after the operator `@`, used to associate an absolute address with a variable, e.g.:

```
int address;
char LOCK @ address;
```

(237) function "*" redefined *(Parser)*

More than one definition for a function has been encountered in this module. Function overloading is illegal, e.g.:

```
int twice(int a)
{
  return a*2;
}
long twice(long a)  /* only one prototype & definition of rv can exist */
{
  return a*2;
}
```

(238) illegal initialisation *(Parser)*

You can't initialise a `typedef` declaration, because it does not reserve any storage that can be initialised, e.g.:

```
typedef unsigned int uint = 99;  /* woops -- uint is a type, not a variable */
```

(239) identifier redefined: * (from line *) *(Parser)*

This identifier has already been defined in the same scope. It cannot be defined again, e.g.:

```
int a;  /* a filescope variable called "a" */
int a;  /* this attempts to define another with the same name */
```

Note that variables with the same name, but defined with different scopes are legal, but not recommended.

(240) too many initializers *(Parser)*

There are too many initializers for this object. Check the number of initializers against the object definition (array or structure), e.g.:

```
int ivals[3] = { 2, 4, 6, 8};  /* three elements, but four initializers */
```

(241) initialization syntax *(Parser)*

The initialisation of this object is syntactically incorrect. Check for the correct placement and number of braces and commas, e.g.:

```
int iarray[10] = {{'a', 'b', 'c'};  /* woops -- one two many {s */
```

(242) illegal type for switch expression *(Parser)*

A `switch` operation must have an expression that is either an integral type or an enumerated value, e.g:

```
double d;
switch(d) {  /* woops -- this must be integral */
  case '1.0':
    d = 0;
}
```

(243) inappropriate break/continue *(Parser)*

A `break` or `continue` statement has been found that is not enclosed in an appropriate control structure. A `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, e.g.:

```
switch(input) {
  case 0:
    if(output == 0)
      input = 0xff;
    }               /* woops -- this shouldn't be here and closed the switch */
    break;          /* this should be inside the switch */
```

(244) default case redefined *(Parser)*

There is only allowed to be one `default` label in a switch statement. You have more than one, e.g.:

```
switch(a) {
default:     /* if this is the default case... */
  b = 9;
  break;
default:     /* then what is this? */
  b = 10;
  break;
```

(245) "default" not in switch *(Parser)*

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

If there is a `switch` statement before this `default` label, there may be one too many closing braces in the `switch` code which would prematurely terminate the `switch` statement. See example for Error Message `'case' not in switch` on page 324.

(246) "case" not in switch *(Parser)*

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

If there is a `switch` statement before this `case` label, there may be one too many closing braces in the `switch` code which would prematurely terminate the `switch` statement, e.g.:

```
switch(input) {
  case '0':
    count++;
    break;
  case '1':
    if(count>MAX)
      count= 0;
    }          /* woops -- this shouldn't be here */
    break;
  case '2':    /* error flagged here */
```

(247) 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, e.g.:

```
start:
  if(a > 256)
    goto end;
start:           /* error flagged here */
  if(a == 0)
    goto start;  /* which start label do I jump to? */
```

(248) 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, e.g.:

```
/* here is a comment which I have forgotten to close...
if(a > b) {
  c = 0;    /* ... that will be closed here, thus removing the "if" */
else        /* my "if" has been lost */
  c = 0xff;
```

(249) 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, e.g.:

```
void set(char a)
{
  PORTA = a;
                  /* the closing brace was left out here */
void clear(void)  /* error flagged here */
{
  PORTA = 0;
}
```

(251) 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, e.g.:

```
extern int array[5];
int array[10];        /* woops -- has it 5 or 10 elements? */
```

(252) argument * conflicts with prototype *(Parser)*

The argument specified (argument 0 is the left most argument) of this function definition does not agree with a previous prototype for this function, e.g.:

```
extern int calc(int, int);   /* this is supposedly calc's prototype */
int calc(int a, long int b)  /* hmmm -- which is right? */
{                            /* error flagged here */
  return sin(b/a);
}
```

(253) 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.

```
extern int calc(int);   /* this is supposedly calc's prototype */
int calc(int a, int b)  /* hmmm -- which is right? */
{                       /* error flagged here */
  return a + b;
}
```

(254) undefined *: * ***(Parser)***

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

(255) 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, e.g.:

```
struct {
  int a, b, c;
} data;
if(data.d)     /* woops -- there is no member d in this structure */
  return;
```

(256) too much indirection ***(Parser)***

A pointer declaration may only have 16 levels of indirection.

(257) only register storage class allowed ***(Parser)***

The only storage class allowed for a function parameter is `register`, e.g.:

```
void process(static int input)
```

(258) 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. For example:

```
typedef volatile int vint;
volatile vint very_vol;   /* woops -- this results in two volatile qualifiers */
```

(259) can't be both far and near ***(Parser)***

It is illegal to qualify a type as both `far` and `near`, e.g.:

```
far near int spooky;  /* woops -- choose either far or near, not both */
```

(260) undefined enum tag: * ***(Parser)***

This enum tag has not been defined, e.g.:

```
enum WHAT what;  /* a definition for WHAT was never seen */
```

(261) member * redefined ***(Parser)***

This name of this member of the struct or union has already been used in this `struct` or `union`, e.g.:

```
struct {
  int a;
  int b;
  int a;  /* woops -- a different name is required here */
} input;
```

(262) struct/union redefined: * ***(Parser)***

A structure or union has been defined more than once, e.g.:

```
struct {
  int a;
} ms;
struct {
  int a;
} ms;    /* was this meant to be the same name as above? */
```

(263) 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, e.g.:

```
struct {
  int a;
  int get(int);  /* this should be a pointer: int (*get)(int); */
} object;
```

(264) bad bitfield type *(Parser)*

A bitfield may only have a type of `int` (`signed` or `unsigned`), e.g.:

```
struct FREG {
  char b0:1;   /* woops -- these must be part of an int, not char */
  char   :6;
  char b7:1;
} freg;
```

(265) 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, e.g.:

```
struct {
  unsigned first:  /* woops -- should be: unsigned first; */
  unsigned second;
} my_struct;
```

If this was meant to be a structure with bitfields, then the following illustrates an example:

```
struct {
  unsigned first : 4;  /* 4 bits wide */
  unsigned second: 4;  /* another 4 bits */
} my_struct;
```

(266) 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, e.g.:

```
struct {
  static int first;  /* no additional qualifiers may be present with members */
} ;
```

(267) bad storage class ***(Code Generator)***

The code generator has encounterd a variable definition whose storage class is invalid, e.g.:

```
auto int foo;            /* auto not permitted with global variables */
int power(static int a)  /* paramters may not be static */
{
  return foo * a;
}
```

(268) inconsistent storage class ***(Parser)***

A declaration has conflicting storage classes. Only one storage class should appear in a declaration, e.g.:

```
extern static int where;  /* so is it static or extern? */
```

(269) inconsistent type ***(Parser)***

Only one basic type may appear in a declaration, e.g.:

```
int float if;  /* is it int or float? */
```

(270) can't be register ***(Parser)***

Only function parameters or `auto` variables may be declared using the `register` qualifier, e.g.:

```
register int gi;                /* this cannot be qualified register */
int process(register int input)  /* this is okay */
{
  return input + gi;
}
```

(271) can't be long ***(Parser)***

Only `int` and `float` can be qualified with `long`.

```
long char lc;  /* what? */
```

(272) can't be short ***(Parser)***

Only `int` can be modified with `short`, e.g.:

```
short float sf;  /* what? */
```

(273) 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, e.g.:

```
signed unsigned int confused;  /* which is it? signed or unsigned? */
```

(274) can't be unsigned ***(Parser)***

A floating point type cannot be made `unsigned`, e.g.:

```
unsigned float uf;  /* what? */
```

(275) ... 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, i.e. K&R-style non-prototype function definitions. For example:

```
int kandr(a, b, ...)  /* K&R-style non-prototyped function definition */
  int a, b;
{
```

(276) type specifier required for proto arg ***(Parser)***

A type specifier is required for a prototyped argument. It is not acceptable to just have an identifier.

(277) 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), e.g.:

```
int plus(int a, b)  /* woops -- a is prototyped, b is not */
int b;
{
  return a + b;
}
```

(278) argument redeclared: * *(Parser)*

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

```
int calc(int a, int a)  /* you cannot have two parameters called "a" */
```

(279) can't initialize arg *(Parser)*

A function argument can't have an initialiser in a declaration. The initialisation of the argument happens when the function is called and a value is provided for the argument by the calling function, e.g.:

```
extern int proc(int a = 9);  /* woops -- a is initialized when proc is called */
```

(280) can't have array of functions *(Parser)*

You can't define an array of functions. You can however define an array of pointers to functions, e.g.:

```
int * farray[]();  /* woops -- should be: int (* farray[])(); */
```

(281) 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.

(282) functions can't return arrays *(Parser)*

A function can return only a scalar (simple) type or a structure. It cannot return an array.

(283) 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 as a constant expression, e.g.:

```
enum { one = 1, two };
int get_element(int array[two][])  /* should be, e.g.: int array[][7] */
{
  return array[1][6];
}
```

(285) 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, e.g.:

```
void interrupt(void)  /* what is the name of this function? */
{
}
```

(286) 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.

(287) 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 variable, e.g.:

```
bit barray[10];  /* wrong -- no bit arrays */
bit * bp;        /* wrong -- no pointers to bit variables */
```

(288) only functions may be void *(Parser)*

A variable may not be `void`. Only a function can be `void`, e.g.:

```
int a;
void b;  /* this makes no sense */
```

(289) only functions may be qualified interrupt *(Parser)*

The qualifier `interrupt` may not be applied to anything except a function, e.g.:

```
interrupt int input;  /* variables cannot be qualified interrupt */
```

(290) illegal function qualifier(s) *(Parser)*

A qualifier has been applied to a function which makes no sense in this context. Some qualifier only make sense when used with an lvalue, e.g. const or volatile. This may indicate that you have forgotten out a star `*` indicating that the function should return a pointer to a qualified object, e.g.

```
const char ccrv(void)  /* woops -- did you mean const * char ccrv(void) ? */
{                      /* error flagged here */
  return ccip;
}
```

(291) 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, e.g.:

```
int process(input)
int unput;         /* woops -- that should be int input; */
{
}
```

(292) 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.

(293) bad size in index_type *(Parser)*

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

(294) can't allocate * bytes of memory *(Code Generator, Hexmate)*

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

(295) 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.

(297) bad arg (*) to tysize ***(Parser)***

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

(298) EOF in #asm ***(Preprocessor)***

An end of file has been encountered inside a `#asm` block. This probably means the `#endasm` is missing or misspelt, e.g.:

```
#asm
  mov  r0, #55
  mov  [r1], r0
}               /* woops -- where is the #endasm */
```

(300) unexpected EOF ***(Parser)***

An end-of-file in a C module was encountered unexpectedly, e.g.:

```
void main(void)
{
  init();
  run();    /* is that it? What about the close brace */
```

(301) EOF on string file ***(Parser)***

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

(302) can't reopen * ***(Parser)***

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

(303) no memory for string buffer ***(Parser)***

The parser 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.

(305) can't open * ***(Code Generator, Assembler, Optimiser, Cromwell)***

An input file could not be opened. Confirm the spelling and path of the file specified on the command line.

(306) out of far memory ***(Code Generator)***

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

(307) too many qualifier names ***(Parser)***

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

(308) 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.

(309) too many symbols ***(Assembler)***

There are too many symbols for the assembler's symbol table. Reduce the number of symbols in your program.

(310)] expected ***(Parser)***

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

```
process(carray[idx);  /* woops -- should be: process(carray[idx]); */
```

(313) 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, e.g.:

```
int get_value(a, b);  /* the function block must follow, not a semicolon */
```

(314) ; expected ***(Parser)***

A *semicolon* is missing from a statement. A close brace or keyword was found following a statement with no terminating *semicolon*, e.g.:

```
while(a) {
  b = a--  /* woops -- where is the semicolon? */
}          /* error is flagged here */
```

Note: Omitting a semicolon from statements not preceeding a close brace or keyword typically results in some other error being issed for the following code which the parser assums to be part of the original statement.

(315) { expected ***(Parser)***

An *opening brace* was expected here. This error may be the result of a function definition missing the *opening brace*, e.g.:

```
void process(char c)      /* woops -- no opening brace after the prototype */
  return max(c, 10) * 2;  /* error flagged here */
}
```

(316) } expected ***(Parser)***

A *closing brace* was expected here. This error may be the result of a initialized array missing the *closing brace*, e.g.:

```
char carray[4] = { 1, 2, 3, 4;  /* woops -- no closing brace */
```

(317) (expected ***(Parser)***

An *opening parenthesis*, `(`, was expected here. This must be the first token after a `while`, `for`, `if`, `do` or `asm` keyword, e.g.:

```
if a == b  /* should be: if(a == b) */
  b = 0;
```

(318) string expected ***(Parser)***

The operand to an `asm` statement must be a string enclosed in parentheses, e.g.:

```
asm(nop);  /* that should be asm("nop");
```

(319) while expected ***(Parser)***

The keyword `while` is expected at the end of a `do` statement, e.g.:

```
do {
  func(i++);
}             /* do the block while what condition is true? */
if(i > 5)     /* error flagged here */
  end();
```

(320) : expected ***(Parser)***

A *colon* is missing after a `case` label, or after the keyword `default`. This often occurs when a *semicolon* is accidentally typed instead of a *colon*, e.g.:

```
switch(input) {
  case 0;         /* woops -- that should have been: case 0: */
    state = NEW;
```

(321) label identifier expected ***(Parser)***

An identifier denoting a label must appear after `goto`, e.g.:

```
if(a)
  goto 20;  /* this is not BASIC -- a valid C label must follow a goto */
```

(322) 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, e.g.:

```
enum 1, 2;  /* should be, e.g.: enum {one=1, two }; */
```

(323) struct/union tag or "{" expected ***(Parser)***

An identifier denoting a structure or union or an opening brace must follow a `struct` or `union` keyword, e.g.:

```
struct int a;  /* this is not how you define a structure */
```

You might mean something like:

```
struct {
  int a;
} my_struct;
```

(324) 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, e.g.:

```
printf("%d - %d", low, high, median);  /* woops -- missed a placeholder? */
```

(325) 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, e.g.:

```
printf("%l", lll);  /* woops -- maybe: printf("%ld", lll); */
```

(326) long argument required ***(Parser)***

A long argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments, e.g.:

```
printf("%lx", 2);  /* woops -- maybe you meant: printf("%lx", 2L);
```

(328) integral argument required ***(Parser)***

An integral argument is required for this printf-style format specifier. Check the number and order of format specifiers and corresponding arguments, e.g.:

```
printf("%d", 1.23);  /* woops -- either wrong number or wrong placeholder */
```

(329) 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.

```
printf("%f", 44);  /* should be: printf("%f", 44.0); */
```

(330) 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.

(331) 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, e.g.:

```
printf("%d - %d", low);  /* woops -- where is the other value to print? */
```

(332) interrupt_level should be 0 to 7 ***(Parser)***

The pragma `interrupt_level` must have an argument from 0 to 7, e.g.:

```
#pragma interrupt_level  /* woops -- what is the level */
void interrupt isr(void)
{
  /* isr code goes here */
}
```

(333) unrecognized qualifier name after "strings" ***(Parser)***

The `pragma strings` was passed a qualifier that was not identified, e.g.:

```
#pragma strings cinst  /* woops -- should that be #pragma strings const ? */
```

(335) unknown pragma * ***(Parser)***

An unknown `pragma` directive was encountered, e.g.:

```
#pragma rugsused w  /* I think you meant regsused */
```

(336) string concatenation across lines ***(Parser)***

Strings on two lines will be concatenated. Check that this is the desired result, e.g.:

```
char * cp = "hi"
  "there";         /* this is okay, but is it what you had intended? */
```

(337) line does not have a newline on the end ***(Parser)***

The last line in the file is missing the *newline* (operating system dependent character) 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.

(338) can't create * file "*" ***(Code Generator, Assembler, Linker, Optimiser)***

The application tried to create the named file, but it could not be created. Check that all file pathnames are correct.

(338) can't create * file "*" ***(Linker, Code Generator Driver)***

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.

(339) 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, e.g.:

```
extern int other = 99;  /* if it's extern and not allocated storage,
                           how can it be initialized? */
```

(343) 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`, e.g.:

```
int mydiv(double a, int b)
{
  if(b != 0)
    return a/b;   /* what about when b is 0? */
}                 /* warning flagged here */
```

(344) 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, e.g.:

```
int get_value(void)
{
  if(flag)
    return val++;
  return;         /* what is the return value in this instance? */
}
```

(345) unreachable code *(Parser)*

This section of code will never be executed, because there is no execution path by which it could be reached, e.g.:

```
while(1)          /* how does this loop finish? */
  process();
flag = FINISHED;  /* how do we get here? */
```

(346) 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, e.g.:

```
int input;           /* input has filescope */
void process(int a)
{
  int input;         /* local blockscope input */
  a = input;         /* this will use the local variable. Is this right? */
```

(347) external declaration inside function *(Parser)*

A function contains an `extern` declaration. This is legal but is invariably not desirable 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. For example:

```
int process(int a)
{
  extern int away;  /* this would be better outside the function */
  return away + a;
}
```

(348) 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. An `auto` variable may be qualified with `static`, but it is then no longer `auto`.

(349) 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, e.g.:

```
int process(input)
int input;          /* warning flagged here */
{
}
```

This would be better written:

```
int process(int input)
{
}
```

(350) unused *: * (from line *) ***(Parser)***

The indicated object was never used in the function or module being compiled. Either this object is redundant, or the code that was meant to use it was excluded from compilation or misspelt the name of the object. Note that the symbols `rcsid` and `sccsid` are never reported as being unused.

(352) 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, e.g.:

```
double inc_flt(f)  /* the parameter f will be converted to double type */
float f;           /* warning flagged here */
{
  return f * 2;
}
```

(353) sizeof external array "*" is zero *(Parser)*

The size of an external array evaluates to zero. This is probably due to the array not having an explicit dimension in the extern declaration.

(354) 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.

(355) implicit 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.:

```
signed char sc;
unsigned int ui;
ui = sc;         /* if sc contains 0xff, ui will contain 0xffff for example */
```

will perform a sign extension of the `char` variable to the longer type. If you do not want this to take place, use a cast, e.g.:

```
ui = (unsigned char)sc;
```

(356) 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.

```
double dd;
int i;
i = dd;    /* is this really what you meant? */
```

If you do intend to use an expression like this, then indicate that this is so by a cast:

```
i = (int)dd;
```

(357) 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. This may also mean you have forgotten the `&` address operator, e.g.:

```
int * ip;
int i;
ip = i;    /* woops -- did you mean ip = &i ? */
```

If you do intend to use an expression like this, then indicate that this is so by a cast:

```
ip = (int *)i;
```

(358) 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. This may also mean you have forgotten the `*` dereference operator, e.g.:

```
int * ip;
int i;
i = ip;    /* woops -- did you mean i = *ip ? */
```

If you do intend to use an expression like this, then indicate that this is so by a cast:

```
i = (int)ip;
```

(359) 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, e.g.:

```
long input;
char * cp;
cp = &input;  /* is this correct? */
```

This is common way of accessing bytes within a multi-byte variable. To indicate that this is the intended operation of the program, use a cast:

```
cp = (char *)&input;  /* that's better */
```

This warning may also occur when converting between pointers to objects which have the same type, but which have different qualifiers, e.g.:

```
char * cp;
cp = "I am a string of characters";  /* yes, but what sort of characters? */
```

If the default type for string literals is `const char *`, then this warning is quite valid. This should be written:

```
const char * cp;
cp = "I am a string of characters";  /* that's better */
```

Omitting a qualifier from a pointer type is often disastrous, but almost certainly not what you intend.

(360) 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. This warning will not be issued when accessing an array element via a pointer variable, e.g.:

```
int i, * ip, input[10];
i = input[-2];           /* woops -- this element doesn't exist */
ip = &input[5];
i = ip[-2];              /* this is okay */
```

(361) 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. For example:

```
void set(long a, int b);  /* I may prevent an error arising from calls below */
void main(void)
{
  set(10L, 6);    /* by here a prototype for set should have seen */
}
```

(362) 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, e.g.:

```
int array[5];
int * ip;
ip = &array;  /* array is a constant, not a variable; the & is redundant. */
```

(364) attempt to modify * object *(Parser)*

Objects declared `const` or `code` may not be assigned to or modified in any other way by your program. The effect of attempting to modify such an object is compiler-specific.

```
const int out = 1234;  /* "out" is read only */
out = 0;               /* woops -- writing to a read-only object */
```

(365) pointer to non-static object returned *(Parser)*

This function returns a pointer to a non-`static` (e.g. `auto`) variable. This is likely to be an error, since the storage associated with automatic variables becomes invalid when the function returns, e.g.:

```
char * get_addr(void)
{
  char c;
  return &c;  /* returning this is dangerous; the pointer could be dereferenced */
}
```

(366) 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.

(367) 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.

```
void main(void)
{
  set(10L, 6);    /* at this point the compiler assumes set is extern... */
}
static void set(long a, int b)  /* now it finds out otherwise */
{
  PORTA = a + b;
}
```

(368) 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. Either remove the dimension from the parameter, or define the parameter using pointer syntax, e.g.:

```
int get_first(int array[10])  /* param should be: "int array[]" or "int *" */
{                             /* warning flagged here */
  return array[0];
}
```

(369) 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`, e.g.:

```
struct {
  signed int sign: 1;    /* this must be unsigned */
  signed int value: 15;
} ;
```

(371) 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, e.g.:

```
char c;
i;      /* don't let the compiler make assumptions, use : int i */
func(); /* ditto, use: extern int func(int); */
```

(372) , expected ***(Parser)***

A *comma* was expected here. This could mean 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, e.g.:

```
unsigned char a;
unsigned chat b; /* thinks: chat & b are unsigned, but where is the comma? */
```

(375) unknown FNREC type * ***(Linker)***

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

(376) 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.

(378) can't create * file "*" ***(Hexmate)***

This type of file could not be created. Is the file or a file by this name already in use?

(379) bad record type * ***(Linker)***

This is an internal compiler error. Ensure the object file is a valid HI-TECH object file. Contact HI-TECH Software technical support with details.

(380) unknown record type: * ***(Linker)***

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

(381) record too long (*): * ***(Linker)***

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

(382) incomplete record: type = *, length = * ***(Dump, Xstrip)***

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. Contact HI-TECH Support with details.

(383) text record has length too small: * ***(Linker)***

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

(384) assertion failed: file *, line *, expr * ***(Linker, Parser)***

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

(386) can't open error file * ***(Linker)***

The error file specified using the -e linker option could not be opened.

(387) illegal or too many -g flags ***(Linker)***

There has been more than one linker `-g` option, or the `-g` option did not have any arguments following. The arguments specify how the segment addresses are calculated.

(388) duplicate -m flag ***(Linker)***

The map file name has been specified to the linker for a second time. This should not occur if you are using a compiler driver. If invoking the linker manually, ensure that only one instance of this option is present on the command line. See Section 5.7.9 for information on the correct syntax for this option.

(389) illegal or too many -o flags ***(Linker)***

This linker `-o` flag is illegal, or another `-o` option has been encountered. A `-o` option to the linker must be immediately followed by a filename with no intervening space.

(390) 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*.

(391) missing arg to -Q ***(Linker)***

The `-Q` linker option requires the machine type for an argument.

(392) missing arg to -u ***(Linker)***

The `-U` (undefine) option needs an argument.

(393) missing arg to -w ***(Linker)***

The `-W` option (listing width) needs a numeric argument.

(394) duplicate -d or -h flag ***(Linker)***

The symbol file name has been specified to the linker for a second time. This should not occur if you are using a compiler driver. If invoking the linker manually, ensure that only one instance of either of these options is present on the command line.

(395) missing arg to -j ***(Linker)***

The maximum number of errors before aborting must be specified following the `-j` linker option.

(396) illegal flag -* ***(Linker)***

This linker option is unrecognized.

(398) 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.

(400) bad object code format ***(Linker)***

This is an internal compiler error. The object code format of an object file is invalid. Ensure it is a valid HI-TECH object file. Contact HI-TECH Software technical support with details.

(401) cannot get memory ***(Linker)***

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

(404) bad maximum length value to -<digits> ***(Objtohex)***

The first value to the OBJTOHEX `-n,m` hex length/rounding option is invalid.

(405) bad record size rounding value to -<digits> ***(Objtohex)***

The second value to the OBJTOHEX `-n,m` hex length/rounding option is invalid.

(410) bad combination of flags ***(Objtohex)***

The combination of options supplied to `OBJTOHEX` is invalid.

(412) text does not start at 0 ***(Objtohex)***

Code in some things must start at zero. Here it doesn't.

(413) write error on * ***(Assembler, Linker, Cromwell)***

A write error occurred on the named file. This probably means you have run out of disk space.

(414) read error on * ***(Linker)***

The linker encountered an error trying to read this file.

(415) text offset too low ***(Objtohex)***

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

(416) bad character in extended Tekhex line (*) ***(Objtohex)***

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

(417) seek error ***(Linker)***

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

(418) image too big ***(Objtohex)***

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

(419) 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. The object output files from the assembler are relocatable, not absolute. The object file output of the linker is absolute.

(420) too many relocation items ***(Objtohex)***

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

(421) too many segments ***(Objtohex)***

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

(422) no end record ***(Linker)***

This object file has no end record. This probably means it is not an object file. Contact HI-TECH Support if the object file was generated by the compiler.

(423) 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. Contact HI-TECH Support with details if the object file was created by the compiler.

(424) record too long ***(Objtohex)***

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

(425) incomplete record ***(Objtohex, Libr)***

The object file passed to OBJTOHEX or the librarian is corrupted. Contact HI-TECH Support with details.

(426) can't open checksum file * ***(Linker)***

The checksum file specified to OBJTOHEX could not be opened. Confirm the spelling and path of the file specified on the command line.

(427) syntax error in checksum list ***(Objtohex)***

There is a syntax error in a checksum list read by OBJTOHEX. The checksum list is read from standard input in response to an option.

(428) too many segment fixups ***(Objtohex)***

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

(429) bad segment fixups ***(Objtohex)***

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

(430) bad checksum specification ***(Objtohex)***

A checksum list supplied to `OBJTOHEX` is syntatically incorrect.

(433) out of memory allocating * blocks of * ***(Linker)***

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

(434) 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.

(435) bad segspec * ***(Linker)***

The segment specification option (`-G`) to the linker is invalid, e.g.:

```
-GA/f0+10
```

Did you forget the radix?

```
-GA/f0h+10
```

(436) psect "*" re-orged *(Linker)*

This psect has had its start address specified more than once.

(437) missing "=" in class spec *(Linker)*

A class spec needs an = sign, e.g. -Ctext=ROM See Section 5.7.9 for more infomation.

(438) bad size in -S option *(Linker)*

The address given in a `-S` 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. A leading `0x` may also be used for hexadecimal. Case in not important for any number or radix. Decimal is the default, e.g.:

```
-SCODE=f000
```

Did you forget the radix?

```
-SCODE=f000h
```

(441) bad -A spec: "*" *(Linker)*

The format of a `-A` specification, giving address ranges to the linker, is invalid, e.g.:

```
-ACODE
```

What is the range for this class? Maybe you meant:

```
-ACODE=0h-1fffh
```

(443) 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. A leading `0x` may also be used for hexadecimal. Case in not important for any number or radix. Decimal is default, e.g.:

```
-ACODE=1fff-3fffh
```

Did you forget the radix?

```
-ACODE=1fffh-3fffh
```

(444) expected "-" in -A spec *(Linker)*

There should be a minus sign, `-`, between the high and low addresses in a `-A` linker option, e.g.

```
-AROM=1000h
```

maybe you meant:

```
-AROM=1000h-1fffh
```

(445) 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. A leading `0x` may also be used for hexadecimal. Case in not important for any number or radix. Decimal is the default, e.g.:

```
-ACODE=0h-ffff
```

Did you forget the radix?

```
-ACODE=0h-ffffh
```

See Section 5.7.20 for more infomation.

(446) 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. A leading `0x` may also be used for hexadecimal. Case in not important for any number or radix. Decimal is default, e.g.:

```
-AENTRY=0-0FFh-1FF
```

Did you forget the radix?

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

(447) 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. A leading `0x` may also be used for hexadecimal. Case in not important for any number or radix. Decimal is default, e.g.:

```
-ACODE=0h-3fffh/a000
```

Did you forget the radix?

```
-ACODE=0h-3fffh/a000h
```

(448) bad repeat count in -A spec - * ***(Linker)***

The repeat count given in a `-A` specification is invalid, e.g.:

```
-AENTRY=0-0FFhxf
```

Did you forget the radix?

```
-AENTRY=0-0FFhxfh
```

(449) syntax error in -A spec: * ***(Linker)***

The `-A` spec is invalid. A valid -A spec should be something like:
-AROM=1000h-1FFFh

(450) unknown psect: * ***(Linker, Optimiser)***

This psect has been listed in a `-P` option, but is not defined in any module within the program.

(451) bad origin format in spec ***(Linker)***

The origin format in a `-p` option is not a validly formed decimal, octal or hex number, nor is it the name of an existing psect. A hex number must have a trailing H, e.g.:

```
-pbss=f000
```

Did you forget the radix?

```
-pbss=f000h
```

(452) bad min (+) format in spec ***(Linker)***

The minimum address specification in the linker's `-p` option is badly formatted, e.g.:

```
-pbss=data+f000
```

Did you forget the radix?

```
-pbss=data+f000h
```

(453) missing number after % in -p option ***(Linker)***

The `%` operator in a `-p` option (for rounding boundaries) must have a number after it.

(455) psect * not relocated on 0x* byte boundary ***(Linker)***

This psect is not relocated on the required boundary. Check the relocatability of the psect and correct the `-p` option. if necessary.

(458) cannot open ***(Objtohex)***

`OBJTOHEX` cannot open the specified input file. Confirm the spelling and path of the file specified on the command line.

(462) can't open avmap file * ***(Linker)***

A file required for producing Avocet format symbol files is missing. Confirm the spelling and path of the file specified on the command line.

(463) missing memory key in avmap file ***(Linker)***

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

(464) missing key in avmap file ***(Linker)***

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

(465) undefined symbol in FNBREAK record: * ***(Linker)***

The linker has found an undefined symbol in the `FNBREAK` record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(466) undefined symbol in FNINDIR record: * ***(Linker)***

The linker has found an undefined symbol in the `FNINDIR` record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(467) undefined symbol in FNADDR record: * ***(Linker)***

The linker has found an undefined symbol in the FNADDR record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(468) undefined symbol in FNCALL record: * ***(Linker)***

The linker has found an undefined symbol in the `FNCALL` record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(469) undefined symbol in FNROOT record: * ***(Linker)***

The linker has found an undefined symbol in the `FNROOT` record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(470) undefined symbol in FNSIZE record: * ***(Linker)***

The linker has found an undefined symbol in the `FNSIZE` record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(471) 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, e.g.:

```
int test(int a)
{
  if(a == 5)
    return test(a++);  /* recursion may not be supported by some compilers */
  return 0;
}
```

(472) function * appears in multiple call graphs: rooted at * and * *(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, e.g.:

```
void interrupt my_isr(void)
{
  scan(6);              /* scan is called from an interrupt function */
}
void process(int a)
{
  scan(a);              /* scan is also called from main-line code */
}
```

(474) no psect specified for function variable/argument allocation *(Linker)*

The `FNCONF` assembler directive which specifies to the linker information regarding the auto/parameter block was never seen. This is supplied in the standard runtime files if necessary. This error may imply that the correct run-time startoff module was not linked. Ensure you have used the `FNCONF` directive if the runtime startup module is hand-written.

(475) conflicting FNCONF records *(Linker)*

The linker has seen two conflicting `FNCONF` directives. This directive should only be specified once and is included in the standard runtime startup code which is normally linked into every program.

(476) fixup overflow referencing * * (loc 0x* (0x*+*), size *, value 0x*) *(Linker)*

The linker was asked to relocate (fixup) an item that would not fit back into the space after relocation. See the following error message (477) for more information..

(477) fixup overflow in expression (loc 0x* (0x*+*), size *, value 0x*) *(Linker)*

Fixup is the process conducted by the linker of replacing symbolic references to variables etc, in an assembler instruction with an absolute value. This takes place after positioning the psects (program sections or blocks) into the available memory on the target device. Fixup overflow is when the value determined for a symbol is too large to fit within the allocated space within the assembler instruction. For example, if an assembler instruction has an 8-bit field to hold an address and the linker determines that the symbol that has been used to represent this address has the value 0x110, then clearly this value cannot be inserted into the instruction.

The causes for this can be many, but hand-written assembler code is always the first suspect. Badly written C code can also generate assembler that ultimately generates fixup overflow errors. Consider the following error message.

```
main.obj: 8: Fixup overflow in expression (loc 0x1FD (0x1FC+1), size 1, value 0x7FC)
```

This indicates that the file causing the problem was `main.obj`. This would be typically be the output of compiling `main.c` or `main.as`. This tells you the file in which you should be looking. The next number (8 in this example) is the record number in the object file that was causing the problem. If you use the `DUMP` utility to examine the object file, you can identify the record, however you do not normally need to do this.

The location (`loc`) of the instruction (0x1FD), the `size` (in bytes) of the field in the instruction for the value (1) , and the `value` which is the actual value the symbol represents, is typically the only information needed to track down the cause of this error. Note that a size which is not a multiple of 8 bits will be rounded up to the nearest byte size, i.e. a 7 bit space in an instruction will be shown as 1 byte.

Generate an assembler list file for the appropriate module. Look for the address specified in the error message.

```
7    07FC    0E21  movlw 33
8    07FD     6FFC  movwf _foo
9    07FE    0012  return
```

and to confirm, look for the symbol referenced in the assembler instruction at this address in the symbol table at the bottom of the same file.

```
Symbol Table                    Fri Aug 12 13:17:37 2004
_foo 01FC   _main 07FF
```

In this example, the instruction causing the problem takes an 8-bit offset into a bank of memory, but clearly the address 0x1FC exceeds this size. Maybe the instruction should have been written as:

```
movwf  (_foo&0ffh)
```

which masks out the top bits of the address containing the bank information.

If the assembler instruction that caused this error was generated by the compiler, in the assembler list file look back up the file from the instruction at fault to determine which C statement has generated this instruction. You will then need to examine the C code for possible errors. incorrectly qualified pointers are an common trigger.

(479) 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.

(480) signatures do not match: * (*): 0x*/0x* *(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, e.g.:

```
extern int get_value(int in);
/* and in another module: */
int get_value(int in, char type)  /* this is different to the declaration */
{
```

(481) common symbol psect conflict: * *(Linker)*

A common symbol has been defined to be in more than one psect.

(482) symbol "*" multiply defined in file "*" *(Assembler)*

This symbol has been defined in more than one place. The assembler will issue this error if a symbol is defined more than once in the same module, e.g.:

```
_next:
  move r0, #55
  move [r1], r0
_next:           ; woops -- choose a different name
```

The linker will issue this warning if the symbol (C or assembler) was defined multiple times in different modules. The names of the modules are given in the error message. Note that C identifiers often have an *underscore* prepended to their name after compilation.

(483) symbol * cannot be global *(Linker)*

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

(484) psect * cannot be in classes * and * ***(Linker)***

A psect cannot be in more than one class. This is either due to assembler modules with conflicting `class=` options to the PSECT directive, or use of the `-C` option to the linker, e.g.:

```
psect final,class=CODE
finish:
/* elsewhere: */
psect final,class=ENTRY
```

(485) 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, e.g.:

```
psect starttext,class=CODE,with=rext ; was that meant to be with text?
```

(486) psect * selector value redefined ***(Linker)***

The selector associated with this psect has been defined differently in two or more places.

(486) psect * selector value redefined ***(Linker)***

The selector value for this psect has been defined more than once.

(487) 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.

(488) 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, e.g.:

```
psect spdata,class=RAM,space=0
  ds 6
; elsewhere:
psect spdata,class=RAM,space=1
```

(489) psect * memory delta redefined: */* *(Linker)*

A global psect has been defined with two different delta values, e.g.:

```
psect final,class=CODE,delta=2
finish:
; elsewhere:
psect final,class=CODE,delta=1
```

(490) 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.

(491) can't find * words for psect "*" in segment "*" *(Linker)*

One of the main tasks the linker performs is positioning the blocks (or psects) of code and data that is generated from the program into the memory available for the target device. This error indicates that the linker was unable to find an area of free memory large enough to accomodate one of the psects. The error message indicates the name of the psect that the linker was attempting to position and the segment name which is typically the name of a class which is defined with a linker `-A` option.

Section 3.9.1 lists each compiler-generated psect and what it contains. Typically psect names which are, or include, `text` relate to program code. Names such as `bss` or `data` refer to variable blocks. This error can be due to two reasons.

First, the size of the program or the progam's data has exceeded the total amount of space on the selected device. In other words, some part of your device's memory has completely filled. If this is the case, then the size of the specified psect must be reduced.

The second cause of this message is when the total amount of memory needed by the psect being positioned is sufficient, but that this memory is fragmented in such a way that the largest contiguous block is too small to accomodate the psect. The linker is unable to split psects in this situation. That is, the linker cannot place part of a psect at one location and part somewhere else. Thus, the linker must be able to find a contiguous block of memory large enough for every psect. If this is the cause of the error, then the psect must be split into smaller psects if possible.

To find out what memory is still available, generate and look in the map file, see Section 2.4.9 for information on how to generate a map file. Search for the string `UNUSED ADDRESS RANGES`. Under this heading, look for the name of the segment specified in the error message. If the name is not present, then all the memory available for this psect has been allocated. If it is present, there will be one address range specified under this segment for each free block of memory. Determine the size of each block and compare this with the number of words specified in the error message.

Psects containing code can be reduced by using all the compiler's optimizations, or restructring the program. If a code psect must be split into two or more small psects, this requies splitting a function into two or more smaller functions (which may call each other). These functions may need to be placed in new modules.

Psects containing data may be reduced when invoking the compiler optimizations, but the effect is less dramatic. The program may need to be rewritten so that it needs less variables. Section 5.10.2.2 has information on interpreting the map file's call graph if the compiler you are using uses a compiled stack. (If the string `Call grpah:` is not present in the map file, then the compiled code uses a hardware stack.) If a data psect needs to be split into smaller psects, the definitions for variables will need to be moved to new modules or more evenly spread in the existing modules. Memory allocation for `auto` variables is entirely handled by the compiler. Other than reducing the number of these variables used, the programmer has little control over their operation. This applies whether the compiled code uses a hardware or compiled stack.

For example, after receiving the message:

```
Can't find 0x34 words (0x34 withtotal) for psect text in segment CODE (error)
```

look in the map file for the ranges of unused memory.

```
UNUSED ADDRESS RANGES
        CODE             00000244-0000025F
                         00001000-0000102f
        RAM              00300014-00301FFB
```

In the `CODE` segment, there is 0x1c (0x25f-0x244+1) bytes of space available in one block and 0x30 available in another block. Neither of these are large enough to accomodate the psect `text` which is 0x34 bytes long. Notice, however, that the total amout of memory available is larger than 0x34 bytes.

(492) psect is absolute: * ***(Linker)***

This psect is absolute and should not have an address specified in a `-P` option. Either remove the `abs` psect flag, or remove the `-P` linker option.

(493) psect origin multiply defined: * ***(Linker)***

The origin of this psect is defined more than once. There is most likely more than one `-p` linker option specifying this psect.

(494) bad -P format "*"/"*" *(Linker)*

The `-P` option given to the linker is malformed. This option specifies placement of a psect, e.g.:

```
-Ptext=10g0h
```

Maybe you meant:

```
-Ptext=10f0h
```

(497) psect exceeds max size: *: *h > *h *(Linker)*

The psect has more bytes in it than the maximum allowed as specified using the `size` psect flag.

(498) psect exceeds address limit: *: *h > *h *(Linker)*

The maximum address of the psect exceeds the limit placed on it using the `limit` psect flag. Either the psect needs to be linked at a different location or there is too much code/data in the psect.

(499) 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.

(500) undefined symbols: *(Linker)*

A list of symbols follows that were undefined at link time. These errors could be due to spelling error, or failure to link an appropriate module.

(501) entry point multiply defined *(Linker)*

There is more than one entry point defined in the object files given the linker. End entry point is specified after the `END` directive. The runtime startup code defines the entry point, e.g.:

```
powerup:
  goto start
  END  powerup  ; end of file and define entry point
; other files that use END should not define another entry point
```

(502) 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. Contact HI-TECH Support with details.

(503) 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.

(504) 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. Contact HI-TECH Support if the object file if you have not patched the linker.

(505) 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. Contact HI-TECH Support if the object file was generated by the compiler.

(506) record too long: *+* ***(Linker)***

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

(507) unexpected end of file ***(Linker)***

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

(508) relocation offset * out of range 0..*-*-1 ***(Linker)***

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

(509) 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. Contact HI-TECH Support with details if the object file was created by the compiler.

(510) complex relocation not supported for -r or -l options ***(Linker)***

The linker was given a `-R` or `-L` option with file that contain complex relocation.

(511) bad complex range check ***(Linker)***

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

(512) unknown complex operator 0x* *(Linker)*

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

(513) bad complex relocation *(Linker)*

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

(514) 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. Contact HI-TECH Support with details if the object file was created by the compiler.

(515) unknown symbol type * *(Linker)*

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

(516) text record has bad length: *-*-(*+1) < 0 *(Linker)*

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

(517) write error (out of disk space?) * *(Linker)*

A write error occurred on the named file. This probably means you have run out of disk space.

(519) can't seek in * *(Linker)*

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

(520) 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. Some assembler library routines are never called, although they are actually execute. In this case, the routines are linked in a special sequence so that program execution falls through from one routine to the next.

(521) call depth exceeded by * *(Linker)*

The call graph shows that functions are nested to a depth greater than specified.

(522) library * is badly ordered ***(Linker)***

This library is badly ordered. It will still link correctly, but it will link faster if better ordered.

(523) argument -W* ignored ***(Linker)***

The argument to the linker option `-w` is out of range. This option controls two features. For warning levels, the range is -9 to 9. For the map file width, the range is greater than or equal to 10.

(524) unable to open list file * ***(Linker)***

The named list file could not be opened. The linker would be trying to fixup the list file so that it will contain absolute addresses. Ensure that an assembler list file was generated during the compilation stage. Alternatively, remove the assembler list file generation option from the link step.

(525) too many address spaces - space * ignored ***(Linker)***

The limit to the number of address spaces (specified with the `PSECT` assembler directive) is currently 16.

(526) psect * not specified in -p option (first appears in *) ***(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.

(528) 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.

(593) can't find 0x* words (0x* withtotal) for psect * in segment * ***(Linker)***

See error (491) in Appendix B.

(596) segment *(*-*) overlaps segment *(*-*) ***(Linker)***

The named segments have overlapping code or data. Check the addresses being assigned by the `-P` linker option.

(597) can't open ***(Linker)***

An object file could not be opened. Confirm the spelling and path of the file specified on the command line.

(602) null format name ***(Cromwell)***

The `-I` or `-O` option to Cromwell must specify a file format.

(603) ambiguous format name "*" ***(Cromwell)***

The input or output format specified to Cromwell is ambiguous. These formats are specified with the `-i`*`key`* and `-o`*`key`* options respectively.

(604) unknown format name "*" ***(Cromwell)***

The output format specified to `CROMWELL` is unknown, e.g.:

```
cromwell -m -P16F877 main.hex main.sym -ocot
```

and output file type of `cot`, did you mean `cof`?

(605) 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.

(606) inconsistent symbol tables ***(Cromwell)***

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

(607) inconsistent line number tables ***(Cromwell)***

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

(609) missing processor spec after -P ***(Cromwell)***

The `-p` option to cromwell must specify a processor name.

(611) too many input files ***(Cromwell)***

To many input files have been specified to be converted by `CROMWELL`.

(612) too many output files ***(Cromwell)***

To many output file formats have been specified to `CROMWELL`.

(613) no output file format specified ***(Cromwell)***

The output format must be specified to CROMWELL.

(614) no input files specified ***(Cromwell)***

CROMWELL must have an input file to convert.

(619) I/O error reading symbol table

Cromwell could not read the symbol table. This could be because the file was truncated or there was some other problem reading the file. Contact HI-TECH Support with details.

(620) file name index out of range in line number record ***(Cromwell)***

The COD file has an invalid format in the specified record.

(625) too many files in COFF file ***(Cromwell)***

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

(626) string lookup failed in coff:get_string() ***(Cromwell)***

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

(634) error dumping * ***(Cromwell)***

Either the input file to `CROMWELL` is of an unsupported type or that file cannot be dumped to the screen.

(635) invalid hex file: *, line * ***(Cromwell)***

The specified HEX file contains an invalid line. Contact HI-TECH Support if the HEX file was generated by the compiler.

(636) checksum error in Intel hex file *, line * *(Cromwell, Hexmate)*

A checksum error was found at the specified line in the specified Intel hex file. The HEX file may be corrupt.

(674) too many references to * *(Cref)*

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

(675) can't open * for input *(Cref)*

CREF cannot open the specified input file. Confirm the spelling and path of the file specified on the command line.

(676) can't open * for output *(Cref)*

CREF cannot open the specified output file. Confirm the spelling and path of the file specified on the command line.

(679) unknown extraspecial: * *(Code Generator)*

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

(680) bad format for -P option *(Code Generator)*

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

(685) bad putwsize *(Code Generator)*

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

(686) bad switch size * *(Code Generator)*

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

(687) bad pushreg "*" *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details. See Section 5.7.2 for more infomation.

(688) bad popreg "*" ***(Code Generator)***

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

(689) unknown predicate * ***(Code Generator)***

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

(693) 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.

```
#pragma interrupt_level 2  /* woops -- only 0 or 1 */
void interrupt isr(void)
{
  /* isr code goes here */
}
```

(695) duplicate case label * ***(Code Generator)***

There are two case labels with the same value in this `switch` statement, e.g.:

```
switch(in) {
case '0':  /* if this is case '0'... */
  b++;
  break;
case '0':  /* then what is this case? */
  b--;
  break;
}
```

(696) 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.

(697) non-constant case label ***(Code Generator)***

A `case` label in this `switch` statement has a value which is not a constant.

(698) 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`, e.g.:

```
bit proc(int a)
{
  bit bb;        /* woops --  this should be: static bit bb; */
  bb = (a > 66);
  return bb;
}
```

(699) no case labels *(Code Generator)*

There are no `case` labels in this `switch` statement, e.g.:

```
switch(input) {
}               /* there is nothing to match the value of input */
```

(701) unreasonable matching depth *(Code Generator)*

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

(702) regused - bad arg to G *(Code Generator)*

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

(703) bad GN *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details. See Section 5.7.2 for more infomation.

(704) bad RET_MASK *(Code Generator)*

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

(705) bad which (*) after I *(Code Generator)*

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

(706) expand - bad which *(Code Generator)*

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

(707) bad SX *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.See Section 5.7.20 for more infomation.

(708) bad mod "+" for how = * *(Code Generator)*

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

(709) metaregister * can't be used directly *(Code Generator)*

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

(710) bad U usage *(Code Generator)*

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

(711) expand - bad how *(Code Generator)*

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

(712) can't generate code for this expression *(Code Generator)*

This error indicates that a C expression is too difficult for the code generator to actually compile. For successful code generation, the code generator must know how to compile an expression and there must be enough resources (e.g. registers or temporary memory locations) available. Simplifying the expression, e.g. using a temporary variable to hold an intermediate result, may get around this message. Contact HI-TECH Support with details of this message.

This error may also be issued if the code being compiled is in some way unusual. For example code which writes to a const-qualified object is illegal and will result in warning messages, but the code generator may unsuccessfully try to produce code to perform the write.

(714) bad intermediate code *(Code Generator)*

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

(715) bad pragma * *(Code Generator)*

The code generator has been passed a `pragma` directive that it does not understand. This implies that the pragma you have used is a HI-TECH specific pragma, but the specific compiler you are using has not implemented this pragma.

(716) bad -M option: -M* *(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.

(718) 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. Contact HI-TECH Support with details if required.

(720) multiple free: * *(Code Generator)*

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

(721) bad element count expr *(Code Generator)*

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

(722) bad variable syntax *(Code Generator)*

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

(723) functions nested too deep *(Code Generator)*

This error is unlikely to happen with C code, since C cannot have nested functions! Contact HI-TECH Support with details.

(724) bad op * to revlog *(Code Generator)*

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

(726) bad uconval - * ***(Code Generator)***

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

(727) bad bconfloat - * ***(Code Generator)***

This is an internal code generator error. Contact HI-TECH technical support with details.

(728) bad confloat - * ***(Code Generator)***

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

(729) bad conval - * ***(Code Generator)***

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

(730) bad op: "*" ***(Code Generator)***

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

(731) expression error with reserved word ***(Code Generator)***

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

(732) can't initialize bit type ***(Code Generator)***

Variables of type `bit` cannot be initialised, e.g.:

```
bit b1 = 1;  /* woops -- b1 must be assigned a value after its definition */
```

(733) bad string "*" in psect pragma ***(Code Generator)***

The code generator has been passed a `pragma psect` directive that has a badly formed string, e.g.:

```
#pragma psect text  /* redirect text psect into what? */
```

Maybe you meant something like:

```
#pragma psect text=special_text
```

(734) too many psect pragmas *(Code Generator)*

Too many `#pragma psect` directives have been used.

(739) error closing output file *(Code Generator, Optimiser)*

The compiler detected an error when closing a file. Contact HI-TECH Support with details.

(740) bad dimensions *(Code Generator)*

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

(741) 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`, e.g. assuming an `int` is 16 bits wide:

```
struct {
  unsigned flag  : 1;
  unsigned value : 12;
  unsigned cont  : 6;   /* woops -- that makes a total of 19 bits */
} object;
```

(742) 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 program structure should be modified to prevent this type of call sequence.

(744) static object has zero size: * ***(Code Generator)***

A static object has been declared, but has a size of zero.

(745) nodecount = * ***(Code Generator)***

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

(747) unrecognized option to -Z: * ***(Code Generator)***

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

(748) 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, e.g.:

```
void main(void)
{
  int a;
  if(a)         /* woops -- a has never been assigned a value */
    process();
}
```

(749) unknown register name * ***(Linker)***

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

(750) 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. This message may also occur if the global optimizer is enabled and one of the operands is an `auto` or `static` local variable whose value has been tracked by the code generator, e.g.:

```
{
int a;
a = 6;
```

```
if(a || b)  /* a is 6, therefore this is always true */
    b++;
```

(751) 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. The most common code to trigger this warning is assignments to `signed` data types. For example:

```
signed char c;
c = 0xFF;
```

As a `signed` 8-bit quantity, `c` can only be assigned values -128 to 127. The constant is equal to 255 and is outside this range. If you mean to set all bits in this variable, then use either of:

```
c = ~0x0;
c = -1;
```

which will set all the bits in the variable regardless of the size of the variable and without warning.

This warning can also be triggered by intermediate values overflowing. For example:

```
unsigned int i;  /* assume ints are 16 bits wide */
i = 240 * 137;   /* this should be okay, right? */
```

A quick check with your calculator reveils that 240 * 137 is 32880 which can easily be stored in an `unsigned int`, but a warning is produced. Why? Because 240 and 137 and both `signed int` values. Therefore the result of the multiplication must also be a `signed int` value, but a `signed int` cannot hold the value 32880. (Both operands are constant values so the code generator can evaluate this expression at compile time, but it must do so following all the ANSI rules.) The following code forces the multiplication to be performed with an `unsigned` result:

```
i = 240u * 137;  /* force at least one operand to be unsigned */
```

(752) conversion to shorter data type ***(Code Generator)***

Truncation may occur in this expression as the lvalue is of shorter type than the rvalue, e.g.:

```
char a;
int b, c;
a = b + c;  /* conversion of int to char may result in truncation */
```

(753) 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. 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, e.g.:

```
int input;
input <<= 33;  /* woops -- that shifts the entire value out of input */
```

(754) 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, e.g.:

```
struct {
  unsigned mask : 2;  /* mask can hold values 0 to 3 */
} value;
int compare(void)
{
  return (value.mask == 6);  /* test can
}
```

(755) division by zero ***(Code Generator)***

A constant expression that was being evaluated involved a division by zero, e.g.:

```
a /= 0;  /* divide by 0: was this what you were intending */
```

(757) constant conditional branch ***(Code Generator)***

A conditional branch (generated by an `if`, `for`, `while` statement etc.) always follows the same path. This will be some sort of comparison involving a variable and a constant expression. For the code generator to issue this message, the variable must have local scope (either `auto` or `static` local) and the global optimizer must be enabled, possibly at higher level than 1, and the warning level threshold may need to be lower than the default level of 0.

The global optimizer keeps track of the contents of local variables for as long as is possible during a function. For C code that compares these variables to constants, the result of the comparison can be deduced at compile time and the output code hard coded to avoid the comparison, e.g.:

```
{
  int a, b;
  a = 5;
  if(a == 4)  /* this can never be false; always perform the true statement */
    b = 6;
```

will produce code that sets `a` to 5, then immediately sets `b` to 6. No code will be produced for the comparison `if(a == 4)`. If a was a global variable, it may be that other functions (particularly interrupt functions) may modify it and so tracking the variable cannot be performed.

This warning may indicate more than an optimization made by the compiler. It may indicate an expression with missing or badly placed parentheses, causing the evaluation to yield a value different to what you expected.

This warning may also be issued because you have written something like `while(1)`. To produce an infinite loop, use `for(;;)`.

A similar situation arises with for loops, e.g.:

```
{
  int a, b;
  for(a=0; a!=10; a++)  /* this loop must iterate at least once */
    b = func(a);
```

In this case the code generator can again pick up that `a` is assigned the value 0, then immediately checked to see if it is equal to 10. Because `a` is modified during the `for` loop, the comparison code cannot be removed, but the code generator will adjust the code so that the comparison is not performed on the first pass of the loop; only on the subsequent passes. This may not reduce code size, but it will speed program execution.

(758) 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 ==, e.g.:

```
int a, b;
if(a = 4)  /* this can never be false; always perform the true statement */
  b = 6;
```

will assign the value 4 to a, then , as the value of the assignment is always true, the comparison can be omitted and the assignment to `b` always made. Did you mean:

```
if(a == 4)  /* this can never be false; always perform the true statement */
    b = 6;
```

which checks to see if a is equal to 4.

(759) expression generates no code ***(Code Generator)***

This expression generates no output code. Check for things like leaving off the parentheses in a function call, e.g.:

```
int fred;
fred;     /* this is valid, but has no effect at all */
```

Some devices require that special function register need to be read to clear hardware flags. To accommodate this, in some instances the code generator *does* produce code for a statement which only consists of a variable ID. This may happen for variables which are qualified as `volatile`. Typically the output code will read the variable, but not do anything with the value read.

(760) 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, e.g.:

```
int a, b, c;
a = b,c;  /* "b" has no effect, was that meant to be a comma? */
```

(761) 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.

(763) 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, e.g.:

```
a = 8 ? b : c;  /* this is the same as saying a = b; */
```

(764) mismatched comparison *(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.:

```
unsigned char c;
if(c > 300)      /* woops -- how can this be true? */
  close();
```

(765) degenerate unsigned comparison *(Code Generator)*

There is a comparison 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.

(766) degenerate signed comparison *(Code Generator)*

There is a comparison of a `signed` value with the most negative value possible for this type, such that the comparison 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.

(768) 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, e.g.:

```
unsigned int a;
if(a == -10)    /* if a is unsigned, how can it be -10? */
  b = 9;
```

(769) no space for macro definition *(Assembler)*

The assembler has run out of memory.

(770) insufficient memory for macro definition *(Assembler)*

There is not sufficient memory to store a macro definition.

(772) 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.

(773) 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.

(774) too many macro parameters *(Assembler)*

There are too many macro parameters on this macro definition.

(778) 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.

(780) too many psects *(Assembler)*

There are too many psects defined! Boy, what a program!

(781) can't enter abs psect *(Assembler)*

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

(782) REMSYM error *(Assembler)*

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

(783) "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. The `with` flag is an attribute of a psect and indicates that this psect must be placed in the same memory page as the specified psect.

Remove a `with` flag from one of the psect declarations. Such an assembler declaration may look like:

```
psect my_text,local,class=CODE,with=basecode
```

which will define a psect called `my_text` and place this in the same page as the psect `basecode`.

(784) overfreed *(Assembler)*

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

(785) too many temporary labels *(Assembler)*

There are too many temporary labels in this assembler file. The assembler allows a maximum of 2000 temporary labels.

(787) copyexpr: can't handle v_rtype = * *(Assembler)*

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

(788) invalid character ("*") in number *(Assembler)*

A number contained a character that was not part of the range 0-9 or 0-F.

(790) EOF inside conditional *(Assembler)*

END-of-FILE was encountered while scanning for an "endif" to match a previous "if".

(793) unterminated macro arg *(Assembler)*

An argument to a macro is not terminated. Note that angle brackets ("< >") are used to quote macro arguments.

(794) 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.

(796) 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.

(798) 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, e.g.:

```
mmm macro a1
  move  r0, #a1
  LOCAL a1       ; woops -- the macro parameter cannot be used with local
ENDM
```

(799) rept argument must be >= 0 *(Assembler)*

The argument to a REPT directive must be greater than zero, e.g.:

```
rept -2            ; -2 copies of this code? */
  move r0, [r1]++
endm
```

(800) undefined symbol * *(Assembler)*

The named symbol is not defined in this module, and has not been specified GLOBAL.

(801) range check too complex *(Assembler)*

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

(802) 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.

(803) undefined temporary label *(Assembler)*

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

(808) add_reloc - bad size *(Assembler)*

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

(809) unknown addressing mode * ***(Assembler, Optimiser)***

An unknown addressing mode was used in the assembly file.

(814) 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.

(815) syntax error in chipinfo file at line * ***(Assembler)***

The chipinfo file contains non-standard syntax at the specified line.

(817) unknown architecture in chipinfo file at line * ***(Assembler, Driver)***

An chip architecture (family) that is unknown was encountered when reading the chip INI file.

(826) inverted ram bank in chipinfo file at line * ***(Assembler, Driver)***

The second hex number specified in the RAM field in the chipinfo file must be greater in value than the first.

(828) inverted common bank in chipinfo file at line * ***(Assembler, Driver)***

The second hex number specified in the COMMON field in the chipinfo file must be greater in value than the first. Contact HI-TECH Support if you have not modified the chipinfo INI file.

(829) unrecognized line in chipinfo file at line * ***(Assembler)***

The chipinfo file contains a processor section with an unrecognised line. Contact HI-TECH Support if the INI has not been edited.

(832) empty chip info file * ***(Assembler)***

The chipinfo file contains no data. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(833) no valid entries in chipinfo file ***(Assembler)***

The chipinfo file contains no valid processor descriptions.

(834) page width must be >= 60 ***(Assembler)***

The listing page width must be at least 60 characters. Any less will not allow a properly formatted listing to be produced, e.g.:

```
LIST C=10  ; the page width will need to be wider than this
```

(835) form length must be >= 15 ***(Assembler)***

The form length specified using the `-Flength` option must be at least 15 lines. Setting this length to zero is allowed and turns off paging altogether. The default value is zero (pageless).

(836) no file arguments ***(Assembler)***

The assembler has been invoked without any file arguments. It cannot assemble anything.

(839) relocation too complex ***(Assembler)***

The complex relocation in this expression is too big to be inserted into the object file.

(840) 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.

(842) bad bit number ***(Assembler, Optimiser)***

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

(843) a macro name cannot also be an EQU/SET symbol ***(Assembler)***

An EQU or SET symbol has been found with the same name as a macro. This is not allowed. For example:

```
getval MACRO
  mov  r0, r1
ENDM
getval EQU 55h  ; woops -- choose a different name to the macro
```

(844) lexical error ***(Assembler, Optimiser)***

An unrecognized character or token has been seen in the input.

(845) multiply defined symbol * *(Assembler)*

This symbol has been defined in more than one place. The assembler will issue this error if a symbol is defined more than once in the same module, e.g.:

```
_next:
  move r0, #55
  move [r1], r0
_next:          ; woops -- choose a different name
```

The linker will issue this warning if the symbol (C or assembler) was defined multiple times in different modules. The names of the modules are given in the error message. Note that C identifiers often have an *underscore* prepended to their name after compilation.

(846) 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.

(847) operand error *(Assembler, Optimiser)*

The operand to this opcode is invalid. Check your assembler reference manual for the proper form of operands for this instruction.

(849) illegal instruction for this processor *(Assembler)*

The instruction is not supported by this processor.

(852) radix must be from 2 - 16 *(Assembler)*

The radix specified using the `RADIX` assembler directive must be in the range from 2 (binary) to 16 (hexadecimal).

(853) invalid size for FNSIZE directive *(Assembler)*

The assembler FNSIZE assembler directive arguments must be positive constants.

(855) 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, e.g.:

```
ORG -10  /* this must a positive offset to the current psect */
```

(857) psect may not be local and global *(Linker)*

A local psect may not have the same name as a global psect, e.g.:

```
psect text,class=CODE       ; text is implicitly global
  move r0, r1
; elsewhere:
psect text,local,class=CODE
  move r2, r4
```

The `global` flag is the default for a psect if its scope is not explicitly stated.

(859) 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 decimal constant number, e.g.:

```
LIST C=a0h  ; constant must be decimal and positive, try: LIST C=80
```

(861) 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, e.g.:

```
LIST N=-3  ; page length must be positive
```

(862) symbol is not external *(Assembler)*

A symbol has been declared as EXTRN but is also defined in the current module.

(863) 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.

(864) SIZE= must specify a positive constant *(Assembler)*

The parameter to the `PSECT` assembler directive's `size` option must be a positive constant number, e.g.:

```
PSECT text,class=CODE,size=-200  ; a negative size?
```

(865) psect size redefined *(Assembler)*

The `size` flag to the `PSECT` assembler directive is different from a previous `PSECT` directive, e.g.:

```
psect spdata,class=RAM,size=400
; elsewhere:
psect spdata,class=RAM,size=500
```

(866) RELOC= must specify a positive constant *(Assembler)*

The parameter to the `PSECT` assembler directive's `reloc` option must be a positive constant number, e.g.:

```
psect test,class=CODE,reloc=-4  ; the reloc must be positive
```

(867) psect reloc redefined *(Assembler)*

The `reloc` flag to the `PSECT` assembler directive is different from a previous `PSECT` directive, e.g.:

```
psect spdata,class=RAM,reloc=4
; elsewhere:
psect spdata,class=RAM,reloc=8
```

(868) DELTA= must specify a positive constant *(Assembler)*

The parameter to the `PSECT` assembler directive's `DELTA` option must be a positive constant number, e.g.:

```
PSECT text,class=CODE,delta=-2  ; a negative delta value does not make sense
```

(871) SPACE= must specify a positive constant *(Assembler)*

The parameter to the PSECT assembler directive's `space` option must be a positive constant number, e.g.:

```
PSECT text,class=CODE,space=-1  ; space values start at zero
```

(872) psect space redefined ***(Assembler)***

The `space` flag to the `PSECT` assembler directive is different from a previous `PSECT` directive, e.g.:

```
psect spdata,class=RAM,space=0
; elsewhere:
psect spdata,class=RAM,space=1
```

(873) 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's class is specified via a flag as in the following:

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

Look for other psect definitions that specify a different class name.

(874) a psect may only have one "with" option ***(Assembler)***

A psect can only be placed `with` one other psect. A psect's with option is specified via a flag as in the following:

```
psect bss,with=data
```

Look for other psect definitions that specify a different `with` psect name.

(875) bad character constant in expression ***(Assembler,Optimizer)***

The character constant was expected to consist of only one character, but was found to be greater than one character or none at all. An assembler specific example:

```
mov  r0, #'12'    ; '12' specifies two characters
```

(876) syntax error ***(Assembler, Optimiser)***

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

(915) no room for arguments ***(Preprocessor, Parser, Code Generator, Linker, Objtohex)***

The code generator could not allocate any more memory.

(916) can't allocate memory for arguments ***(Preprocessor, Parser, Code generator, Assembler)***

The compiler could not allocate any more memory when trying to read in command-line arguments.

(917) argument too long ***(Preprocessor, Parser)***

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

(918) *: no match ***(Preprocessor, Parser)***

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

(921) can't open chipinfo file * ***(Driver, Assembler)***

The chipinfo file could not be opened. This file normally resides in the LIB directory of the compiler distribution. If driving the assembler directly (without the command line driver) ensure that the option to location this file correctly specifies the path, otherwise contact HI-TECH Support with details.

(941) bad * assignment; USAGE: * ***(Hexmate)***

An option to Hexmate was incorrectly used or incomplete. Follow the usage supplied by the message and ensure that that the option has been formed correctly and completely.

(942) unexpected character on line * of file * ***(Hexmate)***

File contains a character that was not valid for this type of file, the file may be corrupt. For example, an Intel hex file is expected to contain only ASCII representations of hexadecimal digits, colons (:) and line formatting. The presence of any other characters will result in this error.

(944) data conflict at address *h between * and * ***(Hexmate)***

Sources to Hexmate request differing data to be stored to the same address. To force one data source to override the other, use the '+' specifier. If the two named sources of conflict are the same source, then the source may contain an error.

(945) checksum range (*h to *h) contained an indeterminate value ***(Hexmate)***

The range for this checksum calculation contained a value that could not be resolved. This can happen if the checksum result was to be stored within the address range of the checksum calculation.

(948) checksum result width must be between 1 and 4 bytes ***(Hexmate)***

The requested checksum byte size is illegal. Checksum results must be within 1 to 4 bytes wide. Check the parameters to the -CKSUM option.

(949) start of checksum range must be less than end of range ***(Hexmate)***

The -CKSUM option has been given a range where the start is greater than the end. The parameters may be incomplete or entered in the wrong order.

(951) start of fill range must be less than end of range ***(Hexmate)***

The -FILL option has been given a range where the start is greater than the end. The parameters may be incomplete or entered in the wrong order.

(953) unknown -HELP sub-option: * ***(Hexmate)***

Invalid sub-option passed to -HELP. Check the spelling of the sub-option or use -HELP with no sub-option to list all options.

(954) incomplete -O option; no file specified ***(Hexmate)***

The output filename option did not contain a filename. A filename must follow -O. Make sure the filename and -O are not separated by a space.

(956) -SERIAL value must be between 1 and * bytes long ***(Hexmate)***

The serial number being stored was out of range. Ensure that the serial number can be stored in the number of bytes permissible by this option.

(958) too many input files specified; * file maximum ***(Hexmate)***

Too many file arguments have been used. Try merging these files in several stages rather than in one command.

(960) unexpected record type(*) on line * of "*" ***(Hexmate)***

Intel hex file contained an invalid record type. Consult the Intel hex format specification for valid record types.

(962) forced data conflict at address *h between * and * *(Hexmate)*

Sources to Hexmate force differing data to be stored to the same address. More than one source using the '+' specifier store data at the same address. The actual data stored there may not be what you expect.

(963) checksum range includes voids or unspecified memory locations *(Hexmate)*

Checksum range had gaps in data content. The runtime calculated checksum is likely to differ from the compile-time checksum due to gaps/unused byes within the address range that the checksum is calculated over. Filling unused locations with a known value will correct this.

(966) no END record for HEX file "*" *(Hexmate)*

Intel hex file did not contain a record of type END. The hex file may be incomplete.

(967) unused function definition: * (from line *) *(Parser)*

The indicated `static` function was never called in the module being compiled. Being static, the function cannot be called from other modules so this warning imples the function is never used. Either the function is redundant, or the code that was meant to call it was excluded from compilation or misspelt the name of the function.

(968) unterminated string *(Assembler, Optimiser)*

A string constant appears not to have a closing quote missing.

(969) end of string in format specifier *(Parser)*

The format specifier for the printf() style function is malformed.

(970) character not valid at this point in format specifier *(Parser)*

The printf() style format specifier has an illegal character.

(971) type modifiers not valid with this format *(Parser)*

Type modifiers may not be used with this format.

(972) only modifiers h and l valid with this format ***(Parser)***

Only modifiers `h` (`short`) and `l` (`long`) are legal with this `printf` format specifier.

(973) only modifier l valid with this format ***(Parser)***

The only modifier that is legal with this format is `l` (for long).

(974) type modifier already specified ***(Parser)***

This type modifier has already be specified in this type.

(975) invalid format specifier or type modifier ***(Parser)***

The format specifier or modifier in the printf-style string is illegal for this particular format.

(976) field width not valid at this point ***(Parser)***

A field width may not appear at this point in a printf() type format specifier.

(978) 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`, e.g.:

```
enum IN {ONE=1, TWO};
struct IN {             /* woops -- IN is already defined */
  int a, b;
};
```

(979) 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`, e.g.:

```
struct IN {
  int a, b;
};
enum IN {ONE=1, TWO};  /* woops -- IN is already defined */
```

(980) 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`, e.g.:

```
union IN {
  int a, b;
};
enum IN {ONE=1, TWO};  /* woops -- IN is already defined */
```

(981) pointer required *(Parser)*

A pointer is required here, e.g.:

```
struct DATA data;
data->a = 9;      /* data is a structure, not a pointer to a structure */
```

(982) nxtuse(): unknown op: * *(Optimiser,Assembler)*

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

(984) 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, e.g.:

```
int a;
char a;  /* woops -- what is the correct type? */
```

(985) qualifiers redeclared *(Parser)*

This function has different qualifiers in different declarations.

(988) number of arguments redeclared *(Parser)*

The number of arguments in this function declaration does not agree with a previous declaration of the same function.

(989) module has code below file base of *h ***(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.

(990) modulus 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, e.g.:

```
#define ZERO 0
#if FOO%ZERO    /* this will have an assumed result of 0 */
  #define INTERESTING
#endif
```

(991) 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`, e.g.:

```
enum { one = 1, two, about_three = 3.12 };  /* no non-int values allowed */
```

(992) can't find op ***(Assembler, Optimiser)***

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

(1198) too many "*" specifications; * maximum ***(Hexmate)***

This option has been specified too many times. If possible, try performing these operations over several command lines.

(1201) all FIND/REPLACE code specifications must be of equal width ***(Hexmate)***

All find, replace and mask attributes in this option must be of the same byte width. Check the parameters supplied to this option. For example finding 1234h (2 bytes) masked with FFh (1 byte) will result in an error, but masking with 00FFh (2 bytes) will be Ok.

(1202) unknown format requested in -FORMAT: * ***(Hexmate)***

An unknown or unsupported INHX format has been requested. Refer to documentation for supported INHX formats.

(1203) unpaired nibble in * value will be truncated ***(Hexmate)***

Data to this option was not entered as whole bytes. Perhaps the data was incomplete or a leading zero was omitted. For example the value Fh contains only four bits of significant data and is not a whole byte. The value 0Fh contains eight bits of significant data and is a whole byte.

(1204) * value must be between 1 and * bytes long ***(Hexmate)***

An illegal legth of data was given to this option. The value provided to this option exceeds the maximum or minimum bounds required by this option.

(1212) Found * (*h) at address *h ***(Hexmate)***

The code sequence specified in a -FIND option has been found at this address.

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.

a maximum of * reserved areas are allowed. remainder of -RES* ignored ***(Driver)***

Too many address ranges were specified with either the `-RESROM` or `-RESRAM` option.

can't create cross reference file * ***(Assembler)***

The assembler attempted to create a cross reference file, but it could not be created. Check that the file's pathname is correct.

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 valid and that has read only access.

duplicate arch for * in chipinfo file at line * ***(Assembler, Driver)***

The chipinfo file has a processor section with multiple ARCH values. Only one ARCH value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

duplicate lib for * in chipinfo file at line * ***(Assembler)***

The chipinfo file has a processor section with multiple `LIB` values. Only one `LIB` value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

duplicate romsize for * in chipinfo file at line * ***(Assembler)***

The chipinfo file has a processor section with multiple ROMSIZE values. Only one ROMSIZE value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

duplicate sparebit for * in chipinfo file at line * ***(Assembler)***

The chipinfo file has a processor section with multiple SPAREBIT values. Only one SPAREBIT value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

duplicate * for * in chipinfo file at line * ***(Assembler, Driver)***

The chipinfo file has a processor section with multiple values for a field. Only one value is allowed per chip. If you have not manually edited the chip info file, contact HI-TECH Support with details.

duplicate zeroreg for * in chipinfo file at line * ***(Assembler)***

The chipinfo file has a processor section with multiple ZEROREG values. Only one ZEROREG value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

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. If you have not manually edited the chip info file, contact HI-TECH Support with details.

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 must be greater in value than the first. Contact HI-TECH Support if you have not modified the chipinfo INI file.

missing arch specification for * in chipinfo file ***(Assembler)***

The chipinfo file has a processor section without an ARCH values. The architecture of the processor must be specified. Contact HI-TECH Support if the chipinfo file has not been modified.

psect * not loaded on 0x* 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.

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, e.g.:

```
-A
```

What is the offset? Maybe you meant:

```
-A200See Section 5.7.2 for more details regarding this option.
```

bad -RES* arguments ***(Driver)***

The address ranges specified to either the `-RESROM` or `-RESRAM` option are invalid.

bad -ROM arguments * ***(Driver)***

The arguments to `-ROM` were either not present or badly formed.

banked/common conflict ***(Assembler)***

The assembler has found conflicting information that suggests that a symbol is located in the access bank, but also in the banked RAM area, e.g.:

```
movwf  c:_foo,b  ; _foo cannot be common and banked
```

bit range check failed * ***(Linker)***

The assembler can place checks associated with an instruction in the output object file that will confirm that the value ultimately assigned to a symbol used within the instruction is within some range. This error indicates that the range check failed, i.e. the value was either too large or too small. This error relates to checks carried on a bit addresses. If there is no hand-written assembler code in this program, then this may be an internal compiler error and you should contact HI-TECH support with details of the code that generated this error. Other causes are numerous.

can't open include file * ***(Assembler)***

The named assembler include file could not be opened. Confirm the spelling and path of the file specified in the INCLUDE directive, e.g.:

```
INCLUDE "misspilt.h"  ; is the filename correct?
```

delete what ? ***(Libr)***

The librarian requires one or more modules to be listed for deletion when using the `d` key, e.g.:

```
libr d c:\ht-pic\lib\pic704-c.lib
```

does not indicate which modules to delete. try something like:

```
libr d c:\ht-pic\lib\pic704-c.lib wdiv.obj
```

direct range check failed * ***(Linker)***

The assembler can place checks associated with an instruction in the output object file that will confirm that the value ultimately assigned to a symbol used within the instruction is within some range. This error indicates that the range check failed, i.e. the value was either too large or too small. If there is no hand-written assembler code in this program, then this may be an internal compiler error and you should contact HI-TECH support with details of the code that generated this error. Other causes are numerous.

duplicate banks for * in chipinfo file at line * ***(Assembler)***

The chipinfo file has a processor section with multiple BANKS values. Only one BANKS value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

file locking not enabled on network drive ***(Driver)***

The driver has attempted to modify the lock file located in the LIB directory but was unable to do so. This has probably resulted from the network drive used to hold the compiler being read only.

identifier expected ***(Parser)***

Inside the braces of an `enum` declaration should be a comma-separated list of identifiers, e.g.:

```
enum { 1, 2};  /* woops -- maybe you mean enum { one = 1, two }; */
```

incomplete ident record ***(Libr)***

The IDENT record in the object file was incomplete. Contact HI-TECH Support with details.

incomplete symbol record ***(Libr)***

The SYM record in the object file was incomplete. Contact HI-TECH Support with details.

library file names should have .lib extension: * ***(Libr)***

Use the `.lib` extension when specifying a library filename.

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.

module * defines no symbols ***(Libr)***

No symbols were found in the module's object file. This may be what was intended, or it may mean that part of the code was inadvertently removed or commented.

no addresses specified with -RES* option ***(Driver)***

No address ranges were specified to either the `-RESROM` or `-RESRAM` option.

no addresses specified with -ROM option ***(Driver)***

No addresses ranges were specified with the `-ROM` option.

no reserved * areas defined ***(Parser)***

No address ranges were specified with the `-RESRAM` or `-RESOM` option.

no ROM banks defined ***(Driver)***

The `-ROM` options was invoked but no valid bank address ranges were present.

no ROM range covering address 0 encountered ***(Driver)***

None of the on-chip memory or memory specified with -ROM was found to include address 0. This may have been deliberate.

psect limit redefined ***(Assembler)***

The psect limit has already been defined using the psect `limit` flag elsewhere, e.g.:

```
psect text,class=CODE,limit=1ffh
  move r0, r1
; elsewhere:
psect text,class=CODE,limit=2ffh
  move r2, r4
```

RAM area * low bound greater than high bound ***(Driver)***

An additional memory bank has been defined which has a lower address bound greater than the high address bound.

replace what ? ***(Libr)***

The librarian requires one or more modules to be listed for replacement when using the `r` key, e.g.:

```
libr r lcd.lib
```

This command needs the name of a module (`.obj` file) after the library name.

reserved * area and reserved ICD * range overlap in region * ***(Driver)***

The -ICD option has been used which reserves memory locations for the debugger. Additional memory areas have been reserved with the `-RESROM` or `-RESRAM` option and these address ranges overlap those required by the ICD.

reserved * area * - * and * - * could be merged ***(Driver)***

Two address ranges are contiguous. These could have been merged into one reserved range.

reserved * area * - * low bound greater than high bound ***(Driver)***

This structure member is never used. Maybe it isn't needed at all.

reserved * area * - * overlaps reserved * area * ***(Driver)***

Two address ranges were specified with either then -RESROM or -RESRAM option that overlap.

ROM bank * low bound greater than high bound ***(Driver)***

An additional memory bank has been defined which has a lower address bound greater than the high address bound.

ROM bank * overlaps ROM bank * ***(Driver)***

The `-ROM` options was invoked but no valid bank address ranges were present.

ROM banks * and * could be merged ***(Driver)***

Two banks specified are contiguous. This may have been intentional to create a memory boundary, otherwise these two banks could be merged into one larger bank.

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.

zero size ROM bank * defined ***(Driver)***

An additional memory bank has been defined which has a size of zero.

Appendix C

Chip Information

The following table lists all devices currently supported by HI-TECH PICC-18 STD.

Table C.1: Devices supported by HI-TECH PICC-18 STD

DEVICE	ROMSIZE	RAMSIZE	EEPROMSIZE	EXTMEM
18C242	4000	200		
18C252	8000	600		
18C442	4000	200		
18C452	8000	600		
18C601	40000	600		3FFFF
18C801	200000	600		1FFFFF
18C658	8000	600		
18C858	8000	600		
18F23K20	2000	200	100	
18F2410	4000	300		
18F242	4000	300	100	
18F2420	4000	300	100	
18F24K20	4000	300	100	
18F2423	4000	300	100	
18F248	4000	300	100	
18F2480	4000	300	100	
18F2510	8000	600		
18F2515	C000	F80		
18F252	8000	600	100	
18F2520	8000	600	100	
18F25K20	8000	600	100	
18F2523	8000	600	100	
18F258	8000	600	100	
18F2580	8000	600	100	

continued...

Table C.1: Devices supported by HI-TECH PICC-18 STD

DEVICE	ROMSIZE	RAMSIZE	EEPROMSIZE	EXTMEM
18F2610	10000	F80		
18F24J10	3FF8	400		
18LF24J10	3FF8	400		
18F24J11	3FF8	E60		
18LF24J11	3FF8	E60		
18F24J50	3FF8	E60		
18LF24J50	3FF8	E60		
18F25J10	7FF8	400		
18LF25J10	7FF8	400		
18F25J11	7FF8	E60		
18LF25J11	7FF8	E60		
18LF25J50	7FF8	E60		
18F25J50	7FF8	E60		
18F26J11	FFF8	F80		
18LF26J11	FFF8	F80		
18F26J50	FFF8	E60		
18LF26J50	FFF8	E60		
18F44J10	3FF8	400		
18LF44J10	3FF8	400		
18F44J11	3FF8	E60		
18LF44J11	3FF8	E60		
18F44J50	3FF8	E60		
18LF44J50	3FF8	E60		
18F45J10	7FF8	400		
18LF45J10	7FF8	400		
18F45J11	7FF8	E60		
18LF45J11	7FF8	E60		
18F45J50	7FF8	E60		
18LF45J50	7FF8	E60		
18F46J11	FFF8	F80		
18LF46J11	FFF8	F80		
18F46J50	FFF8	E60		
18LF46J50	FFF8	E60		
18F26K20	10000	F60	400	
18F43K20	2000	200	100	
18F4410	4000	300		
18F442	4000	300	100	
18F4420	4000	300	100	
18F44K20	4000	300	100	
18F4423	4000	300	100	
18F448	4000	300	100	
18F4480	4000	300	100	
18F4510	8000	600		
18F4515	C000	F80		
18F452	8000	600	100	

continued...

Table C.1: Devices supported by HI-TECH PICC-18 STD

DEVICE	ROMSIZE	RAMSIZE	EEPROMSIZE	EXTMEM
18F4520	8000	600	100	
18F45K20	8000	600	100	
18F4523	8000	600	100	
18F458	8000	600	100	
18F4580	8000	600	100	
18F4610	10000	F80		
18F46K20	10000	F60	400	
18F1220	1000	100	100	
18F1230	1000	100	80	
18F1320	2000	100	100	
18F1330	2000	100	80	
18F13K20	2000	300	100	
18F13K22	2000	A0	100	
18LF13K22	2000	A0	100	
18F14K20	4000	300	100	
18F14K22	4000	1A0	100	
18LF14K22	4000	1A0	100	
18F13K50	2000	200	100	
18LF13K50	2000	200	100	
18F14K50	4000	200	100	
18LF14K50	4000	200	100	
18F2220	1000	200	100	
18F2221	1000	200	100	
18F2320	2000	200	100	
18F2321	2000	200	100	
18F2331	2000	300	100	
18F2431	4000	300	100	
18F2439	3000	280	100	
18F2450	4000	200		
18F2455	6000	400	100	
18F2458	6000	400	100	
18F2525	C000	F80	400	
18F2550	8000	400	100	
18F2553	8000	400	100	
18F2539	6000	580	100	
18F2585	C000	D00	400	
18F2620	10000	F80	400	
18F2680	10000	D00	400	
18F2682	14000	D00	400	
18F2685	18000	D00	400	
18F4220	1000	200	100	
18F4221	1000	200	100	
18F4320	2000	200	100	
18F4321	2000	200	100	
18F4331	2000	300	100	

continued...

Table C.1: Devices supported by HI-TECH PICC-18 STD

DEVICE	ROMSIZE	RAMSIZE	EEPROMSIZE	EXTMEM
18F4431	4000	300	100	
18F4439	3000	280	100	
18F4455	6000	400	100	
18F4458	6000	400	100	
18F4525	C000	F80	400	
18F4539	6000	580	100	
18F4450	4000	200		
18F4550	8000	400	100	
18F4553	8000	400	100	
18F4585	C000	D00	400	
18F4620	10000	F80	400	
18F4680	10000	D00	400	
18F4682	14000	D00	400	
18F4685	18000	D00	400	
18F6310	2000	300		
18F63J11	1FF8	400		
18F6390	2000	300		
18F63J90	1FF8	400		
18F6393	2000	300		
18F6410	4000	300		
18F64J11	3FF8	400		
18F6490	4000	300		
18F64J90	3FF8	400		
18F6493	4000	300		
18F6520	8000	800	400	
18F6525	C000	F00	400	
18F6527	C000	F60	400	
18F6585	C000	D00	400	
18F65J10	7FF8	800		
18F65J11	7FF8	800		
18F65J15	BFF8	800		
18F65J50	7FF8	F40		
18F65J90	7FF8	800		
18F66J11	FFF8	F40		
18F66J16	17FF8	F40		
18F6620	10000	F00	400	
18F6621	10000	F00	400	
18F6622	10000	F60	400	
18F6627	18000	F60	400	
18F6628	18000	F60	400	
18F6680	10000	D00	400	
18F66J10	FFF8	800		
18F66J15	17FF8	F00		
18F66J50	FFF8	F40		
18F66J55	17FF8	F40		

continued...

Table C.1: Devices supported by HI-TECH PICC-18 STD

DEVICE	ROMSIZE	RAMSIZE	EEPROMSIZE	EXTMEM
18F67J11	1FFF8	F40		
18F6720	20000	F00	400	
18F6722	20000	F60	400	
18F6723	20000	F60	400	
18F67J10	1FFF8	F00		
18F67J50	1FFF8	F40		
18F8310	2000	300		1FFFFF
18F83J11	1FF8	400		
18F8390	2000	300		
18F83J90	1FF8	400		
18F8393	2000	300		
18F8410	4000	300		1FFFFF
18F84J11	3FF8	400		
18F8490	4000	300		
18F84J90	3FF8	400		
18F8493	4000	300		
18F85J11	7FF8	800		
18F8520	8000	800	400	1FFFFF
18F8525	C000	F00	400	1FFFFF
18F8527	C000	F60	400	1FFFFF
18F8585	C000	D00	400	1FFFFF
18F85J10	7FF8	800		1FFFFF
18F85J15	BFF8	800		1FFFFF
18F85J50	7FF8	F40		
18F85J90	7FF8	800		
18F86J11	FFF8	F40		
18F86J16	17FF8	F40		
18F8620	10000	F00	400	1FFFFF
18F8621	10000	F00	400	1FFFFF
18F8622	10000	F60	400	1FFFFF
18F8627	18000	F60	400	1FFFFF
18F8628	18000	F60	400	1FFFFF
18F8680	10000	D00	400	1FFFFF
18F86J10	FFF8	800		1FFFFF
18F86J15	17FF8	F00		1FFFFF
18F86J50	FFF8	F40		
18F86J55	17FF8	F40		
18F87J11	1FFF8	F40		
18F8720	20000	F00	400	1FFFFF
18F8722	20000	F60	400	1FFFFF
18F8723	20000	F60	400	1FFFFF
18F87J10	1FFF8	F00		1FFFFF
18F66J60	FFF8	E80		1FFFFF
18F66J65	17FF8	E80		1FFFFF
18F67J60	1FFF8	E80		1FFFFF

continued...

Table C.1: Devices supported by HI-TECH PICC-18 STD

DEVICE	ROMSIZE	RAMSIZE	EEPROMSIZE	EXTMEM
18F86J60	FFF8	E80		1FFFFF
18F86J65	17FF8	E80		1FFFFF
18F66J90	FFF8	EF4		
18F67J90	1FFF8	EF4		
18F86J90	FFF8	EF4		
18F87J90	1FFF8	EF4		
18F66J93	FFF8	EF4		
18F67J93	1FFF8	EF4		
18F86J93	FFF8	EF4		
18F87J93	1FFF8	EF4		
18F87J50	1FFF8	F40		
18F87J60	1FFF8	E80		1FFFFF
18F87K22	20000	EB6		
18F87K90	20000	E94	400	
18F96J60	FFF8	E80		1FFFFF
18F96J65	17FF8	E80		1FFFFF
18F97J60	1FFF8	E80		1FFFFF
18F23K22	2000	1A0	100	1FFFFF
18LF23K22	2000	1A0	100	1FFFFF
18F24K22	4000	2A0	100	1FFFFF
18LF24K22	4000	2A0	100	1FFFFF
18F25K22	8000	5A0	100	1FFFFF
18LF25K22	8000	5A0	100	1FFFFF
18F26K22	10000	9A0	400	1FFFFF
18LF26K22	10000	9A0	400	1FFFFF
18F43K22	2000	1A0	100	1FFFFF
18LF43K22	2000	1A0	100	1FFFFF
18F44K22	4000	2A0	100	1FFFFF
18LF44K22	4000	2A0	100	1FFFFF
18F45K22	8000	5A0	100	1FFFFF
18LF45K22	8000	5A0	100	1FFFFF
18F46K22	10000	9A0	400	1FFFFF
18LF46K22	10000	9A0	400	1FFFFF
18F25K80	8000	DE1	400	1FFFFF
18LF25K80	8000	DE1	400	1FFFFF
18F26K80	10000	DE1	400	1FFFFF
18LF26K80	10000	DE1	400	1FFFFF
18F45K80	8000	DE1	400	1FFFFF
18LF45K80	8000	DE1	400	1FFFFF
18F46K80	10000	DE1	400	1FFFFF
18LF46K80	10000	DE1	400	1FFFFF
18F65K80	8000	DE1	400	1FFFFF
18LF65K80	8000	DE1	400	1FFFFF
18F66K80	10000	DE1	400	1FFFFF
18LF66K80	10000	DE1	400	1FFFFF

continued...

Table C.1: Devices supported by HI-TECH PICC-18 STD

DEVICE	ROMSIZE	RAMSIZE	EEPROMSIZE	EXTMEM
18F86J72	FFF8	EF4		1FFFFF
18F87J72	1FFF8	EF4		1FFFFF
18F26J13	FFF8	E50		1FFFFF
18LF26J13	FFF8	E50		1FFFFF
18F27J13	1FFF8	E50		1FFFFF
18LF27J13	1FFF8	E50		1FFFFF
18F46J13	FFF8	E50		1FFFFF
18LF46J13	FFF8	E50		1FFFFF
18F47J13	1FFF8	E50		1FFFFF
18LF47J13	1FFF8	E50		1FFFFF
18F65K90	8000	7A0	400	1FFFFF
18F66K90	10000	E94	400	1FFFFF
18F67K90	20000	E94	400	1FFFFF
18F85K90	8000	7A0	400	1FFFFF
18F86K90	10000	E94	400	1FFFFF
18F26J53	FFF8	E50		1FFFFF
18LF26J53	FFF8	E50		1FFFFF
18F27J53	1FFF8	E50		1FFFFF
18LF27J53	1FFF8	E50		1FFFFF
18F46J53	FFF8	E50		1FFFFF
18LF46J53	FFF8	E50		1FFFFF
18F47J53	1FFF8	E50		1FFFFF
18LF47J53	1FFF8	E50		1FFFFF
18F65K22	8000	7A0	400	1FFFFF
18F66K22	10000	EB6	400	1FFFFF
18F67K22	20000	EB6	400	1FFFFF
18F85K22	8000	7A0	400	1FFFFF
18F86K22	10000	EB6	400	1FFFFF

Index

PICC18 Command-line Options

Option	**Meaning**
-C	Compile to object files only
-D*macro*	Define preprocessor macro
-E+*file*	Redirect and optionally append errors to a file
-G*file*	Generate source-level debugging information
-I*path*	Specify a directory pathname for include files
-L*library*	Specify a library to be scanned by the linker
-L-*option*	Specify *-option* to be passed directly to the linker
-M*file*	Request generation of a MAP file
-N*size*	Specify identifier length
-O*file*	Output file name
-P	Preprocess assembler files
-Q	Specify quiet mode
-S	Compile to assembler source files only
-U*symbol*	Undefine a predefined preprocessor symbol
-V	Verbose: display compiler pass command lines
-X	Eliminate local symbols from symbol table
--ASMLIST	Generate assembler .LST file for each compilation
--CALLGRAPH=*type*	Control the level of information displayed in the call graph
--CHAR=*type*	Make the default char signed or unsigned
--CHIP=*processor*	Selects which processor to compile for
--CHIPINFO	Displays a list of supported processors
--CODEOFFSET=*address*	Offset program code to address
--CP=*size*	Set size of pointers to code space
--CR=*file*	Generate cross-reference listing
--DEBUGGER=*type*	Select the debugger that will be used
--DOUBLE=*type*	Selects size/kind of double types
--EMI=*type*	Select the type of external memory interface used
--ERRATA=*type*	Add or remove specific software workarounds for silicon errata issues.
--ERRFORMAT<=*format*>	Format error message strings to the given style
--ERRORS=*number*	Sets the maximun number of errors displayed
--FILL=*opcode*	Specify a hexadecimal opcode to program in all unused program memory locations.
continued...	

PICC18 Command-line Options

Option	**Meaning**
--GETOPTION=*app,file*	Get the command line options for the named application
--HELP<=option>	Display the compiler's command line options
--IDE=*ide*	Configure the compiler for use by the named IDE
--LANG=*language*	Specify language for compiler messages
--MAPFILE<=*file*>	Generates a map file
--MEMMAP=*file*	Display memory summary information for the map file
--MSGFORMAT<=*format*>	Format general message strings to the given style
--NODEL	Do not remove temporary files generated by the compiler
--NOEXEC	Go through the motions of compiling without actually compiling
--OPT<=*type*>	Enable general compiler optimizations
--OUTDIR	Specify output files directory
--OUTPUT=*type*	Generate output file type
--PRE	Produce preprocessed source files
--PROTO	Generate function prototype information
--RAM=lo-hi<,lo-hi,...>	Specify and/or reserve RAM ranges
--ROM=lo-hi<,lo-hi,...>	Specify and/or reserve ROM ranges
--RUNTIME=*type*	Configure the C runtime libraries to the specified type
--SCANDEP	Generate file dependency ".DEP files"
--SERIAL=*code@address*	Store this hexadecimal code at an address in program memory
--SETOPTION=*app,file*	Set the command line options for the named application
--SETUP=argument	Setup the product
--STRICT	Enable strict ANSI keyword conformance
--SUMMARY=*type*	Selects the type of memory summary output
--VER	Display the compiler's version number
--WARN=*level*	Set the compiler's warning level
--WARNFORMAT=*format*	Format warning message strings to given style