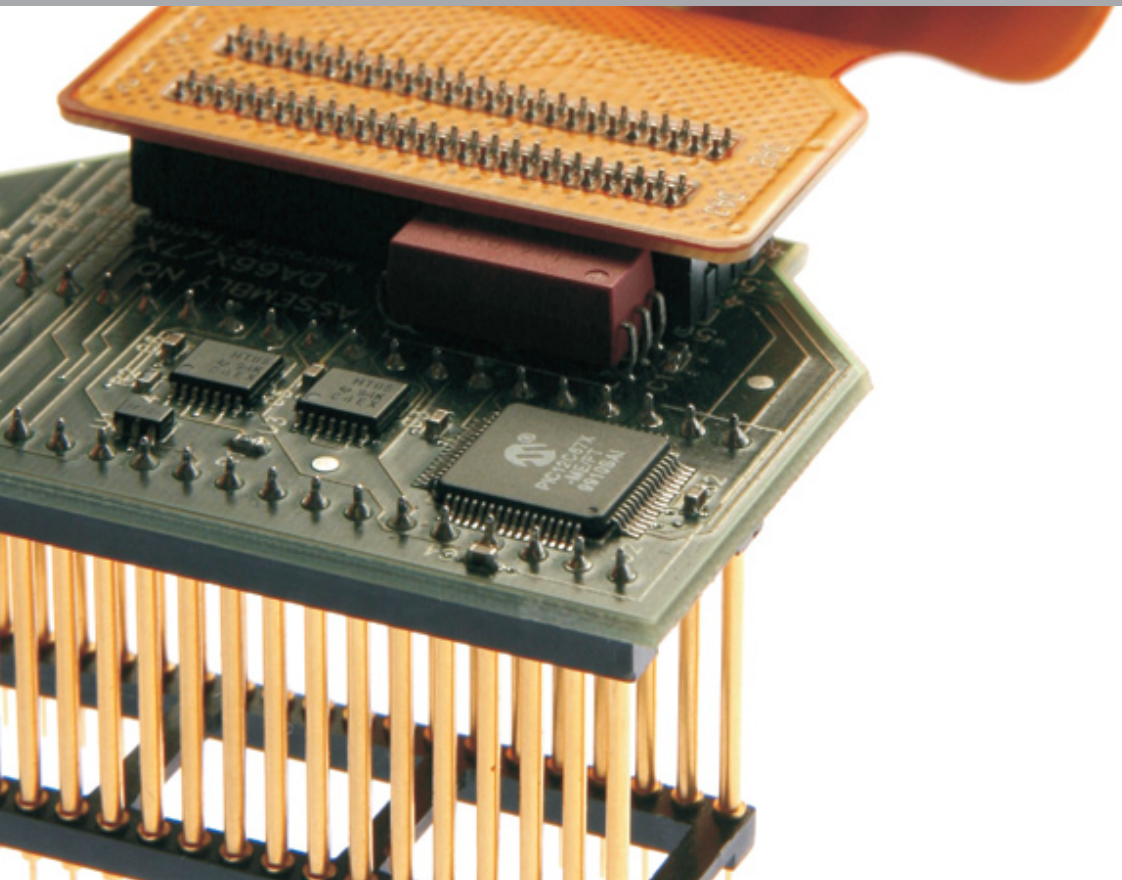
HI-TECH SOFTWARE

Manual

HI-TECH C® Tools for the PIC10/12/16 MCU Family



HI-TECH C Tools for the PIC10/12/16 MCU Family

HI-TECH Software.

Produced on: March 26, 2009

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

email: hitech@htsoft.com
web: http://microchip.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 Command-line Driver

PICC is the driver invoked from the command line to perform all aspects of compilation, including C code generation, assembly and link steps. It is the recommended way to use the compiler as it hides the complexity of all the internal applications used in the compilation process and provides a consistent interface for all compilation steps.

This chapter describes the steps the driver takes during compilation, files that the driver can accept and produce, as well as the command-line options that control the compiler's operation.

WHAT IS "THE COMPILER"? Throughout this manual, the term "the compiler" is used to refer to either all, or some subset of, the collection of applications that form the HI-TECH C PRO for the PIC10/12/16 MCU Family package. Often it is not important to know, for example, whether an action is performed by the parser or code generator application, and it is sufficient to say it was performed by "the compiler".

It is also reasonable for "the compiler" to refer to the command-line driver (or just "driver"), PICC, as this is the application executed to invoke the compilation process. Following this view, "compiler options" should be considered command-line driver options, unless otherwise specified in this manual.

Similarly "compilation" refers to all, or some part of, the steps involved in generating source code into an executable binary image.

Table 2.1: PICC input file types

File Type	Meaning
`.c`	C source file
`.p1`	p-code file
`.lpp`	p-code library file
`.as`	Assembler source file
`.obj`	Relocatable object code file
`.lib`	Relocatable object library file
`.hex`	Intel HEX file

2.1 Invoking the Compiler

This chapter looks at how to use PICC as well as the tasks that it and the internal applications perform during compilation.

`PICC` has the following basic command format:

```
PICC [options] files [libraries]
```

It is conventional to supply *`options`*(identified by a leading *dash* "-" or *double dash* "–") before the filenames, although this is not mandatory.

The formats of the options are discussed below in Section 2.6, and a detailed description of each option follows.

The *`files`* may be any mixture of C and assembler source files, and precompiled intermediate files, such as relocatable object (`.obj`) files or p-code (`.p1`) files. The order of the files is not important, except that it may affect the order in which code or data appears in memory, and may affect the name of some of the output files.

`Libraries` is a list of either object code or p-code library files that will be searched by the linker. The `-L` option, see Section 2.6.6, can also be used to specify library files to search.

`PICC` distinguishes source files, intermediate files and library files 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 of the extension is not important from the compiler's point of view.

MODULES AND SOURCE FILES: A C *source file* is a file on disk that contains all or part of a program. C source files are initially passed to the preprocessor by the driver. A *module* is the output of the preprocessor, for a given source file, after inclusion of any header files (or other source files) which are specified by `#include` preprocessor

directives. These modules are then passed to the remainder of the compiler applications. Thus, a module may consist of several source and header files. A module is also often referred to as a *translation unit*. These terms can also be applied to assembly files, as they too can include other header and source files.

Some of the compiler's output files contain project-wide information and are not directly associated with any one particular input file, e.g. the map file. If the names of these project-wide files are not specified on the command line, the basename of these files is derived from the first C source file listed on the command line. If there are no files of this type being compiled, the name is based on the first input file (regardless of type) on the command line. Throughout this manual, the basename of this file will be called the *project name*.

Most IDEs use project files whose names are user-specified. Typically the names of project-wide files, such as map files, are named after the project, however check the manual for the IDE you are using for more details.

2.1.1 Long Command Lines

The PICC 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 which should be immediately followed (i.e. no intermediate space character) by the name of the file containing the command line arguments.

The file may contain blank lines, which are simply skipped by the driver. The command-line arguments may be placed over several lines by using a *space* and *backslash* character for all non-blank lines, except for the last line.

The use of a command file means that compiler options and project filenames can be stored along with the project, making them more easily accessible and permanently recorded for future use.

TUTORIAL

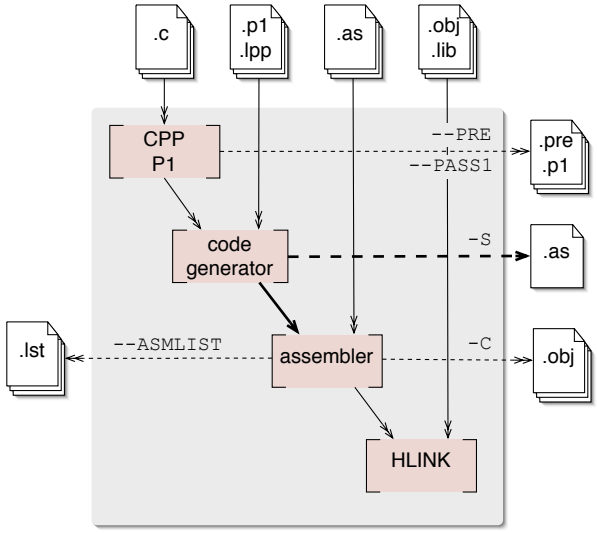
USING COMMAND FILES A command file `xyz.cmd` is constructed with your favorite text editor and contains both the options and file names that are required to compile your project as follows:

```
--chip=16F877A -m \
--opt=all -g \
main.c isr.c
```

After it is saved, the compiler may be invoked with the command:

```
PICC @xyz.cmd
```

Figure 2.1: Flow diagram of the initial compilation sequence



2.2 The Compilation Sequence

`PICC` will check each file argument and perform appropriate actions on each file. The entire compilation sequence can be thought of as the initial sequence up to the link stage, and the final sequence which takes in the link step and any post link steps required.

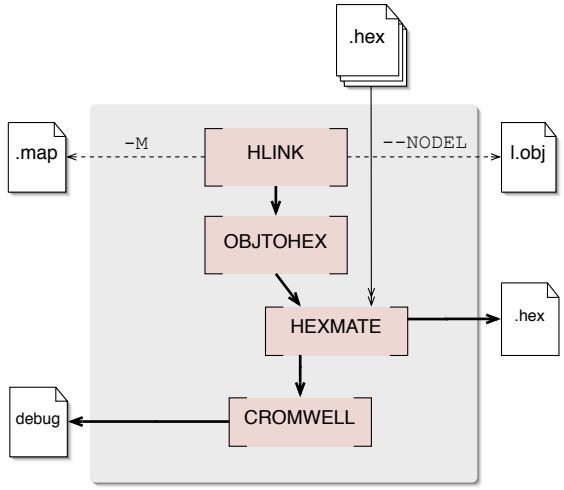
Graphically the compilation steps up to the link stage are illustrated in Figure 2.1. This diagram shows all possible input files along the top; intermediate and transitional files, along the right side; and useful compiler output files along the left. Generated files are shown along with the options that are used to generate and preserve these. All the files shown on the right, can be generated and fed to the compiler in a subsequent compile step; those on the left are used for debug purposes and cannot be used as an input to any subsequent compilation.

The individual compiler applications are shown as boxes. The C preprocessor, `CPP`, and parser, `P1`, have been grouped together for clarity.

The thin, multi-arrowed lines indicate the flow of multiple files — one for each file being processed by the revel ant application. The thick single-arrowed lines indicate a single file for the project being compiled. Thus, for example, when using the `--PASS1` driver option, the parser produces one `.p1` file for each C source file that is being compiled as part of the project, but the code generator produces only one `.as` file from all `.c`, `.p1` and `.lpp` input files which it is passed.

Dotted lines indicate a process that may require an option to create or preserve the indicated file.

Figure 2.2: Flow diagram of the final compilation sequence



The link and post-link steps are graphically illustrated in Figure 2.2.

This diagram shows `.hex` files as additional input file type not considered in the initial compilation sequence. These files can be merged into the `.hex` file generated from the other input files in the project by an application called `HEXMATE`. See Section 5.15 for more information on this utility.

The output of the linker is a single absolute object file, called `l.obj`, that can be preserved by using the `--NODEL` driver option. Without this option, this temporary file is used to generate an output file (e.g. a HEX file) and files used for debugging by development tools (e.g. COFF files) before it is deleted. The file `l.obj` can be used as the input to `OBJTOHEX` if running this application manually, but it cannot be passed to the driver as an input file as it absolute and cannot be further processed.

2.2.1 Single-step Compilation

The command-line driver, PICC, can compile any mix of input files in a single step. All source files will be re-compiled regardless of whether they have been changes since that last time a compilation was performed.

Unless otherwise specified, a default output file and debug file are produced. All intermediate files (`.p1` and `.obj`) remain after compilation has completed, but all other transitional files are deleted, unless you use the `--NODEL` option which preserves all generated files. Note some generated files may be in a temporary directory not associated with your project and use a pseudo-

randomly generated filename.

TUTORIAL

SINGLE STEP COMPILATION The files, `main.c`, `io.c`, `mdef.as`, `sprt.obj`, `a_sb.lib` and `c_sb.lpp` are to be compiled. To perform this in a single step, the following command line can be used as a starting point for the project development.

```
PICC --chip=16F877A main.c io.c mdef.as sprt.obj a_sb.lib c_sb.lpp
```

This will run the C pre-processor then the parser with `main.c` as input, and then again for `io.c` producing two p-code files. These two files, in addition to the library file `c_sb.lpp`, are passed to the code generator producing a single temporary assembler file output. The assembler is then executed and is passed the output of the code generator. It is run again with `mdef.as`, producing two relocatable object files. The linker is then executed, passing in the assembler output files in addition to `sprt.obj` and the library file `a_sb.lib`. The output is a single absolute object file, `l.obj`. This is then passed to the appropriate post-link utility applications to generate the specified output file format and debugging files. All temporary files, including `l.obj`, are then deleted. The intermediate files: p-code and relocatable object files, are not deleted. This tutorial does not consider the runtime startup code that is automatically generated by the driver.

2.2.2 Generating Intermediate Files

The HI-TECH C PRO for the PIC10/12/16 MCU Family version compiler uses two types of intermediate files. For C source files, the p-code file (`.p1` file) is used as the intermediate file. For assembler source files, the relocatable object file (`.obj` file) is used.

You may wish to generate intermediate files for several reasons, but the most likely will be if you are using an IDE or make system that allows an incremental build of the project. The advantage of a incremental build is that only the source files that have been modified since the last build need to be recompiled before again running the final link step. This dependency checking may result in reduced compilation times, particularly if there are a large number of source files.

You may also wish to generate intermediate files to construct your own library files, although PICC is capable of constructing libraries in a single step, so this is typically not necessary. See Section 2.6.46 for more information.

Intermediate files may also assist with debugging a project that fails to work as expected.

If a multi-step compilation is required the recommended compile sequence is as follows.

- Compile all modified C source files to p-code files using the `--PASS1` driver option

- Compile all modified assembler source files to relocatable object files using the `-C` driver option
- Compile all p-code and relocatable object files into a single output object file

The final step not only involves the link stage, but also code generation of all the p-code files. In effect, the HI-TECH C PRO for the PIC10/12/16 MCU Family version code generator performs some of the tasks normally performed by the linker. Any user-specified (non standard) libraries also need to be passed to the compiler during the final step. This is the incremental build sequence used by HI-TIDE™.

TUTORIAL

MULTI-STEP COMPILATION The files in the previous example are to be compiled using a multi-step compilation. The following could be used.

```
PICC --chip=16F877A --pass1 main.c
PICC --chip=16F877A --pass1 io.c
PICC --chip=16F877A -c mdef.as
PICC --chip=16F877A main.p1 io.p1 mdef.obj sprt.obj c_sb.lpp
a_sb.lib
```

If using a make system with incremental builds, only those source files that have changed since the last build need the first compilation step performed again, so not all of the first three steps need be executed.

If is important to note that the code generator needs to compile all p-code or p-code library files in the one step. Thus, if the `--PASS1` option is not used (or `--PRE` is not used), all C source files, and any p-code libraries, must be built together in the one command.

If a compilation is performed, and the source file that contains `main()` is not present in the list of C source files, an undefined symbol error for `_main` will be produced by the code generator. If the file that contains the definition for `main()` is present, but it is a subset of the C source files making up a project that is being compiled, the code generator will not be able to see the entire C program and this will defeat most of the optimization techniques employed by the code generator.

There may be multi-step compilation methods employed that lead to compiler errors as a result of the above restrictions, for example you cannot have an C function compiled into a p-code library that is called only from assembler code.

2.2.3 Special Processing

There are several special steps that take place during compilation.

2.2.3.1 Printf check

An extra execution of the code generator is performed for prior to the actual code generation phase. This pass is part of the process by which the printf library function is customized, see Section 2.3.4 for more details.

2.2.3.2 Assembly Code Requirements

After pre-processing and parsing of any C source files, but before code generation of these files, the compiler assembles any assembly source files to relocatable object files. These object files, together with any object files specified on the command line, are scanned by the compiler driver and certain information from these files are collated and passed to the code generator. Several actions are taken based on this information. See Section 3.10.4.

The driver instructs the code generator to preserve any C variables which map to symbols which are used, but not defined, in the assembly/object code. This allows variables to be defined in C code, and only every referenced in assembly code. Normally such C variables would be removed as the code generator would consider them to be used (from the C perspective). Specifically, the C variables are automatically qualified as being `volatile` which is sufficient to prevent the code generator making this optimization.

The driver also takes note of any absolute psects (viz. use the `abs` and `ovrld PSECT` directive flags) in the assembly/object code. The memory occupied by the psects is removed from the available memory ranges passes to the code generator and linker. This information ensures that this memory is not allocated to any C resources.

2.3 Runtime Files

In addition to the input files specified on the command line by the user, there are also compiler-generated source files and pre-compiled library files which might be compiled into the project by the driver. These are:

- Library files;
- The runtime startup module;
- The powerup routine; and
- The `printf` routine.

Strictly speaking the powerup routine is neither compiler-generated source, nor a library routine. It is fully defined by the user, however as it is very closely associated with the runtime startup module, it is discussed with the other runtime files in the following sections.

By default, libraries appropriate for the selected driver options are automatically passed to the code generator and linker. Although individual library functions or routines will be linked in once referenced in C code, the compiler still requires the inclusion of the appropriate header file for the library function that is being used. See the appropriate library function section in Chapter A for the header file that should be used.

2.3.1 Library Files

By default,PICC will search the `LIB` directory of the compiler distribution for several p-code and relocatable object library files, which are then passed to the code generator and linker, respectively. These library files are associated with:

- The C standard library functions
- Assembly routines implicitly called by the code generator
- Chip-specific peripherals functions
- Chip-specific memory functions

These library files are always scanned after scanning any user-specified libraries passed to the driver on the command line, thus allowing library routines to be easily replaced with user-defined alternatives. See Section 3.12.1.

The C standard libraries and libraries of implicitly-called assembly routines can be omitted from the project by disabling the `clib` suboption of `--RUNTIME`. 2.6.52. For example:

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

If these libraries are excluded from the project then calls to any routine, or access of any variable, that is defined in the omitted library files will result in an error from the linker. The user must provide alternative libraries or source files containing definitions for any routine or symbol accessed by the project.

Do not confuse the actual library (`.lib`) files and the header (`.h`) files. Both are covered by a library package, but the library files contain precompiled code, typically functions and variable definitions; the header files provide declarations (as opposed to definitions) for functions, variables and types in the library files, as well as other preprocessor macros. PICC will always link in all the library files associated with the C standard library (unless you have used an option to prevent this), however with user-defined library packages, the inclusion of a header does not imply that the corresponding library file(s) will be searched.

2.3.1.1 Standard Libraries

The C standard libraries contain a standardised collection of functions, such as string, math and input/output routines. The range of these functions are described in Appendix A.

These libraries also contain C routines that are implicitly called by the output code of the code generator. These are routines that perform tasks such as floating point operations, and that do not directly correspond to a C function call in the source code.

The general form of the standard library names is `htpic-`*dc*`.`*ext*. The meaning of each field is described by:

- Processor Type is always `pic`.
- The double type, *d*, is `-` for 24-bit doubles, and `d` for 32-bit doubles.
- Library Type is always `c`.
- The extension is `.lpp` for p-code libraries, or `.lib` for relocatable object libraries.

Typically there will only be an `.lpp` version of each library, however there may also be a `.lib` version in some cases.

2.3.2 Runtime Startup Module

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, specifically:

- Initialisation of global variables assigned a value when defined
- Clearing of non-initialised global variables
- General setup of registers or processor state

Rather than the traditional method of linking in a generic, precompiled routine, HI-TECH C PRO for the PIC10/12/16 MCU Family uses a more efficient method which actually determines what runtime startup code is required from the user's program. It does this by performing an additional link step, the output of which is used to determine the requirements of the program. From this information PICC then "writes" the assembler code which will perform the startup sequence. This code is stored into a file which is then assembled and linked into the remainder of the program automatically.

The runtime startup code is generated automatically on every compilation. If required, the assembler file which contains the runtime startup code can be deleted after compilation by using the driver option:

```
--RUNTIME=default,-keep
```

If the startup module is kept, it 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. Section 2.6.52 describes the use of this option, and the following sections describes the functional aspects of the code contained in this module and its effect on program operation.

If you require any special initialization to be performed immediately after reset, you should use the *powerup* routine feature described later in Section 2.3.3.

2.3.2.1 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, for example `input` in the following example.

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

Such initialized objects have two components: their initial value stored in a psect destined for non-volatile memory (i.e. placed in the HEX file), and space for the variable in RAM psect where the variable will reside and be accessed during program execution.

The actual initial values are placed in a psect called `idata`. Space is reserved for the runtime location of initialized variables in a psect called `rdata`. This psect does not contribute to the output file and constitutes a reservation of space in the RAM once the program is running.

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 the corresponding 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.

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 and are not considered by the runtime startup code.

Variables whose contents should be preserved over a reset, or even power off, should be qualified with persistent, see Section 3.3.10.1. Such variables are linked at a different area of memory and are not altered by the runtime startup code in any way.

2.3.2.2 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 one of the bss psects. 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.

HI-TECH C PRO for the PIC10/12/16 MCU Family uses several bss psects. There are the more traditional psects: `rbss` and `bss`, which are used for uninitialized variables placed in the access bank memory, and in banked memory, respectively. However, most uninitialized variables are allocated memory by the code generator directly in the data space RAM without being located in a psect at all. They are then handled as if they were absolute variables.

To ensure that variables allocated memory by the code generator are cleared, symbols are defined that are used by the command line driver to generate the appropriate code. The symbols have the form: `__Labsbss`*n* and `__Habsbss`*n*, where *n* is a number starting from 0. As these uninitialized absolute variables can be placed anywhere in available memory, and are not restricted to being placed in a single large block, there may be more than one set of these symbols defined to ensure that all blocks are cleared.

These symbols look like the `__Lxxxx` and `__Hxxxx` symbols defined by the linker to represent the upper and lower bounds of a psect, and can be used in the same way. See Section 3.12.3.

Assembly code that defines variables which should be cleared at startup should be placed in the `rbss` and `bss` psects. Do not create and use a psect with a name of the form `absbss`*n*. Appropriate default linker options will be issued for `rbss` and `bss`, thus their use does not require

modification of the linker options, and the command-line driver will automatically check the size of these psects to determine if block-clear code is required. Variables placed into psects other than the compiler-defined bss psects will not be cleared at startup by default.

The block clear of all the bss psects (including the memory allocated by the code generator) can 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.

Variables whose contents should be preserved over a reset, or even power off, should be qualified with `persistent`. See Section 3.3.10.1 for more information. Such variables are linked at a different area of memory and are not altered by the runtime startup code in anyway.

2.3.3 The Powerup Routine

Some hardware configurations require special initialization, often within the first few instruction cycles after reset. To achieve this there is a hook to the reset vector provided via the *powerup* routine.

This routine can be supplied in a user-defined assembler module that will be executed immediately after reset. An empty powerup routine is provided in the file `powerup.as` which is located in the SOURCES directory of your compiler distribution. Refer to comments in this file for more details.

The file should be copied to your working directory, modified and included into your project as a source file. No special linker options or other code is required; the compiler will detect if you have defined a powerup routine and will automatically use it, provided the code in this routine is contained in a psect called `powerup`.

For correct operation (when using the default compiler-generated runtime startup code), the code must contain at its end a `GOTO` instruction to the label called `start`. As with all user-defined assembly code, it must take into consideration program memory paging and/or data memory banking, as well as any applicable errata issues for the device you are using. The program's entry point is already defined by the runtime startup code, so this should not be specified in the powerup routine at the `END` directive (if used). See Section 4.3.9.2 for more information on this assembler directive.

2.3.4 The `printf` Routine

The code associated with the `printf` function is not found in the library files. The `printf` function is generated from a special C source file that is customized after analysis of the user's C code. See page 281 for more information on the `printf` library function.

This template file is found in the LIB directory of the compiler distribution and is called `doprnt.c`. It contains a minimal implementation of the `printf` function, but with the more advanced features included as conditional code which can be utilized via preprocessor macros that are defined when it is compiled.

The parser and code generator analyze the C source code, searching for calls to the `printf` function. For all calls, the placeholders that were specified in the `printf` format strings are collated to produce a list of the desired functionality of the final function. The `doprnt.c` file is then preprocessed with the those macros specified by the preliminary analysis, thus creating a custom `printf` function for the project being compiled. After parsing, the p-code output derived from `doprnt.c` is then combined with the remainder of the C program in the final code generation step.

TUTORIAL

CALLS TO PRINTF A program contains one call to `printf`, which looks like:

```
printf("input is:  $d");
```

The compiler will note that only the `%d` placeholder is used and the doprnt module that is linked into the program will only contain code that handles printing of decimal integers.

The code is latter changed and another call to `printf` is added. The new call looks like:

```
printf("output is %6d");
```

Now the compiler will detect that in addition there must be code present in the doprnt module that handles integers printed to a specific width. The code that handles this flag will be introduced into the doprnt module.

The size of the doprnt module will increase as more `printf` features are detected.

If the format string in a call to `printf` is not a string literal as in the tutorial, but is rather a pointer to a string, then the compiler will not be able to reliably predict the `printf` usage, and so it forces a more complete version of `printf` to be generated. However, even without being able to scan `printf` placeholders, the compiler can still make certain assumptions regarding the usage of the function. In particular, the compiler can look at the number and type of the additional arguments to `printf` (those following the format string expression) to determine which placeholders could be valid. This enables the size and complexity of the generated `printf` routine to be kept to a minimum.

TUTORIAL

PRINTF WITHOUT LITERAL FORMAT STRINGS If there is only one reference to `printf` in a program and it appears as in the following code:

```
void my_print(const char * mes) {
  printf(mes);
}
```

the compiler cannot determine the exact format string, but can see that there are no additional arguments to `printf` following the format string represented by `mes`. Thus, the only valid format strings will not contain placeholders that print any arguments, and a minimal version of `printf` will be generated and compiled. If the above code was rewritten as:

```
void my_print(const char * mes, double val) {
  printf(mes, val);
}
```

the compiler will detect that the argument being printed has `double` type, thus the only valid placeholders would be those that print floating point types, for example `%e`, `%f` and `%g`.

No aspect of this operation is user-controllable (other than by adjusting the calls to `printf`), however the actual `printf` code used by a program can be observed. If compiling a program using `printf`, the driver will leave behind the pre-processed version of `doprnt.c`. This module, called `doprnt.pre` in your working directory, will show the C code that will actually be contained in the `printf` routine. As this code has been pre-processed, indentation and comments will have been stripped out as part of the normal actions taken by the C pre-processor.

2.4 Debugging Information

Several driver options and output files are related to allow development tools, such as HI-TIDE™ or MPLAB®, to perform source-level debugging of the output code. These are described in the following sections.

2.4.1 Output File Formats

The compiler is able to directly produce a number of the output file formats which are used by common PROM programmers and in-circuit emulators.

The default behaviour of the PICC command is to produce *Bytecraft* COD, *Microchip* COFF and *Intel* HEX output. If no output filename or type is specified, PICC will produce a *Bytecraft* COD, *Microchip* COFF and *Intel* HEX file with the same base name as the first source or object file specified on the command line. Table 2.12 shows the output format options available with HI-TECH

C PRO for the PIC10/12/16 MCU Family. 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`, PICC will produce a binary file. Likewise, if you need to produce UBROF files, you can use the -O option to specify an output file with type `.ubr`, for example `-Otest.ubr`.

2.4.2 Symbol Files

The PICC `-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 been 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:

```
PICC --CHIP=16F877A -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`.

2.4.3 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 options generates additional global symbols which can be used to represent most local symbols in a program. The format for the symbols is ***function***.*symbol*. Thus, if a variable called `foo` was defined inside the function `main()`, MPLAB would allow access to a global object called `main.foo`. This symbol format is not available in assembler code. References to this object in assembler would be via the symbol

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.

2.5 Compiler Messages

All compiler applications, including the command-line driver, PICC, use textual messages to report feedback during the compilation process. A centralized messaging system is used to produce the messages which allows a consistancy during all stages of the compilation process.

2.5.1 Messaging Overview

A message is referenced by a unique number which is passed to the alert system by the compiler application that needs to convey the information. The message string corresponding to this number is obtained from Message Description Files (MDF) which are stored in the `DAT` directory of the compiler distribution.

When a message is requested by a compiler application, its number is looked up in the MDF which corresponds to the currently selected language. The language of messages can be altered as discussed in Section 2.5.2.

Once found, the alert system determines the message type that should be used to display the message. There are several different message types which are described in Section 2.5.3. The default type is stored in the MDF, however this can be overridden by the user, as described in Section 2.5.3. The user is also able to set a threshold for warning message importance, so that only those which the user considers significant will be displayed. In addition, messages with a particular number can be disabled. Both of these methods are explained in Section 2.5.5.1.

Provided the message is enabled and it is not a warning messages that is below the warning threshold, the message string will be displayed.

In addition to the actual message string, there are several other pieces of information that may be displayed, such as the message number, the name of the file for which the message is applicable, the file's line number and the application that requested the message, etc.

If a message is being displayed as an error, a counter is incremented. After a certain number of errors has been reached, compilation of the current module will cease. The default number of errors that will cause this termination can be adjusted by using the `--ERRORS` option, see Section 2.6.30. This counter is reset after each compilation step of each module, thus specifying a maximum of five errors will allow up to five errors from the parser, five from the code generator, five from the linker, five from the driver, etc.

If a language other than English is selected, and the message cannot be found in the appropriate non-English MDF, the alert system tries to find the message in the English MDF. If an English message string is not present, a message similar to:

```
error/warning (*) generated, but no description available
```

where * indicates the message number that was generated, will be printed, otherwise the message in the requested language will be displayed.

Table 2.2: Support languages

Language	MDF name
English	en_msgs.txt
German	de_msgs.txt
French	fr_msgs.txt

2.5.2 Message Language

HI-TECH C PRO for the PIC10/12/16 MCU Family Supports more than one language for displayed messages. There is one MDF for each language supported.

The language used for messaging may be specified with each compile using the --LANG option, see Section 2.6.36. Alternatively it may be set up in a more permanent manner by using the --LANG option together with the --SETUP option which will store the default language in either the registry, under Windows, or in a configuration file on other systems. On subsequent builds the default language used will be that specified.

Table shows the MDF applicable for the currently supported languages.

2.5.3 Message Type

There are four types of message whose default behaviour is described below.

Advisory Messages convey information regarding a situation the compiler has encountered or some action the compiler is about to take. The information is being displayed "for your interest" and typically require no action to be taken.
Unless prevented by some driver option or another error message, the project will be linked and the requested output file(s) will be generated.

Warning Messages indicate source code or some other situation that is valid, but which may lead to runtime failure of the code. The code or situation that triggered the warning should be investigated, however, compilation of the current module will continue, as will compilation of any remaining modules.
Unless prevented by some driver option or another error message, the project will be linked and the requested output file(s) will be generated.

Error Messages indicate source code that is illegal and that compilation of this code either cannot or will not take place. Compilation will be attempted for the remaining source code in the current module, but no additional modules will be compiled and the compilation process will then conclude.
The requested output files will not be produced.

Table 2.3: Messaging environment variables

Variable	Effect
HTC_MSG_FORMAT	All advisory messages
HTC_WARN_FORMAT	All warning messages
HTC_ERR_FORMAT	All error and fatal error messages

Fatal Error Messages indicate a situation that cannot allow compilation to proceed and which required the the compilation process to stop immediately.
The requested output files will not be produced.

2.5.4 Message Format

By default, messages are printed in the most useful human-readable format as possible. This format can vary from one compiler application to another, since each application reports information about different file formats. Some applications, for example the parser, are typically able to pinpoint the area of interest down to a position on a particular line of C source code, whereas other applications, such as the linker, can at best only indicate a module name and record number, which is less directly associated with any particular line of code. Some messages relate to driver options which are in no way associated with any source code.

There are several ways of changing the format in which message are displayed, which are discussed below.

The driver option -E (with or without a filename) alters the format of all displayed messages. See Section 2.6.3. Using this option produces messages that are better suited to machine parsing, and user-friendly. Typically each message is displayed on a single line. The general form of messages produced with the -E option in force is:

```
filename line_number: (message number) message string (message type)
```

The -E option also has another effect. If it is being used, the driver first checks to see if special environment variables have been set. If so, the format dictated by these variables are used as a template for all messages produced by all compiler applications. The names of these variables are given in Table 2.3.

The value of these environment variables are strings that are used as templates for the message format. Printf-like placeholders can be placed within the string to allow the message format to be customised. The placeholders and what they represent are indicated in Table 2.4.

If these options are used in a DOS batch file, two percent characters will need to be used

Table 2.4: Messaging placeholders

Placeholder	Replacement
%a	application name
%c	column number
%f	filename
%l	line number
%n	message number
%s	message string (from MDF)

to specify the placeholders, as DOS interprets a single percent character as an argument and will not pass this on to the compiler. For example:

```
--ERRFORMAT="file %%f:  line %%l"
```

Environment variables, in turn, may be overridden by the driver options: --MSGFORMAT, --WARNFORMAT and --ERRFORMAT, see Section 2.6.29. These options take a string as their argument. The option strings are formatted, and can use the same placeholders, as their variable counterparts.

TUTORIAL

CHANGING MESSAGE FORMATS A project is compiled, but produces a warning from the parser and an error from the linker. By default the following messages are displayed when compiling.

```
main.c:  main()
17:  ip = &b;
^ (362) redundant "&" applied to array (warning)
(492) attempt to position absolute psect "text" is illegal
```

Notice that the format of the messages from the parser and linker differ since the parser is able to identify the particular line of offending source code. The parser has indicated the name of the file, indicated the function in which the warning is located, reproduced the line of source code and highlighted the position at which the warning was first detected, as well as show the actual warning message string.

The -E option is now used and the compiler issues the same messages, but in a new format as dictated by the -E option. Now environment variables are set and no other messaging driver options were specified so the default -E format is used.

```
main.c:  12:  (362) redundant "&" applied to array (warning)
(492) attempt to position absolute psect "text" is illegal (error)
```

Notice that now all message follow a more uniform format and are displayed on a single line.

The user now sets the environment variable HTC_WARN_FORMAT to be the following string. (Under Windows, this can be performed via the Control Panel's System panel.)

```
%a %n %l %f %s
```

and the project recompiled. The following output will be displayed.

```
parser 362 12 main.c redundant "&" applied to array (492)
attempt to position absolute psect "text" is illegal (error)
```

Notice that the format of the warning was changed, but that of the error message was not. The warning format now follows the specification of the environment variable. The application name (parser) was substituted for the %a placeholder, the message number (362) substituted the %n placeholder, etc.

The option --ERRFORMAT="%a %n %l %f %s" is then added to the driver command line and the following output is observed.

```
parser 362 12 main.c redundant "&" applied to array
linker 492   attempt to position absolute psect "text" is illegal
```

Note that now the warning and error formats have changed to that requested. For the case of the linker error, there is no line number information so the replacement for this placeholder is left blank.

2.5.5 Changing Message Behaviour

Both the attributes of individual messages and general settings for messaging system can be modified during compilation. There are both driver command-line options and C pragmas that can be used to achieve this.

2.5.5.1 Disabling Messages

Each warning message has a default number indicating a level of importance. This number is specified in the MDF and ranges from -9 to 9. The higher the number, the more important the warning.

Warning messages can be disabled by adjusting the warning level threshold using the --WARN driver option, see Section 2.6.60. Any warnings whose level is below that of the current threshold are not displayed. The default threshold is 0 which implies that only warnings with a warning level of 0

or higher will be displayed by default. The information in this option is propagated to all compiler applications, so its effect will be observed during all stages of the compilation process.

Warnings may also be disabled by using the `--MSGDISABLE` option, see Section 2.6.39. This option takes a comma-separated list of message numbers. Any warnings which are listed are disabled and will never be issued, regardless of any warning level threshold in place. This option cannot be used to disable error messages.

Some warning messages can also be disabled by using the `warning disable` pragma. This pragma will only affect warnings that are produced by either parser or the code generator, i.e. errors directly associated with C code. See Section 3.11.3.8 for more information on this pragma.

Error messages can also be disabled, however a slight more verbose form of the command is required to confirm the action required. To specify an error message number in the `--MSGDISABLE` command, the number must be followed by `:off` to ensure that it is disabled. For example: `--MSGDISABLE=195:off` will disable error number 195.

Disabling error or warning messages in no way fixes any potential problems reported by the message. Always use caution when exercising this option.

2.5.5.2 Changing Message Types

It is also possible to change the type of some messages. This is only possible by the use of the `warning pragma` and only affects messages generated by the parser or code generator. See Section 3.11.3.8 for more information on this pragma.

2.6 PICC Driver Option Descriptions

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

PICC 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.

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 represent the default specification that would be used if this option was absent altogether; `all`, which indicates that all 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.

See the –HELP option, Section 2.6.34, for more information about options and suboptions.

2.6.1 `-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 assembly source files using a "make" utility. Use of this option when only a subset of all the C source files in a project are being compiled will result in an error from the code generator. See Section 2.2.2 for more information on generating and using intermediate files.

2.6.2 `-Dmacro`: 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:

```
PICC --CHIP=16F877A -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
```

See Section 2.7 for use of this option in MPLAB IDE.

2.6.3 -Efile: Redirect Compiler Errors to a File

This option has two purposes. The first is to change the format of displayed messages. The second is to optionally allow messages to be directed to a file as some editors do not allow the standard command line redirection facilities to be used when invoking the compiler.

The gernal form of messages produced with the -E option in force is:

`filename line_number: (message number) message string (message type)`

If a filename is specified immediately after -E, it is treated as the name of a file to which all messages (errors, warnings etc) will be printed. For example, to compile `x.c` and redirect all errors to `x.err`, use the command:

```
PICC --CHIP=16F877A -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:

```
PICC --CHIP=16F877A -E+x.err y.c
```

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

```
PICC --CHIP=16F877A -Eproject.err -O --PASS1 main.c
PICC --CHIP=16F877A -E+project.err -O --PASS1 part1.c
PICC --CHIP=16F877A -E+project.err -C asmcode.as
```

Section 2.5 has more information regarding this option as well as an overview of the messaging system and other related driver options.

2.6.4 -Gfile: 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 project name (see Section 2.1), 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:

```
PICC --CHIP=16F877A -G --PASS1 test.c modules1.c
PICC --CHIP=16F877A -Gtest.sym test.p1 module1.p1
```

The --IDE option, see Section 2.6.35 will typically enable the -G option.

2.6.5 -I*path*: Include Search Path

Use -I to specify an additional directory to use when searching for header files which have been included using the #include directive. The -I option can be used more than once if multiple directories are to be searched.

The default include directory containing all standard header files are 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:

```
PICC --CHIP=16F877A -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).

This option has no effect for files that are included into assembly source using the INCLUDE directive. See Section 4.3.10.3.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.6 -L*library*: Scan Library

The -L option is used to specify additional libraries which are to be scanned by the linker. Libraries specified using the -L option are scanned before the standard C library, allowing additional versions of standard library functions to be accessed.

The argument to -L is a library keyword to which the prefix pic; numbers representing the processor range, number of ROM pages and the number of RAM banks; and the suffix .lib are added. Thus the option -LL when compiling for a 16F877 will, for example, scan the library pic42c-l.lib and the option -Lxx will scan a library called pic42c-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.

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

The `-L` driver option can also be used to specify an option which will be passed directly to the linker. If `-L` is followed immediately by text starting with a *dash* character "–", the text will be passed directly to the linker without being interpreted by PICC. For example, if the option `-L-FOO` is specified, the `-FOO` option will be passed on to the linker. The linker will then process this option, when, and if, it is invoked, and perform the appropriate function, or issue an error if the option is invalid.

Take care with command-line options. The linker cannot interpret driver options; similarly the command-line driver cannot interpret linker options. In most situations, it is always the command-line driver, PICC, that is being executed. If you need to add alternate settings in the linker tab in an MPLAB `Build options...` dialogue, these are the *driver* options (not linker options), but which are used by the driver to generate the appropriate linker options during the linking process.

The `-L` option is especially useful when linking code which contains non-standard program sections (or psects), as may be the case if the program contains assembly code which contains user-defined psects. Without this `-L` option, it would be necessary to invoke the linker manually to allow the linker options to be adjusted.

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 PICC as the option `-L-N`.

This option can also be used to replace default linker options: If the string starting from the first character after the `-L` up to the first = character matches first part of a default linker option, then that default linker option is replaced by the option specified by the `-L`.

TUTORIAL

REPLACING DEFAULT LINKER OPTIONS In a particular project, the psect `entry` is used, but the programmer needs to ensure that this psect is positioned above the address 800h. This can be achieved by adjusting the default linker option that positions this psect. First, a map file is generated to determine how this psect is normally allocated memory. The `Linker command line:` in the map file indicates that this psect is

normally linked using the linker option:
`-pentry=CODE`
Which places `entry` anywhere in the memory defined by the `CODE` class. The programmer then re-links the project, but now using the driver option:
`-L-pentry=CODE+800h`
to ensure that the psect is placed above 800h. Another map file is generated and the `Linker command line:` section is checked to ensure that the option was recieved and executed by the linker. Next, the address of the psect `entry` is noted in the psect lists that appear later in the map file. See Section 5.10 for more information on the contents of the map file.

If there are no characters following the first = character in the `-L` option, then any matching default linker option will be deleted. For example: `-L-pfirst=` will remove any default linker option that begins with the string `-pfirst=`. No warning is generated if such a default linker option cannot be found.

TUTORIAL

ADDING AND DELETING DEFAULT LINKER OPTIONS The default linker options for for a project links several psects in the following fashion.
`-pone=600h,two,three`
which links `one` at 600h, then follows this with `two`, then `three`. It has been decided that the psects should be linked so that `one` follows `two`, which follows `three`, and that the highest address of `one` should be located at 5FFh. This new arragement can be specified issuing the following driver option:
`-L-pthree=-600h,two,one`
which creates passes the required linker options to the linker. The existing default option is still present, so this must be removed by use the driver option:
`-L-pone=`
which will remove the existing option.

The default option that you are deleting or replacing must contain an *equal* character.

2.6.8 `-Mfile`: Generate Map File

The `-M` option is used to request the generation of a map file. The map is generated by the linker an includes detailed information about where objects are located in memory, see Section 5.10 for information regarding the content of map files.

If no filename is specified with the option, then the name of the map file will have the project name, with the extension `.map`.

This option is on by default when compilingin under MPLAB IDE and using the HI-TECH Universal Toolsuite.

2.6.9 `-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.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.10 `-Ofile`: Specify Output File

This option allows the basename 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`. This will then also specify the output directory for any files produced by the linker or subsequently run applications. Any relative paths specified are with respect to the current working directory.

Any extension supplied with the filename will be ignored. The name and path specified by the `-O` option will apply to all output files.

The options that specify MAP file creation (`-M`, see 2.6.8), and SYM file creation (`-G`, see 2.6.4) override any name or path information provided by `-O` relevant to the MAP and SYM file.

To change the directory in which all output and intermediate files are written, use the `--OUTDIR` option, see Section 2.6.45. Note that if `-O` specifies a path which is inconsistent with the path specified in the `--OUTDIR` option, this will result in an error.

2.6.11 `-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.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.12 -Q: Quiet Mode

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

2.6.13 -S: Compile to Assembler Code

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

```
PICC --CHIP=16F877A -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.

2.6.14 -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.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.15 -V: Verbose Compile

The -V is the *verbose* option. The compiler will display the command lines used to invoke each of the compiler applications or compiler passes. Displayed will be the name of the compiler application being executed, plus all the command-line arguments to this application. This option may be useful for determining the exact linker options if you need to directly invoke the HLINK command.

If this option is used twice, it will display the full path to each compiler application as well as the full command line arguments. This would be useful to ensure that the correct compiler installation is being executed if there is more than one installed.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.16 -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.

2.6.17 --APBANK=*bank*: Specify Bank for Autos and Parameters

The auto and parameter variables of all functions are grouped into a compiled stack – see 5.9. The location of the compiled stack is automatically determined by the compiler, but use of this option can override the bank in which these objects are stored. The option is passed a number which corresponds to the bank number of desired location. A specific address cannot be specified with this option, only the bank number.

2.6.18 --ASMLIST: Generate Assembler .LST Files

The --ASMLIST option tells PICC to generate one or more *assembler listing file* for the C and assembly source modules being compiled. PRO compilers produce one assembly list file for the entire C program, including C library functions. All the output from every C source file will be continaed in the one assemby list file. One assembly list file is produce for each assembly source file in the project, including the runtime start up code (see Section 2.3.2).

In the case of C source code, the list file shows both the original C code and the corresponding assembly code generated by the code generator. For both C and assembly source code, a line number, the binary op-codes and addresses are shown. If the assembler optimizer is enabled (default operation) the list file may differ from the original assembly source code if the asmfile suboption is specified to the --OPT option. See Section 2.6.44 for more information.

The assembler optimizer may also simpify some expressions and remove some assembler directives from the listing file for clarity, although they are processed in the usual way in the assembly intermediate file. If you are using the assembly list file to look at the code produced by the compiler, you may which to turn off the assembler optimizer so that all the compiler-generated directives are shown in this file. Re-enable the optimizer when continuing developement. See Section 2.6.44 for more information.

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.

This option is on by default when compilingin under MPLAB IDE and using the HI-TECH Universal Toolsuite.

Table 2.5: Compiler responses to bank qualifiers

Selection	Response
`ignore`	bank qualifiers are ignored
`request`	attempt to locate variable according to bank qualifier, if bank unavailable try elsewhere
`require`	attempt to locate variable according to bank qualifier, if bank unavailable produce an error
`reject`	bank qualifiers are not allowed and will result in an error if seen

2.6.19 `--ADDRQUAL=`*`selection`*: Set Compiler Response to Bank Selection Qualifiers

The `--ADDRQUAL` option selects the compiler's response to a bank qualifier in source and can be used to provide functional compatibility with PICC STD compiler or to request that a particualr variable be positioned in a specific RAM bank. The selections are detailed in Table 2.5. By default, the compiler will ignore all bank qualifiers.

2.6.20 `--CALLGRAPH=type`: Select call graph type

This option allows control over the type of callgraph produced in the map file. Allowable suboption include: `none`, to specify that no callgraph should be produced; and `full` to indicate that the full callgraph 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 callgraph.

See also Sections

See Section 2.7 for use of this option in MPLAB IDE.

2.6.21 `--CHECKSUM=`*`start-end@destination<,specs>`*: Calculate a checksum

This option will perform a checksum over the address range specified and store the result at the destination address specified. Additional specifications can be appended as a comma separated list to this option. Such specifications are:

,width=*n* select the byte-width of the checksum result. A negative width will store the result in little-endian byte order. Result widths from one to four bytes are permitted.

,offset=*nnnn* An initial value or offset to be added to this checksum.

,algorithm=*n* Select one of the checksum algorithms implemented in hexmate. The selectable algorithms are described in Table 5.10.

Table 2.6: Default values for filling unprogrammed code space

Architecture	Default value
Baseline PIC	FFFh
Midrange PIC	3FFFh
High-end PIC	FFFFh

The *start*, *end* and *destination* attributes can be entered as word addresses as this is the native format for PICC program space. If an accompanying --FILL option has not been specified, unused locations within the specified address range will be filled with a default value for the selected device based on the values in table 2.6. This is to remove any unknown values from the equation and ensure the accuracy of the checksum result.

This option can be used to specify the target processor for the compilation.

To see a list of supported processors that can be used with this option, use the --CHIPINFO option described in Section 2.6.23.

See also Section 4.3.9.25 for information on setting the target processor from within assembly files.

The full list of supported devices is also included in Appendix C of this manual.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.22 --CHIP=*processor*: Define Processor

This option can be used to specify the target processor for the compilation.

To see a list of supported processors that can be used with this option, use the --CHIPINFO option described in Section 2.6.23.

See also Section 4.3.9.25 for information on setting the target processor from within assembly files.

2.6.23 --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.

2.6.24 --CODEOFFSET: Offset Program Code to Address

In some circumstances, such as bootloaders, it is necessary to shift the program image to an alternative 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.

Table 2.7: Selectable debuggers

Suboption	Debugger selected
none	No debugger
icd or icd1	MPLAB ICD
icd2	MPLAB ICD2
pickit2	MPLAB PICKit2

See Section 2.7 for use of this option in MPLAB IDE.

2.6.25 --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, PICC will invoke CREF to process the cross reference information into the listing file, in this case test.crf. If multiple source files are to be included in the cross reference listing, all must be compiled and linked with the one PICC command. For example, to generate a cross reference listing which includes the source modules main.c, module1.c and nvram.c, compile and link using the command:

```
PICC --CHIP=16F877A --CR=main.crf main.c module1.c nvram.c
```

Thus this option can not be used when using any compilation process that compiles each source file separately using the -C or --PASS1 options. Such is the case for most IDEs, including HI-TIDE, and makefiles.

2.6.26 --DEBUGGER=*type*: Select Debugger Type

This option is intended for use for compatibility with debuggers. PICCsupports the Microchip ICD2 debugger and using this option will configure the compiler to conform to the requirements of the ICD2 (reserving memory addresses, etc.). For example:

```
PICC --CHIP=16F877A --DEBUGGER=icd2 main.c
```

The possible selections for this option are defined in Table 2.7.
See Section 2.7 for use of this option in MPLAB IDE.

2.6.27 --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.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.28 --ECHO: Echo command line before processing

Use of this option will result in the command line being echoed to the `stderr` stream before compilation is commenced. Each token of the command line will be printed on a separate line and will appear in the order in which they are placed on the command line.

2.6.29 --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. 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 option allows the exact format of printed error messages to be specified using special place-holders embedded within a message template. See Section 2.5 for full details of the messaging system employed by PICC.

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

See Section 2.6.36 for the appropriate option to change the message language.

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

This option sets the maximum number of errors each compiler application, as well as the driver, will display before stopping. By default, up to 20 error messages will be displayed. See Section 2.5 for full details of the messaging system employed by PICC.

2.6.31 --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.

See Section 2.7 for use of this option in MPLAB IDE.

Table 2.8: Floating point selections

Suboption	Effect
`double`	Size of float matches size of double type
`24`	24 bit float
`32`	32 bit float (IEEE754)
`fast32`	32 bit with accelerated library routines

Table 2.9: Supported IDEs

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

2.6.32 `--FLOAT=`*`type`*: Select kind of Float Types

This option allows the size of `float` types to be selected. The types available to be selected are given in Table 2.8. See also the `--double` option in Section 2.6.27.

2.6.33 `--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.

2.6.34 `--HELP<=`*`option`*`>`: Display Help

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

```
PICC --help=warn
```

This will display more detailed information about the `--WARN` option, the available suboptions, and which suboptions are enabled by default.

2.6.35 `--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 2.9.

Table 2.10: Supported languages

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

2.6.36 `--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 2.10 shows those langauges currently supported.

See Section 2.5 for full details of the messaging system employed by PICC.

2.6.37 `--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.

2.6.38 `--MODE=`*`operatingmode`*: Choose Compiler Operating Mode

This option selects the basic operating mode of the compiler. The available types are `pro` and `lite`. A compiler operating in `pro` mode employs Omniscient Code Generation™ and allows full optimization to be enabled. A compiler operating in lite mode will only allow limited optimizations to be enabled, and thus the code size will be larger. Only those modes permitted by the compiler license status will be accepted.

See also Section 2.6.44 on enabling specific optimizations.

2.6.39 `--MSGDISABLE=`*`messagelist`*: Disable Warning Messages

This option allows warning or advisory messages to be disabled during compilation of all modules within the project, and during all stages of compilation. Warning mesasges can also be disabled using pragma directives. For full information on the compiler's messaging system, see Section 2.5.

The `messagelist` is a comma-separated list of warning numbers that are to be disabled. If the number of an error is specified, it will be ignored by this option. If the message list is specified as `0`, then all warnings are disabled.

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

This option sets the format of advisory messages produced by the compiler. See Section 2.5 for full information.

2.6.41 --NODEL: Do not remove temporary files

Specifying --NODEL when building will instruct PICCnot to remove the intermediate and temporary files that were created during the build process.

2.6.42 --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.

2.6.43 --OBJDIR: Specify a directory for intermediate files

This option allows a directory to be nominated in for PICCto locate its intermediate files. If this option is omitted, intermediate files will be created in the current working directory. This option will not set the location of output files, instead use --OUTDIR. See 2.6.45 and 2.6.10 for more information.

2.6.44 --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 some assembler optimizations. Table 2.11 lists the available optimization types. The optimizations that are controlled through specifying a level 1 through 9 affect optimization during the code generation stage. The level selected is commonly referred to as the *global optimization level*.

2.6.45 --OUTDIR: Specify a directory for output files

This option allows a directory to be nominated in for PICCto locate its output files. If this option is omitted, output files will be created in the current working directory. This option will not set the location of intermediate files, instead use --OBJDIR. See 2.6.43 and 2.6.10 for more information.

Table 2.11: Optimization Options

Option name	Funcion
`1..9`	Select global optimization level (1 through 9)
`asm`	Select optimizations of assembly derived from C source
`asmfile`	Select optimizations of assembly source files
`debug`	Favor accurate debugging over optimization
`all`	Enable all compiler optimizations
`none`	Do not use any compiler optimziations

Table 2.12: Output file formats

Type tag	File format
`lib`	Library File
`lpp`	P-code library
`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

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

This option allows the type of the output file(s) 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 format are shown in Table 2.12. More than one output format may be specified by supplying a comma-separated list of tags. Those output file types which specify library formats stop the compilation process before the final stages of compilation are executed. Hence specifying an output file format list containing, e.g. `lib` or `all` will over-ride the non-library output types, and only the library file will be created.

2.6.47 --PASS1: Compile to P-code

The --PASS1 option is used to generate a p-code intermediate files (.p1 file) from the parser, then stop compilation. Such a file needs to be generated if creating a p-code library file.

2.6.48 --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.

If you wish to see the preprocessed source for the printf family of functions, do *not* use this option. The source for this function is customised by the compiler, but only after the code generator has scanned the project for printf usage. Thus, as the –PRE option stops compilation after the preprocessor stage, the code generator will not execute and no printf code will be processed. If this option is omitted, the preprocessed source for printf will be retained in the file doprnt.pre.

If you wish to see the preprocessed source for the printf family of functions, do *not* use this option. The source for this function is customised by the compiler, but only after the code generator has scanned the project for printf usage. Thus, as the –PRE option stops compilation after the preprocessor stage, the code generator will not execute and no printf code will be processed. If this option is omitted, the preprocessed source for printf will be retained in the file doprnt.pre.

2.6.49 --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:

```
PICC --CHIP=16F877A --PROTO test.c
```

PICC will produce `test.pro` containing the following declarations which may then be edited as necessary:

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

2.6.50 `--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.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.51 `--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.

See Section 2.7 for use of this option in MPLAB IDE.

Table 2.13: 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.
`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.
`no_startup`	Whether the startup module is linked in with user-defined code.	The start-up module is generated and linked into the program.
`stackwarn`	Checking the depth of the stack used.	The stack depth is monitored at compile time and a warning will be produced if a potential stack overflow is detected.
`stackcall`	Allow function calls to use the hardware stack	Functions called via `call` instruction while stack not exhausted.

2.6.52 --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 2.13.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.53 --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.

2.6.54 --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 5.15.1.15 of this manual.

The driver will also define a label at this location which can be referenced from the C context as __serial0. For code to access this symbol, remember to declare it specifying what data type to use. For example:

```
extern const int __serial0;
```

See Section 2.7 for use of this option in MPLAB IDE.

2.6.55 --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. The *app* component specifies the application that will recieve the new options. The *file* component specifies the name of the file that contains the additional options that will be passed to

the application. This option is not required for most projects. If specifying more than one option to a component, each option must be entered on a new line in the option file.
This option can also be used to remove an application from the build sequence. If the *file* parameter is specified as *off*, execution of the named application will be skipped. In most cases this is not desirable as almost all applications are critical to the success of the build process. Disabling a critical application will result in catastrophic failure. However it is permissible to skip a non-critical application such as *clist* or *hexmate* if the final results are not reliant on their function.

2.6.56 --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>).

2.6.57 --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 total memory usage for all memory spaces.

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 2.14 shows what summary types are available.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.58 --TIME: Report time taken for each phase of build process

Adding --TIME when building generate a summary which shows how much time each stage of the build process took to complete.

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

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

Table 2.14: 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 concise summary of memory used will be shown.
class	Summary of class usage.	A summary of all classes in each memory space 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.

2.6.60 --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 higher the warning level, the more important the warning message. The default warning level is 0 and will allow all normal warning messages.

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

Warning message can be individually disabled with the --MSGDISABLE option, see 2.6.39. See also Section 2.5 for full information on the compiler's messaging system.

See Section 2.7 for use of this option in MPLAB IDE.

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

This option sets the format of warning messages produced by the compiler. See Section 2.6.29 for more information on this option. For full information on the compiler's messaging system, see Section 2.5.

2.7 MPLAB Universal Toolsuite Equivalents

When compiling under the MPLAB IDE, it is still the compiler's command-line driver that is being executed and compiling the program. The HI-TECH Universal Toolsuite plugin controls the MPLAB IDE build options dialog that is used to access the compiler options, however these graphical controls ultimately adjust the command-line options passed to the command-line driver when compiling. You can see the command-line options being used when building in MPLAB IDE in the Output window.

The following dialogs and descriptions identify the mapping between the dialog controls and command-line options. As the toolsuite is universal across all HI-TECH compilers supporting Microchip, not all options are applicable for HI-TECH C PRO for the PIC10/12/16 MCU Family.

2.7.1 Directories Tab

The options in this dialog control the output and search directories for some files. See Figure 2.3 in conjunction with the following command line option equivalents.

1. Output directory
   This selection uses the buttons and fields grouped in the bracket to specify an output directory for files output by the compiler.

2. Include Search path
   This selection uses the buttons and fields grouped in the bracket to specify include (header) file search directories. See 2.6.5.

2.7.2 Compiler Tab

The options in this dialog control the aspects of compilation up to code generation. See Figure 2.4 in conjunction with the following command line option equivalents.

1. Define macros
   The buttons and fields grouped in the bracket can be used to define preprocessor macros. See 2.6.2.

2. Undefine macros
   The buttons and fields grouped in the bracket can be used to undefine preprocessor macros. See 2.6.14.

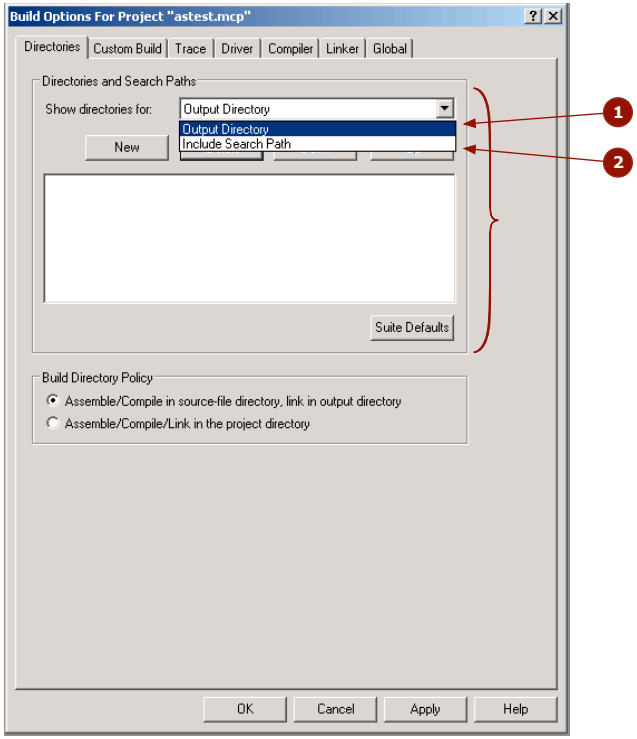


Figure 2.3: The Directories dialog

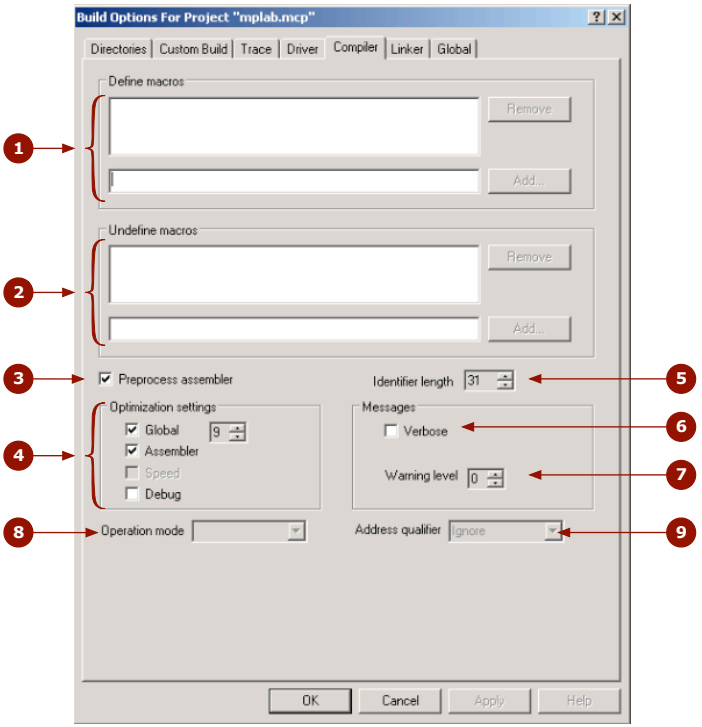


Figure 2.4: The Compiler dialog

3. Preprocess assembly
   This checkbox controls whether assembly source files are scanned by the preprocessor. See 2.6.11.

4. Optimization settings
   These controls are used to adjust the different optimizations the compiler employs. See 2.6.44.

5. Identifier length
   This selector controls the maximum identifier length in C source. See 2.6.9.

6. Verbose
   This checkbox controls whether the full command-lines for the compiler applications are displayed when building. See 2.6.15.

7. Warning level
   This selector allows the warning level print threshold to be set. See 2.6.60.

8. Operation Mode
   This selector allows the user to force another available operating mode (e.g. Lite, Standard or PRO) other then the default. See 2.6.38

9. Address Qualifier
   This selector allows the user to select the behaviour of the address qualifier. See 2.6.19

2.7.3 Linker Tab

The options in this dialog control the link step of compilation. See Figure 2.5 in conjunction with the following command line option equivalents.

1. Runtime options
   These checkboxes control the many runtime features the compiler can employ. See 2.6.52.

2. Fill
   This field allows a fill value to be specified for unused memory locations. See 2.6.31.

3. Codeoffset
   This field allows an offset for the program to be specified. See 2.6.24.

4. Checksum
   This field allows the checksum specification to be specified. See 2.6.21.

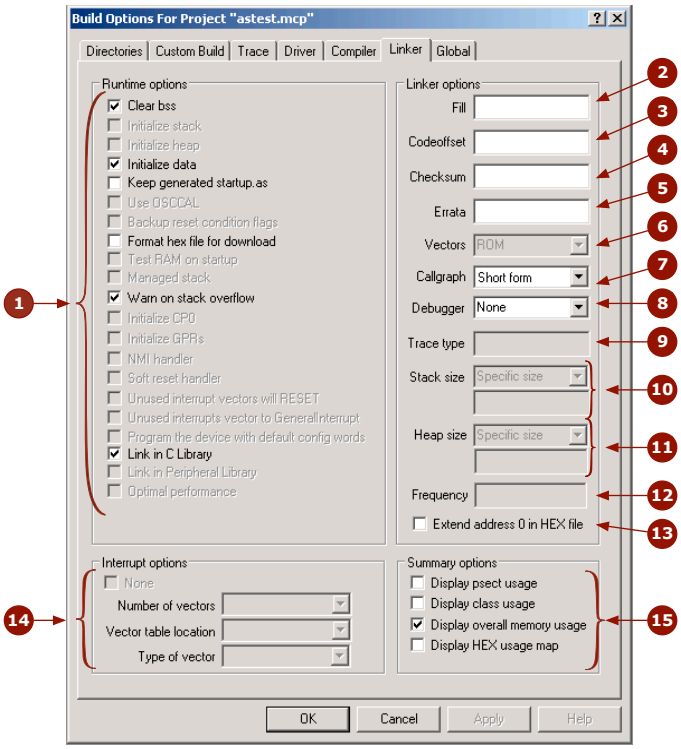


Figure 2.5: The Linker dialog

5. Errata
   This field allows the errata workarounds employed by the compiler to be controlled. See **??**.
6. Vectors
   Not applicable.
7. Callgraph
   This selector allows the type of call graph printed in the map file to be chosen. See 5.10.2.2.
8. Debugger
   This selector allows the type of hardware debugger to be chosen. See 2.6.26.
9. Trace type
   Not yet implemented.
10. Stack size
    Not applicable.
11. Heap size
    Not applicable.
12. Frequency
    Not applicable.
13. Extend address 0 in HEX file
    This option specifies that the intel HEX file should have initialization to zero of the upper address . See 2.6.46.
14. Interrupt options
    Not applicable.
15. Summary Options
    These checkboxes control which summaries are printed after compilation. See 2.6.57.

2.7.4 Global Tab

The options in this dialog control aspects of compilation that are applicable throughout code generation and link steps. See Figure 2.6 in conjunction with the following command line option equivalents.

1. Memory model
   Not applicable.

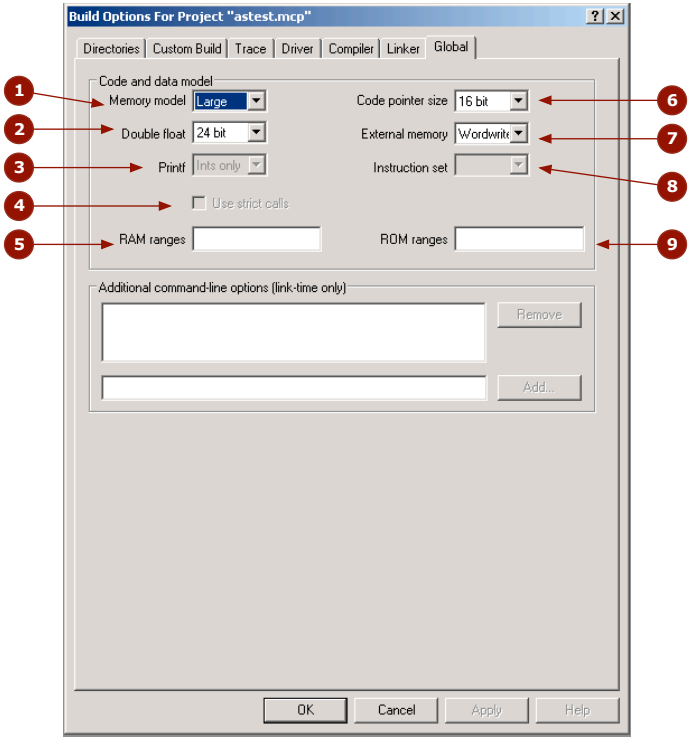


Figure 2.6: The Global dialog

2. Double float
   This selector allows the size of double float objects to be selected. See 2.6.27.

3. Printf
   Not applicable.

4. Use strict calls
   Not applicable.

5. RAM ranges
   This field allows the default RAM (data space) memory used to be adjusted. See 2.6.50.

6. Code pointer size
   Not applicable.

7. External memory
   This selector allows the type of external memory access to be specified. See **??**.

8. Instruction set
   Not applicable.

9. ROM ranges
   This field allows the default ROM (program space) memory used to be adjusted. See 2.6.51.

Chapter 3

C Language Features

HI-TECH C PRO for the PIC10/12/16 MCU Family 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 these devices.

3.1 ANSI Standard Issues

3.1.1 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 C compiler behaves in such situations.

3.2 Processor-related Features

HI-TECH C has several features which relate directly to the PIC architecture and instruction set. These detailed in the following sections.

3.2.1 Stack

The stack on PIC processors is limited in depth and cannot be manipulated directly. It is left up to the programmer to ensure that the maximum stack dept is not exceeded. A call graph is provided by the linker when generating a MAP file. This will indicate the stack levels at each function call and

can be used as a guide to stack depth. The compiler may also produce warnings if the maximum stack depth is reached.

3.2.2 Configuration Fuses

The PIC processor's configuration fuses (or configuration bits) may be set using the __CONFIG macro as follows:

```
__CONFIG(x);
```

Note there are two leading *underscore* characters and x is the value that is to be in the configuration word. The macro is defined in <htc.h>, so be sure to include this into the module that uses this macro.

Specially named quantities are defined in the header file appropriate for the processor you are using to help you set the required features. These names usually follow the same names as used in the datasheet. Refer to your processor's header file for details. For devices that have more than one configuration word, each subsequent invocation of __CONFIG will modify the next configuration word in sequence. Typically this might look like:

```
#include <htc.h>
__CONFIG(WDTDIS & XT & UNPROTECT);  // Program config. word 1
__CONFIG(FCMEN);   // Program config. word 2
```

3.2.3 ID Locations

Some PIC 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(15F0);
```

will attempt to fill ID locations which the values: 1, 5, F, 0. 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. Some devices will permit programming up to seven bits within each ID location. To program the full seven bits, the regular __IDLOC macro is not suitable. For this situation the __IDLOC7(a,b,c,d) macro is available. The parameters a to d are a comma separated list of values. The values can be entered as either decimal or hexadecimal format such as:

```
__IDLOC7(0x7f,1,70,0x5a);
```

It is not appropriate to use the __IDLOC7 macro on a device that does not permit seven bit programming of ID locations.

3.2.4 Bit Instructions

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

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

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

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

3.2.5 EEPROM Access

For most devices that come with on-chip EEPROM, the compiler offers several methods of accessing this memory. The EEPROM access methods are described in the following sections.

3.2.5.1 The __EEPROM_DATA() macro

For those PIC devices that support external programming of their EEPROM data area, the __EEPROM_DATA() macro can be used to place the initial 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, it is to be used for preloading EEPROM contents at program time only.

3.2.5.2 EEPROM Access Functions

The library functions `eeprom_read()` and `eeprom_write()`, 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 and retrieve it using these functions would be:

```
#include <htc.h>
void eetest(void){
   unsigned char value = 1;
   unsigned char address = 0;
   // write value to EEPROM address
   eeprom_write(address,value);
   // read from EE at address
   value = eeprom_read(address);
}
```

These functions test and wait for any concurrent writes to EEPROM to conclude before performing their required operation. The `eeprom_write()` function will initiate the process of writing to EEPROM and this process will not have completed by the time that `eeprom_write()` returns. The new data written to EEPROM will become valid approximately four milliseconds later. In the above example, the new value will not yet be ready at the time when `eeprom_read()` is called, however because this function waits for any concurrent writes to complete before initiating the read, the correct value will be read.

It may also be convenient to use the preprocessor symbol, `_EEPROMSIZE` in conjunction with some of these access methods. This symbol defines the number of EEPROM bytes available for the selected chip.

3.2.5.3 EEPROM Access Macros

Although these macros perform much the same service as their library function counterparts, these should only be employed in specific circumstances. It is appropriate to select `EEPROM_READ` or `EEPROM_WRITE` in favour of the library equivalents if any of the following conditions are true:

- You cannot afford the extra level of stack depth required to make a function call
- You cannot afford the added code overhead to pass parameters and perform a call/return
- You cannot afford the added processor cycles to execute the function call overhead

Be aware that if a program contains multiple instances of either macro, any code space saving will be negated as the full content of the macro is now duplicated in code space.

In the case of `EEPROM_READ()`, there is another very important detail to note. Unlike `eeprom_read()`, this macro does not wait for any concurrent EEPROM writes to complete before proceeding to select and read EEPROM. Had the previous example used the `EEPROM_READ()` macro in place of `eeprom_read()` the operation would have failed. If it cannot be guaranteed that all writes to EEPROM have completed at the time of calling `EEPROM_READ()`, the appropriate flag should be polled prior to executing `EEPROM_READ()`. For example:

```
#include <htc.h>
void eetest(void){
   unsigned char value = 1;
   unsigned char address = 0;
   // Initiate writing value to address
   EEPROM_WRITE(address,value);
   // wait for end-of-write before EEPROM_READ
   while(WR) continue;
   // read from EEPROM at address
   value = EEPROM_READ(address);
}
```

3.2.6 Flash Runtime Access

HI-TECH C PRO for the PIC10/12/16 MCU Family provides a number of methods to access the contents of program memory at runtime. Particular care must be taken when modifying the contents of program memory. If the location being modified is that which is currently being executed or you've accidentally selected a region of your executable code for use as non-volatile storage, the result could be disastrous so take care.

For those devices requiring a flash erasure operation be performed prior to writing to flash, this step will be performed internally by the compiler within the access routine and does not need to be implemented as a separate stage. Data within the same flash erasure block that is unrelated to the write operation will be backed up before the block is erased and restored after the erasure.

3.2.6.1 Flash Access Macros

Similar to the EEPROM read/write routines described above, there are equivalent Flash memory routines. For example, to write a byte-sized value to an address in flash memory:

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

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

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

3.2.6.2 Flash Access Functions

The `flash_read()` function provides the same functionality as the `FLASH_READ()` macro but will potentially cost less in code space if multiple invocations are required.

The `flash_copy()` function allows duplication of a block of memory at a location in flash memory. The block of data being duplicated can be sourced from either RAM or program memory. This routine is only available for those devices which support writing to flash memory in sizes greater than one word at a time.

For the small subset of devices which allow independent control over a flash block erasure process, the `flash_erase()` function provides this service if required.

3.2.7 Baseline PIC special instructions

The PIC baseline (12-bit instruction word) devices have some registers which are not in the normal SFR area and cannot be accessed using an ordinary move instruction. The HI-TECH C compiler can be instructed to automatically use the special instructions intended for such cases when pre-defined symbols are accessed.

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

3.2.7.1 The OPTION instruction

Some baseline PIC devices use an option instruction to load the OPTION register. The appropriate header files contain a special definition for a C object called OPTION and macros for the bit symbols which are stored in this register. HI-TECH C PRO for the PIC10/12/16 MCU Family will automatically use the OPTION instruction when an appropriate processor is selected and the OPTION register is accessed.

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

```
OPTION = PSA;
```

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

3.2.7.2 The TRIS instructions

Some PIC devices use a TRIS instruction to load the TRIS register. The appropriate header files contain a special definition for a C object called TRIS. HI-TECH C PRO for the PIC10/12/16 MCU Family will automatically use the TRIS instruction when an appropriate processor is selected and the TRIS register is accessed.

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

```
TRIS = 0xFF;
```

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

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

3.2.7.3 Calibration Space

The Microchip-modified IEEE754 32-bit floating point format parameters in the calibration space in the PIC14000 processor may be accessed using the get_cal_data() function. The byte parameters may be accessed directly using the identifiers defined in the header file.

3.2.7.4 Oscillator calibration constants

Some PIC devices come with an oscillator calibration constant which is pre-programmed into the devices program memory. This constant can be read and written to the OSCCAL register to calibrate the internal RC oscillator. On some baseline PIC devices the calibration constant is stored as a

movlw instruction at the top of program memory, e.g. the 12C50X and 16C505 parts. On reset the program counter is made to point to this instruction and it is executed first before the program counter wraps around to 0x0000 which is the effective reset vector for the device. The HI-TECH C PRO for the PIC10/12/16 MCU Family compiler default startup routine will automatically include code to load the OSCCAL register with the value contained in the W register after reset on such devices. No other code is required by the programmer.

For other chips, such as 12C67X chips, the oscillator constant is also stored at the top of program memory, but as a retlw instruction. The compiler's startup code will automatically generate code to retrieve this value and do the configuration. This feature can be turned off via the --RUNTIME option.

At runtime this value may be read using the macro _READ_OSCCAL_DATA(). To be able to use this macro, make sure that <htc.h> is included into the relevant modules of your program. This macro returns the calibration constant which can then be stored into the OSCCAL register, as follows:

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

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

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

3.3 Supported Data Types and Variables

The HI-TECH C PRO for the PIC10/12/16 MCU Family compiler supports basic data types with 1, 2, 3 and 4 byte sizes. Table 3.1 shows the data types and their corresponding size and arithmetic type.

3.3.1 Radix Specifiers and Constants

The format of integral constants specifies their radix. HI-TECH C supports the ANSI standard radix specifiers as well as ones which enables binary constants to specified in C code. The format

Table 3.1: Basic data types

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

Table 3.2: 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

used to specify the radices are given in Table 3.2. 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 C`.

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"`.

3.3.2 Bit Data Types and Variables

HI-TECH C PRO for the PIC10/12/16 MCU Familysupports `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 functions prototype in the usual way. The bit return value will be returning in the carry flag in 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 declared pointers to `bit` variables or statically initialise `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 PICC flag `--STRICT` is used, the `bit` keyword becomes unavailable.

3.3.3 8-Bit Integer Data Types and Variables

HI-TECH C PRO for the PIC10/12/16 MCU Family 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`. 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 C the `char` types will commonly be used for a number of purposes, as 8-bit integers, as storage for ASCII characters, and for access to I/O locations.

3.3.4 16-Bit Integer Data Types

HI-TECH C PRO for the PIC10/12/16 MCU Family 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`.

3.3.5 24-Bit Integer Data Types

HI-TECH C PRO for the PIC10/12/16 MCU Family supports four 24-bit integer types. `short long` are 24-bit two's complement signed integer types, representing integral values from -8,388,608 to +8,388,607 inclusive. `Unsigned short` and `unsigned int` are 16-bit unsigned integer types, representing integral values from 0 to 16,777,215 inclusive. All 24-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 long int` and `unsigned short long int` keyword sequences, respectively, to hold values of these types. When specifying a `short long int` type, the keyword `int` may be omitted. Thus a variable declared as `short`

`long` will contain a `signed short long int` and a variable declared as `unsigned short long` will contain an `unsigned short long int`.

3.3.6 32-Bit Integer Data Types and Variables

HI-TECH C PRO for the PIC10/12/16 MCU Familysupports 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`.

3.3.7 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 3.3, 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 3.4 (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).

Table 3.3: 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 3.4: 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)	

The 32-bit example in 3.4 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`.

3.3.8 Structures and Unions

HI-TECH C PRO for the PIC10/12/16 MCU Family 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.

3.3.8.1 Bit-fields in Structures

HI-TECH C PRO for the PIC10/12/16 MCU Family 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};
```

3.3.8.2 Structure and Union Qualifiers

HI-TECH C 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};
```

3.3.9 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 C supports both ANSI qualifiers and additional special qualifiers which are useful for embedded applications and which take advantage of the PIC architecture.

3.3.9.1 Const and Volatile Type Qualifiers

HI-TECH C supports the use of the ANSI type qualifiers `const` and `volatile`.

The `const` type qualifier is used to tell the compiler that an object 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 a 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 unsigned int  TACTL @ 0x160;
```

Volatile objects may be accessed using different generated code to non-`volatile` objects.

3.3.10 Special Type Qualifiers

HI-TECH C PRO for the PIC10/12/16 MCU Familysupports the special type qualifiers to allow the user to control placement of `static` and `extern` class variables into particular address spaces.

3.3.10.1 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 or even power cycles (on-off-on).

The `persistent` type qualifier is used to qualify variables that should not be cleared on startup. In addition, any `persistent` variables will be stored in a different area of memory to other variables. `persistent` objects are placed within the psect `nvram`.

This type qualifier may not be used on variables of class `auto`; if used on variables local to a function they must be combined with the `static` 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;

If the `PICC` option, `--STRICT` is used, this type qualifier is changed to `__persistent`.

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

3.3.10.2 Near Type Qualifier

The `near` type qualifier is a recommendation to place static variables in the *common memory* of the PIC MCU. `Near` objects are represented by 8 bit addresses and are always accessible 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 common memory:

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

The memory allocation scheme used by HI-TECH C PRO for the PIC10/12/16 MCU Family automatically allocates variables to the common memory. The common memory will be filled before banked memory is used. The allocation scheme uses the number of times the each object is referenced, as well the size of the object to determine which are given preference to the common memory. The `near` qualifier increases the probability of a variable being placed in the common memory.

3.3.10.3 Bank0, Bank1, Bank2 and Bank3 Type Qualifiers

The `bank0`, `bank1`, `bank2` and `bank3` type qualifiers are always recognised by HI-TECH C PRO for the PIC10/12/16 MCU Family so that code may be easily ported from other compilers, however by default, these qualifiers have no effect.

If the `--ADDRQUAL` option is set to `request` (see Section 2.6.19), the `bank0`, `bank1`, `bank2` and `bank3` qualifiers become a recommendation to place non-auto variables in RAM bank 0, bank 1, bank2 and bank 3, respectively. If space is not available in these banks, another bank will be used.

If the `--ADDRQUAL` option is set to `require`, the bank qualifiers will always place non-auto objects in the bank specified. If no room exists in that bank, an error will be generated.

The following example of bank qualifier usage will either recommend or force (dependent on the `--ADDRQUAL` option settings) the `unsigned char` to be stored in `bank3`:

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

3.3.11 Pointer Types

There are two basic pointer types supported by HI-TECH C PRO for the PIC10/12/16 MCU Family: data pointers and function pointers. Data pointers hold the address of variables which can be read, and possible written, indirectly by the program. Function pointers hold the address of an executable routine which can be called indirectly via the pointer.

Typically qualifiers are used with pointer definitions to customise the scope of the pointer, allowing the code generator to set an appropriate size and format for the addresses the pointer will hold. Pro version compilers use sophisticated algorithms to track the assignment of addresses to data pointers, and, as a result, many of these qualifiers no longer need to be used, and the size of the pointer is optimal for the its intended usage.

It is helpful to first review the ANSI standard conventions for definitions of pointer types.

3.3.11.1 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 *target* that the pointer references, or to which the pointer points. The general form of a pointer definition looks like the following.

```
target_type_&_qualifiers * pointer's_qualifiers pointer's_name;
```

Any qualifiers to the right of the * (i.e. next to the pointer's name) relate to the pointer variable itself. The type and any qualifiers to the left of the * relate to the pointer's targets.

TUT•RIAL

EXAMPLE OF POINTER QUALIFIERS Here are three examples of pointer definitions using the volatile qualifier. The fields in the definitions have been highlighted with spacing:

```
volatile int *          vip ;
int          * volatile ivp ;
volatile int * volatile vivp ;
```

The first example is a pointer called `vip`. It contains the address of an `int` objects that are qualified `volatile`. The pointer itself — the variable that holds the address — is *not* `volatile`, however the objects that are accessed when the pointer is dereferenced is `volatile`. That is, the target objects accessible via the pointer may be externally modified.

The second example is a pointer called `ivp` which also contains the address of `int` objects. In this example, the pointer itself is `volatile`, that is, the address the pointer contains may be externally modified, however the objects that can be accessed when dereferencing the pointer are not `volatile`.

The last example is of a pointer called `vivp` which is itself qualified `volatile` and which also holds the address of a `volatile` object.

Bare in mind that one pointer can be assigned the address of many objects, for example a pointer that is a parameter to a function is assigned a new object address every time the function is called. The definition of the pointer must be valid for every target address assigned.

•

Care must be taken when describing pointers: Is a "const pointer" a pointer that points to `const` objects, or a pointer that is `const` itself. You can talk about "pointers to

const" and "const pointers" to help clarify the definition, but such terms may not be universally understood.

3.3.11.2 Data Pointers

HI-TECH C PRO for the PIC10/12/16 MCU Family monitors and records all assignments of addresses to each data pointer the program defines. The size and format of the address held by each pointer is based on this information. When more than one address is assigned to a pointer at different places in the code, a set of all possible targets the pointer can address is maintained. This information is specific to each pointer defined in the program, thus two pointers with the same type may hold addresses of different sizes and formats due to the different nature of objects they address in the program.

The following pointer classifications are currently implemented:

- An 8-bit pointer capable of accessing common memory and two consecutive banks, e.g. banks 0 and 1, or banks 7 and 8;
  - Address is an offset into the lower bank of the bank pair, e.g. bank 0 (extending into bank 1) or bank 7 (extending into bank 8)
- A 16-bit pointer capable of accessing the entire data memory space;
- An 8-bit pointer capable of accessing up to 256 bytes of program space data;
  - Address is an index into a lookup table;
- A 16-bit pointer capable of accessing up to 64 kbytes of program space data;
  - Address is an offset into psect `strings`;
- A 16-bit pointer capable of accessing the entire data space memory and up to 64 kbytes of program space data;
  - The most significant bit indicates the memory space of the target: 1 indicates an address of an object in the data space; 0 indicates the address of an object in the program space;

Each data pointer will be allocated one of the above classifications after preliminary scans of the source code. There is no mechanism by which the programmer can specify the style of pointer required (other than by the address assignments to the pointer).

TUTORIAL

DYNAMIC POINTER SIZES A program in the early stages of development contains the following code;

```
void main(void) {
int i, *ip;
ip = &i;
}
```

The code generator allocates the variable `i` to bank 0. The code generator notes that the pointer `ip` only points to an object in one memory bank, so this pointer is made an 8-bit wide data pointer.

As the program is developed, other variables are defined and allocated space in the other memory banks. The pointer, `ip`, is also assigned the address of another object that has been placed in bank 2. When the program is next compiled, the pointer `ip` will automatically become a 16-bit pointer to all of the data space, and the code used to initialize and dereference the pointer will change accordingly. This takes place without any modification to the source code.

One positive aspect of tracking pointer targets is less of a dependence on pointer qualifiers. The standard qualifiers `const` and `volatile` must still be used in pointer definitions to indicate a read-only or externally-modifiable target object, respectively. However this is in strict accordance with the ANSI standard. HI-TECH specific qualifiers, like `near` and `far`, do not need to be used to indicate pointer targets, and should be avoided. Omitting these qualifiers will result in more portable and readable code, and lessen the chance of extraneous warnings being issued by the compiler.

3.3.11.3 Pointers to Const

The `const` qualifier plays no direct part in specifying the pointer classification that the compiler will allocate to a pointer. This qualifier should be only used when the target, or targets, referenced by the pointer should be read-only. The addresses of `const` objects assigned to a pointer will result in that pointer having a classification capable of accessing the program space. The exact classification will also depend on other factors.

The code generator tracks the total size of `const` qualified variables that are defined. It uses this information to determine how large any pointers that can access `const` objects must be. Such pointers may be either 1 or 2 bytes wide.

TUTORIAL

POINTERS AND CONST DATA Assume a program contains of the following:

```
void main(void) {
const char in_table[20] = { /* values */ };
char * cp;
cp = &in_table;
}
```

If the array above is the only `const` data in the program, then there are 20 bytes of `const` data used in the program. In this instance, the code generator will make the pointer, `cp`, a one byte wide pointer to objects in the program space.

Later, the program is changed and another `const` array is added to the code:

```
const char out_table[240] = { /* values */ };
```

As the total size of `const` data for this program now exceeds 255 bytes, the size of *any* pointer that can access `const` objects will be made 2 bytes long. Even if the pointer, `cp`, is not assigned the address of this new array, `out_table`, its size will increase.

3.3.11.4 Pointers to Both Memory Spaces

When a pointer is assigned the address of one or more objects allocated memory in the data space, and also assigned the address of one or more `const` objects, the pointer will be classified such that it can dereference both memory spaces, and the address will be encoded so that the target memory space can be determined at runtime.

A 16-bit mixed space pointer is encoded such that if it holds an address that is higher than the highest general purpose RAM address, it holds the address of a program space object; all other address reference objects in the data space.

TUTORIAL

POINTERS TO DIFFERENT TARGETS A program in the early stages of development contains the following code;

```
int getValue(int * ip) {
return 2 + *ip ;
}
void main(void) {
int j, i = setV();
j = getValue(&i)
}
```

The code generator allocate the variable `i` to the access bank and the pointer `ip` (the parameter to the function `getValue`) is made an 8-bit wide access bank pointer. At a later date, the function `main` is changed, becoming:

```
void main(void) {
int j, i = setV();
const int start = 0x10;
j = getValue(&i)
j += getValue(&start);
}
```

Now the pointer, `ip`, is assigned addresses of both data and `const` objects. After the next compilation the size and encoding of `ip` will change, as will the code that assigns the addresses to `ip`. The generated code that dereferences `ip` (in `getValue`) will check the address to determine the memory space of the target address.

3.4 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.

3.4.1 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 stack frame, and `static` variables which are always given a fixed memory location and have permanent duration.

3.4.1.1 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 off the symbol that represents that block. The variables will not necessarily be allocated in the order declared - in contrast to parameters which are always in lexical order. Note that most type qualifiers cannot be used with `auto` variables, since there is no control over the storage location. The exceptions are `const` and `volatile`.

All `auto` variables are allocated memory wholly within one bank. The bank qualifiers cannot be used with objects of type `auto`.

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 `??_main` and `??_main+1` since an int is two bytes long.

Note that standard version compilers use the prefix `?a` instead of `??` However the allocation and access of auto variables is otherwise the same.

3.4.1.2 Static Variables

Uninitialized static variables are allocated a fixed memory location which will not be overlapped by storage for other functions. Static variables are local in scope to the function in which they are declared, but may be accessed by other functions via pointers since they have permanent duration. Static variables are guaranteed to retain their value between calls to a function, unless explicitly modified via a pointer. Static variables are not subject to any architectural limitations on the PIC.

Static variables which are initialised are only done so once during the programs 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.

3.4.2 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 @ 0x06;
```

will declare a variable called `Portvar` located at 06h. The compiler will reserve storage for this object via the assembler `DABS` directive, and will equate the variable to that address. The compiler-generated assembler will include a line of the form:

```
_Portvar EQU 06h
```

This construct is primarily intended for equating the address of a C identifier with a microprocessor special function register, but can be used to place user-defined variables at an absolute address. The compiler and linker do not make any checks for overlap of absolute variables with other absolute variables.

Defining absolute objects can fragment memory and may make it impossible for the linker to position other objects. Avoid absolute objects if at all possible. If absolute object s must be defined, try to place them at one end of a memory bank.

3.4.3 Objects in Program Space

`Const` objects are usually placed in program space. On most PIC devices, the program space is word-addressable but not directly readable by the device — the compiler stores one character per location encapsulated in a `retlw` instruction which can be called, and which will return with the encoded value in `W`. This is also true with Enhanced Mid-range PIC devices, although the dereference method is different.

All `const`-qualified data objects and string literals are placed in the `strings` psect. See also Section 3.3.11.

3.5 Functions

3.5.1 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.

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
movwf ?_test
clrf  ?_test+1
movf  _a,w
call (_test)
```

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

If you need to determine, for assembly 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, compile this, and then inspect the assembly list file (PICC --ASMLIST option), allowing you to examine the assembly code.

3.5.2 Function Return Values

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

3.5.2.1 8-Bit Return Values

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

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

will exit with the following code:

```
retlw 0
```

3.5.2.2 16-bit and 32-bit values

Larger values are returned in the parameter 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 code similar to:

```
movlw 34h
movwf (?_return_16)
movlw 12h
movwf (?_return_16)+1
return
```

3.5.2.3 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 address of the structure or union is returned in `W`. For example:

```
struct fred
{
    int ace[4];
} ;
struct fred return_struct(void)
{
    struct fred wow;
    return wow;
}
```

will exit with the following code:

```
retlw ??_return_struct+0
```

3.6 Function Calling Convention

The baseline PIC devices have a two-level deep hardware stack which is used to store the return address of a subroutine call. Typically, call instructions are used to transfer control to a C function when it is called, however where the depth of the stack will be exceeded, HI-TECH C PRO for the PIC10/12/16 MCU Family will automatically swap to using a method that involves the use of a lookup table.

When the lookup method is being employed, a function is called by jumping directly to its address after storing the address of a jump table instruction which will be able to return control back to the calling function. The address is stored as an object local to the function being called. The lookup table is accessed after the function called has finished executing. This method allows functions to be nested without overflowing the stack, however it does come at the expense of memory and program speed.

By default the compiler will determine which functions are permitted to be called via a `call` assembly instruction and which will be called via the jump table, however this authority can be taken away from the compiler by disabling the `stackcall` suboption to the `--RUNTIME` option. With this feature turned off all function calls execute via a lookup table unless a function definition is qualified as `fastcall`. A `fastcall`-qualified function will be called via a `call` instruction. Extreme care must be used when functions are declared as `fastcall`, since the each nested fastcall function call will use one word of available stack space. Check the call graph in the map file to ensure that the stack will not overflow.

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

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

The Mid-range PIC devices have a larger hardware stack and are thus allow a higher degree of function nesting. These devices do not use the lookup table method when calling functions.

A function can return with any bank selected.

The standard version compilers assume that bank 0 will always be selected at the end of a function. This is not a requirement with PRO compilers.

The compiler tracks the bank selection throughout each function as much as possible, even across function calls. If the bank that is selected when a function returns can be determined the compiler will use this information to try to reduce the number of bank selection instructions inserted into the generated code.

The compiler will not be able to track the bank selected by functions written in assembler, even if they are called from C code. The compiler will make no assumptions about the selected bank with such routines when they return.

3.7 Operators

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

3.7.1 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 C 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 bit-field types to either `signed int` or `unsigned int`. If the result of the conversion can be represented by an `signed int`, then that is the destination type, otherwise the conversion is to `unsigned int`.

Consider the following example.

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

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

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

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

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

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

If `c` contains the value 55h, it often assumed that `~c` will produce AAh, however the result is FFAAh and so the comparison in the above example would fail. The compiler may be able to issue a

mismatched comparison error to this effect in some circumstances. Again, a cast could be used to change this behaviour.

The consequence of integral promotion as illustrated above is that operations are not performed with `char`-type operands, but with `int`-type operands. However there are circumstances when the result of an operation is identical regardless of whether the operands are of type `char` or `int`. In these cases, HI-TECH C 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. If an 8-bit addition is more efficient than a 16-bit addition, 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.

3.7.2 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.

HI-TECH C PRO for the PIC10/12/16 MCU Family 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 0x0124 shifted right one bit will yield the value 0x0092 and the value 0x8024 shifted right one bit will yield the value 0xC012.

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.

3.7.3 Division and modulus with integral types

The sign of the result of division with integers when either operand is negative is implementation specific. Table 3.5 shows the expected sign of the result of the division of operand 1 with operand 2

Table 3.5: Integral division

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

when compiled with HI-TECH C.

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

3.8 Psects

The compiler splits code and data objects into a number of standard program sections referred to as *psects*. The HI-TECH assembler allows an arbitrary number of named psects to be included in assembler code. The linker will group all data for a particular psect into a single segment.

If you are using `PICC` 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 4.3.9.3).

3.8.1 Compiler-generated Psects

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

The compiler-generated psects which are placed in the program space are:

checksum If a checksum has been requested, the result will be stored in this psect.

config Used to store the configuration word.

eeprom_data Used to store data into EEPROM memory.

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

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

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

idloc Used to store the ID location words.

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

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

intentry Contains the entry code for the interrupt service routine which is linked to the interrupt vector. This code saves the necessary registers and jumps to the main interrup code in the case of Mid-range devices; for Enhanced Mid-ranges devices this psect will contain the interrupt function body.

intrtext Contains the interrupt function body for Mid-range PIC devices. See also `intentry`.

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

maintext This psect will contain the main() function. It is used so that main() can be directly linked.

powerup Contains executable code for a user-supplied power-up routine.

pstrings For processors that support string packing, this psect will contain the packed strings.

reset_vec The reset vector.

reset_wrap For baseline PICs, this psect contains code which appears after the reset vector has wrapped around to address 0x0.

strings The `strings` psect is used for `const` objects. It also includes all unnamed string constants, such as string constants passed as arguments to routines like printf() and puts(). This psect is linked into ROM, since it does not need to be modifiable.

strings12 The Baseline equivalent of the strings psect.

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

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

The compiler-generated psects which are placed in the data space are:

intsave Holds the W register saved by the interrupt service routine. If necessary, the W register will also be saved in the intsave_n psects.

intsave_n May also hold the W register saved by the interrupt service routine. (See the description of the intsave psect.)

nvbit_n These psects are used to store persistent bit variables. They are not cleared or otherwise modified at startup.

nvram_n These psects are used to store persistent variables. They are not cleared or otherwise modified at startup.

rbit_n These psects are used to store all bit variables except those declared at absolute locations.

rbss_n These psects contain any uninitialized variables.

rdata_n These psects contain any initialised variables.

fnauto*n* These psects contain the auto and parameter variables for the entire program. ***n*** is a number and represents the bank in which it will be linked.

3.9 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*. Baseline PIC devices do not utilize interrupts and so the following sections are only applicable for Mid-range PIC devices.

3.9.1 Interrupt Functions

The function qualifier `interrupt` may be applied to C function definitions 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 appropriate instruction.

If the `PICC` option `--STRICT` is used, the `interrupt` keyword becomes `__interrupt`.

An interrupt function must be declared as type `void interrupt` and may not have parameters. This is the only function prototype that makes sense for an interrupt function. Interrupt functions may not be called directly from C code (due to the different return instruction that is used), but they themselves may call other functions.

Mid-range PIC devices have many sources of interrupt, but only one interrupt vector, and hence should only have one interrupt function defined.

An example of an interrupt function for a Mid-range PIC processor is shown here.

```
int tick_count;

void interrupt tc_int(void)
{
    if (T0IE && T0IF) {
        T0IF=0;
        ++tick_count;
    }
}
```

The interrupt vector will automatically be set to point to this function.

3.9.1.1 Context Saving on Interrupts

Some registers are automatically saved by the hardware when an interrupt occurs. Any registers or compiler temporary objects used by the interrupt function other than those saved by the hardware must be saved in software.

All Mid-range PIC processors automatically saves the entire PC (including the `PCLATH` register) on its stack whenever an interrupt occurs. Enhanced Mid-range PIC devices also save the `W`, `STATUS`, `BSR` and `FSR` registers in hardware (using special shadow registers) and hence these registers also do not need to be saved by software. In fact, the compiler will never have to produce code to save any registers when compiling for an Enhanced Mid-range as no registers other than those saved by hardware are ever used. This makes interrupt functions on Enhanced Mid-range PIC devices very fast and efficient.

The HI-TECH C PRO for the PIC10/12/16 MCU Family compiler full determines which registers and objects are used by an interrupt function, or any of the functions that it calls (or functions that they call, etc), and saves these appropriately.

HI-TECH C PRO for the PIC10/12/16 MCU Family 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. The same is true for any assembly routines called by the interrupt code.

The `intentry` psect will be linked at the interrupt vector location. This psect will hold the code that is executed immediately after the interrupt takes place. For Enhanced Mid-range devices, which have no context saving code, this will be the code associated with the body of the interrupt function. For other Mid-range devices this psect will hold the code that saves context.

If the `W` register is to be saved, it is stored to memory reserved in the common RAM. If the processor for which the code is written does not have common memory, a byte is reserved in all RAM banks for the storage location for `W` register.

Other registers to be saved are done so in the interrupt function's `auto` area, and thus look like ordinary `auto` variable.

3.9.1.2 Context Restoration

Any objects saved by the compiler are automatically restored before the interrupt function returns. The order of restoration is the reverse to that used when context is saved.

3.9.2 Enabling Interrupts

Two macros are available in <htc.h> which control the masking of all available interrupts. These macros are ei(), which enable or unmask all interrupts, and di(), which disable or mask all interrupts.

On all Mid-range PIC devices, they affect the GIE bit in the INTCON register. These macros should be used once the appropriate interrupt enable bits for the interrupts that are required in a program have been enabled.

For example:

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

3.9.3 Function Duplication

It is assumed by the compiler that an interrupt may occur at any time. As functions are not reentrant, if a user-defined function appears to be called by an interrupt function and by main-line code, or another interrupt function, this has the potential to lead to code failure.

HI-TECH C PRO for the PIC10/12/16 MCU Family has a feature which will duplicate any function called from more than one call tree in the program's call graph. A duplicate will be made for each call tree from which the function is called. The original and any duplicates contribute to the output.

These duplicate functions will have unique names for the assembly function labels themselves, labels within the functions, and local variables defined in the functions.The name consists of the usual name prefixed with i*n*, where *n* is the level number of the interrupt function. The function called from main-line code will retain its original name.

TUTORIAL

In a program the function main calls a user-defined function called input. This function is also called by an interrupt function. The output will contain the code corresponding to the original function, called input, as well as the code corresponding

to a duplicate of this, called `i1_input`. If there was a compiler-generated local label placed in the generated assembly code call `l26`, the Assembly associated with the duplicate function will contain the label `i1l26`. An auto variable defined in `input` would be referred to by the symbol `??_input` in the assembly code generated; in the duplicate this would become `??i1_input`. The assembly code for both functions will appear in the assembly list file, and all symbols associated with these functions will appear in the map file in the usual way. The call graph, in the map file, will show the calls made to both of these functions as if they were independently written.

This feature allows the programmer to write code which is independent of whether the target device allows re-entrant functions. PRO compilers will have as many duplicates of these routines precompiled in the object code libraries as there are interrupt levels. It does not handle cases where functions are called recursively.

3.9.3.1 Disabling Duplication

The automatic duplication of function may be inhibited by the use of a special pragma. This should only be done if the source code guarantees that an interrupt cannot occur while the function is being called from any main-line code. Typically this would be achieved by disabling interrupts before calling the function. It is not sufficient to disable the interrupts inside the function after it has been called. If an interrupt can occur when executing the function, the code may fail. See 3.9.2 for more information on how interrupts may be disabled.

The pragma has the form:

```
#pragma interrupt_level level
```

where *level* is the level number of the interrupt. For PICmicro devices, there is only one interrupt, and hence only one number, that being 1. The pragma should be placed before the definition of the function that is not to be duplicated. The pragma will only affect the first function whose definition follows.

For example, if the function `read` is only ever called from main-line code when the interrupts are disabled, then duplication of the function can be prevented if it is also called from an interrupt function as follows.

```
#pragma interrupt_level 1
int read(char device)
{
    // ...
}
```

in main-line code, this function would typically be called as follows.

```
di();  // turn off interrupts
read(IN_CH1);
ei();  // re-enable interrupts
```

3.9.3.2 Implicit Calls to Library Routines

Evaluation of certain C operators will require the use of C functions that are precompiled into the p-code library files. The code generator will call these routines as required. These routines are subject to the same duplication as user-defined routines, described above.

For example: if a compiler uses a routine to perform word multiplication, and this is called `wmul`, then an expression in main-line code involving such a multiplication will call `wmul`; the same code used in an `interrupt` function of level 1 will result in a call to the routine `i1wmul`; in an `interrupt` function of level 2 will call `i2wmul`, etc. These function names will be shown in the callgraph section of the map file, see Section 5.10.2.2.

3.10 Mixing C and Assembler Code

Assembly language code can be mixed with C code using two different techniques: writing assembly code and placing it into a separate assembler module, or including it as in-line assembler in a C module. For the latter, there are two formats in which this can be done.

3.10.1 External Assembly Language Functions

Entire functions may be coded in assembly language as separate `.as` source files, assembled and combined into the output 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 accessible 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 accessible

- Ensure any symbol used to hold a return value is globally accessible
- 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 Section 3.10.3.1 for a complete description of mappings between C and assembly identifiers.

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 quantity.

Most compiler-generated executable code is placed in a psect called `textn` (see Section 3.8.1). 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 executable 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 text0, will be linked in the CODE class, which will reside in a memory space that has 2 bytes per addressable location:

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

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 associate with the assembly code:

```
_add:
```

We need to be able to call this from other modules, some make this label globally accessible:

```
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 can add a assembler directive to make this signature value known, although this is not a requirement:

```
SIGNAT _add,8298
```

This function will have 4 bytes of paramters, but does not need any local (assembly equivalent of C's auto) variables. The `FNSIZE` directive informs the compiler as the memory required for the routine. See 4.3.9.17 for more information on this directive.

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 return value of the function is also located in the

same area. Here is an example of the routine for a Mid-range device. To comile this, the assembly file must be preprocessed. See Section 2.6.11.

```
#include <aspic.h>
GLOBAL _add
SIGNAT _add,8298
FNSIZE _add,0,4
psect text0,local,class=CODE,delta=2
_add:
  movf   ?_add+2,w
  addwf  ?_add,f
  btfsc  STATUS,0
  inc    ?_add+1,f
  movf   ?_add+3,w
  addwf  ?_add+1,f
  return
```

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 parameters 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.

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);
  }
```

3.10.2 #asm, #endasm and asm()

PIC 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:

```
unsigned int var;
void main(void)
{
    var = 1;
#asm          // like this...
    bcf 0,3
    rlf _var
    rlf _var+1
#endasm
              // or like this
    asm("bcf 0,3");
    asm("rlf _var");
    asm("rlf _var+1");
}
```

When using in-line assembler code, great care must be taken to avoid interacting with compiler-generated code. The code generator cannot scan the assembler code for register usage and so will remain unaware if registers are clobbered or used by the code. If in doubt, compile your program with the `PICC --ASMLIST` option (see Section 2.6.18) and examine the assembler code generated by the compiler.

3.10.3 Accessing C objects from within Assembly Code

The following applies regardless of whether the assembly is part of a separate assembly module, or in-line with C code.

For any non-local assembly symbol, the `GLOBAL` directive must be used to link in with the symbol if it was defined elsewhere. If it is a local symbol, then it may be used immediately.

3.10.3.1 Equivalent Assembly Symbols

The assembler equivalent identifier to an identifier in C code follows a form that is dependent on the scope and type of the C identifier. The different forms are discussed below. Accessing the C

identifier in C code and its assembly equivalent in assembly code implies accessing the same object. Here, "global" implies defined outside a function; "local" defined within a function.

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.

3.10.3.2 Accessing Special Function Register Names from Assembly Code

If writing separate assembly modules, SFR definitions will not automatically be present. The assembly header file `<aspic.h>` can be included (either using the assembler's `INCLUDE` directive, or using a preprocessor `#include` directive, providing the `-P` option to preprocess assembly files is selected — see Section 2.6.11).

If writing assembly code in-line in a C module, SFRs may be accessed by referring to the symbols defined by the chip-specific C header files provided that other C code also access these symbols. 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 in-line assembly code may be able to use these definitions. To use a SFR in in-line assembly 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;  // PORTA referenced in C code
#asm
    movlw #0xAA
    movwf _PORTA
#endasm
```

Note that if the registers unused in the assembly are not referenced by any C code, the necessary code to allow these symbols to be used from assembly will be missing, and the assembly header file must be included and the assembly symbols used instead. For example:

```
#include <htc.h>
void main(void)
{
// PORTA not referenced in any C code
#asm
```

```
#include <aspic.h>
    movlw #0xAA
    movwf PORTA
#endasm
```

Note in this second example that the assembly symbol for port A is used — `PORTA` — rather than the assembly symbol relating to the C definition, `_PORTA`. This is the preferred method of accessing SFRs from in-line assembly.

3.10.4 Interaction between Assembly and C Code

HI-TECH C PRO for the PIC10/12/16 MCU Family incorporates several features designed to allow C code to obey requirements of user-defined assembly code.

The command-line driver ensures that all user-defined assembly files have been processed first, before compilation of C source files begin. The driver is able to read and analyse certain information in the relocatable object files and pass this information to the code generator. This information is used to ensure the code generator takes into account requirement of the assembly code.

3.10.4.1 Absolute Psects

Some of the information that is extracted from the relocatable objects by the driver relates to absolute psects, specifically psects defined using the `abs` and `ovrld`, `PSECT` flags, see Section 4.3.9.3 for more information. These are psects have been rarely required in general coding, but do allow for data to be collated over multiple modules in a specific order.

HI-TECH C PRO for the PIC10/12/16 MCU Family is able to determine the address bounds of absolute psects to enure that the output of C code does not consume specific resources required by the assembly code. The code generator will ensure that any memory used by these psects are reserved and not used by C code. The linker options are also adjusted by the driver to enure that this memory is not allocated.

TUTORIAL

PROCESSING OF ABSOLUTE PSECTS An assembly code files defines a table that must be located at address 210h in the data space. The assembly file contains:

```
PSECT lkuptbl,class=RAM,space=1,abs,ovlrd
ORG 110h
lookup:
ds 20h
```

When the project is compiled, this file is assembled and the resulting relocatable object file scanned for absolute psects. As this psect is flagged as being `abs` and `ovlrd`, the

bounds and space of the psect will be noted — in this case a memory range from address 110h to 12fh in memory space 1 is being used. This information is passed to the code generator to ensure that these address spaces are not used by C code. The linker will also be told to remove these ranges from those available, and this reservation will be observable in the map file. The RAM class definition, for example, may look like:

```
-ARAM=020-06Fh,0A0h-0EFh,130h-16Fh,0190h-01EFh
```

for an 16F877 device, showing that addresses 110h through 12F were reserved from this class range.

3.10.4.2 Undefined Symbols

Variables can be defined in assembly code if required, but in some instances it is easier to do so in C source code, in other cases, the symbols may need to be accessable from both assembly and C source code.

A problem can occur if there is a variable defined in C code, but is never referenced throughout the entire the C program. In this case, the code generator may remove the variable believing it is unused. If assembly code is relying on this definition an error will result.

To work around this issue, HI-TECH C PRO for the PIC10/12/16 MCU Family also searches assembly-derived object files for symbols which are undefined. These will typically be symbols that are used, but not defined, in assembly code. The code generator is informed of these symbols, and if they are encountered in the C code the variable is automatically marked as being volatile. This is the equivalent of the programmer having qualified the variable as being `volatile` in the source code, see Section 3.3.9. Variables qualified as `volatile` will never be removed by the code generator, even if they appear to be unused throughout the program.

TUTORIAL

PROCESSING OF UNDEFINED SYMBOLS A C source module defines a global variable as follows:

```
int input;
```

but this variable is only ever used in assembly code. The assembly module(s) can simply declare and link in to this symbol using the `GLOBAL` assembler directive, and them make use of the symbol.

```
GLOBAL _input
PSECT text,class=CODE,delta=2
movf _input,w
```

In this instance the C variable `input` will not be removed and be treated as if it was qualified `volatile`.

3.10.4.3 Assembly Stack Usage

Assembly routines can define their own local storage — the equivalent of C auto variables — by using the `FNSIZE` directive. To ensure that any variables defined in this way are taken into account by the code generator which allocates space for the compiled stack, the `FNSIZE` directives are also scanned and information relating to these passed to the code generator.

The `FNCALL` directives are also processed so that if assembly routine call other assembly routines, this information can be passed to the code generator and the call graph updated.

TUTORIAL

PROCESSING OF ASSEMBLY LOCAL VARIABLES An assembly routine requires 2 local memory locations for storage (no arguments). The definition for the function looks like:

```
GLOBAL _read
FNSIZE _read,2,0
PSECT text,class=CODE,delta=2
_read:
movf PORTA,w
movwf ?_read     ;1st byte local storage
movf PORTB,w
mowf ?_read+1    ;2nd byte local storage
```

The compiler will take into account the 2 bytes required by this routine when calculating the stack requirements. If this routine is called from C code, then the compiler will automatically see that the routine is part of the call graph. If this routine is called from other assembly code, that code must use the `FNCALL` directive to indicate that the call has been made. For example, if this routine was called by another assembly routine called `_scan`, then the following would be required.

```
FNCALL _scan,_read
```

This directive would also be taken into account by the code generator in its generation of the call graph.

3.11 Preprocessing

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

3.11.1 Preprocessor Directives

HI-TECH C PRO for the PIC10/12/16 MCU Family accepts several specialised preprocessor directives in addition to the standard directives. All of these are listed in Table 3.6.

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

3.11.2 Predefined Macros

The compiler drivers define certain symbols to the preprocessor (`CPP`), allowing conditional compilation based on chip type etc. The symbols listed in Table 3.7 show the more common symbols defined by the drivers. Each symbol, if defined, is equated to 1 unless otherwise stated.

3.11.3 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 3.9. Those keywords not discussed elsewhere are detailed below.

3.11.3.1 The #pragma inline Directive

The #pragma inline directive is used to indicate to the compiler that a function is to be inlined. The directive is only able to be used on functions that are hard coded in the code generator of the compiler. User defined and library function are not able to be inlined. This directive should be placed directly before the function prototype of the inline function. Below is example usage

```
#pragma inline(__va_start)
extern void *           __va_start(void);
```

Table 3.6: 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 movlw FFh #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	Refer to section 3.11.3
#undef	undefines preprocessor symbol	#undef FLAG
#warning	generate a warning message	#warning Length not set

Table 3.7: Predefined macros

Symbol	When set	Usage
HI_TECH_C	Always	To indicate that the compiler in use is HI-TECH C.
_HTC_VER_MAJOR_	Always	To indicate the integer component of the compiler's version number.
_HTC_VER_MINOR_	Always	To indicate the decimal component of the compiler's version number.
_HTC_VER_PATCH_	Always	To indicate the patch level of the compiler's version number.
_HTC_EDITION_	Always	Indicates which of PRO, Standard or Lite compiler is in use. Values of 2, 1 or 0 are assigned respectively.
__PICC__	Always	Indicates HI-TECH compiler for Microchip PIC10/12/16 in use.
MPC	Always	Indicates compiling for Microchip PIC family.
_PIC12	If 12-bit device	To indicate selected device is a baseline PIC.
_PIC14	If 14-bit device	To indicate selected device is a Mid-range PIC.
COMMON	If common RAM present	To indicate whether device has common RAM area.
BANKBITS	Always	Assigned 0, 1 or 2 to indicate 1, 2 or 4 available banks or RAM.
GPRBITS	Always	Assigned 0, 1 or 2 to indicate 1, 2 or 4 available banks or general purpose RAM.
__MPLAB_ICD__	If compiling for MPLAB ICD or ICD2 debugger	Assigned 1 to indicate that the code is generated for use with the Microchip MPLAB ICD1. Assigned 2 for ICD2.
_ROMSIZE	Always	To indicate how many words of program memory are available.
_EEPROMSIZE	Always	To indicate how many bytes of EEPROM are available.
_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`.

Table 3.9: Pragma directives

Directive	Meaning	Example
`inline`	Specify function as inline	`#pragma inline(fabs)`
`interrupt_level`	Allow call from interrupt and mainline code	`#pragma interrupt_level 1`
`jis`	Enable JIS character handling in strings	`#pragma jis`
`nojis`	Disable JIS character handling (default)	`#pragma nojis`
`pack`	Specify structure packing	`#pragma pack 1`
`printf_check`	Enable printf-style format string checking	`#pragma printf_check(printf) const`
`regsused`	Specify registers used by function	`#pragma regsused wreg,fsr`
`switch`	Specify code generation for switch statements	`#pragma switch direct`
`warning`	Control messaging parameters	`#pragma warning disable 299,407`

3.11.3.2 The #pragma interrupt_level Directive

The #pragma interrupt_level directive can be used to prevent function duplication of functions called from main-line and interrupt code. See Section 3.9.3.1 for more information.

3.11.3.3 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.

3.11.3.4 The #pragma pack Directive

Some MCUs requires word accesses to be aligned on word boundaries. Consequently the compiler will align all word or larger quantities onto a word boundary, including structure members. This can lead to "holes" in structures, where a member has been aligned onto the next word boundary.

This behaviour can be altered with this directive. Use of the directive `#pragma pack 1` will prevent any padding or alignment within structures. Use this directive with caution - in general if you must access data that is not aligned on a word boundary you should do so by extracting individual bytes and re-assembling the data. This will result in portable code. Note that this directive must *not* appear before any system header file, as these must be consistent with the libraries supplied.

PICs can only perform byte accesses to memory and so do not require any alignment of memory objects. This pragma will have no effect when used.

3.11.3.5 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 2.6.60.

Table 3.10: Valid Register Names

Register Name	Description
`fsr`	indirect data pointer
`wreg`	the working register
status	the status register

3.11.3.6 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. 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. This cannot be used for specifying the used registers for C functions.

The general form of the pragma is:

```
#pragma regsused routine_name register_list
```

where *routine_name* is the assembly name of the function or routine whose register usage is being defined, and *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 any unspecified 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.

3.11.3.7 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 .

Table 3.11: switch types

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

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.

3.11.3.8 The #pragma warning Directive

The warning disable pragma Some warning messages can be disabled by using the `warning disable` pragma. This pragma will only affect warnings that are produced by either parser or the code generator, i.e. errors directly associated with C code. The position of the pragma is only significant for the parser, i.e. a parser warning number may be disabled, then re-enabled around a section of the code to target specific instances of the warning. Specific instances of a warning produced by the code generator cannot be individually controlled. The pragma will remain in force during compilation of the entire module.

The state of those warnings which have been disabled can preserved and recalled using the `warning push` and `warning pop` pragmas. Pushes and pops can be nested to allow a large degree of control over the message behaviour.

TUTORIAL

DISABLING A WARNING The following example shows the warning associated with qualifying an `auto` object being disabled, number 348.

```
void main(void)
{
#pragma warning disable 348
near int c;
#pragma warning enable 348
/* etc */
}
int rv(int a)
{
near int c;
```

```
/* etc */
}
```

which will issue only one warning associated with the second definition of the auto variable `c`. Warning number 348 is disabled during parsing of the definition of the auto variable, `c`, inside the function `main`.

```
altst.c:  35:  (348) auto variable "c" should not be qualified
(warning)
```

This same affect would be observed using the following code.

```
void main(void)
{
#pragma warning push
#pragma warning disable 348
near int c;
#pragma warning pop
/* etc */
}
int rv(int a)
{
near int c;
/* etc */
}
```

Here the state of the messaging system is saved by the `warning push` pragma. Warning 348 is disabled, then after the source code which triggers the warning, the state of the messaging system is retrieved by the use of the `warning pop` pragma.

The warning error/warning pragma It is also possible to change the type of some messages. This is only possible by the use of the `warning pragma` and only affects messages generated by the parser or code generator. The position of the pragma is only significant for the parser, i.e. a parser message number may have its type changed, then reverted back around a section of the code to target specific instances of the message. Specific instances of a message produced by the code generator cannot be individually controlled. The pragma will remain in force during compilation of the entire module.

TUTORIAL

The following shows the warning produced in the previous example being converted to an error for the instance in the function `main()`.

```
void main(void)
{
#pragma warning error 348
near int c;
#pragma warning warning 348
/* etc */
}
int rv(int a)
{
near int c;
/* etc */
}
```

Compilation of this code would result in an error, and as with any error, this will force compilation to cease after the current module has concluded, or the maximum error count has been reached.

3.12 Linking Programs

The compiler will automatically invoke the linker unless requested to stop after producing an intermediate file as specified by any of the options: `--PRE`, `--PASS1`, `-S` or `-C`.

The linker will run with a set of options that are passed to it from the command-line driver. These options specify the memory of the device. No linker scripts are used. The linker options passed to the linker can be adjusted by the user, but this is only required in special circumstances. See 2.6.7 for more information.)

HI-TECH C, by default, generates Intel HEX. 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 `PICC --SUMMARY=psect` option. Generate a map file for the complete memory specification of the program.

3.12.1 Replacing Library Modules

Although HI-TECH C comes with a librarian (LIBR) which allows you to unpack a library files and replace modules with your own modified versions, you can easily replace a library module that is linked into your program without having to do this. If you add the source file, which contains a definition for a library routine you wish to replace, to your project then the routine will replace the routine in the library file with the same name.

This method works due to the way the compiler scans source and library files. When trying to resolve a symbol (in this instance a function name) the compiler 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 compiler 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, which will result in an error.

For example, if you wished to make changes to the library function max() which resides in the file max.c in the SOURCES directory, you could make a copy of this source file, make the appropriate changes and then compile and use it as follows.

```
PICC --chip=16F877A 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.

Most libraries are written C code, and the p-code libraries that contain these library routines are actually passed to the code generator, not the linker, but both these applications work in the way described above in resolving library symbols.

You cannot replace a C library function with an equivalent written in assembly code using the above method. If this is required, you will need to use the librarian to edit or create a new library file.

3.12.2 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 C PRO for the PIC10/12/16 MCU Family is only likely to issue a mismatch error from the linker when the routine is either a pre-compiled object file or an assembly routine. Other function mismatches are reported by the code generator.

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 check the assembly list file using the PICC --ASMLIST option (see Section 2.6.18). 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)
{
}
```

The resultant assembler code seen in the assembly list file 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 mis-match which will alert you to the possible existence of incompatible calling conventions.

3.12.3 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*.

Table 3.12: 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`

3.13 Standard I/O Functions and Serial I/O

A number of the standard I/O functions are provided, specifically those functions intended to read and write formatted text on standard output and input. A list of the available functions is in Table 3.12. 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()`.

Chapter 4

Macro Assembler

The Macro Assembler included with HI-TECH C PRO for the PIC10/12/16 MCU Family assembles source files for PIC 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 `ASPIC` and is available to run on *Windows*, *Linux* and *Mac OS* systems. 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, PICC, 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 2.6.55.

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:

```
ASPIC [ options ] files
```

files is a space-separated list of one or more assembler source files. Where more than one source file is specified the assembler treats them as a single module, i.e. a single assembly will be performed on the concatenation of all the source files specified. The files must be specified in full, no default extensions or suffixes are assumed.

options is an optional space-separated list of assembler options, each with a *minus sign* - as the first character. A full list of possible options is given in Table 4.1, and a full description of each option follows.

Table 4.1: ASPIC 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.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 ASPIC 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 assembler produced using the `-O` option.

-C A cross reference file will be produced when this option is used. This file, called *`srcfile`*`.crf`, where *`srcfile`* is the base portion of the first source file name, will contain raw cross reference 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.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.
Specifying a digit as argument has a similar effect, only it allows selection of any of the available 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 the listing format is pageless, i.e. the assembler 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 ASPIC` 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.

-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 4.3.9.25. 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.

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

-X The object file created by the assembler contains symbol information, including local symbols, i.e. symbols that are neither public or external. The `-X` option will prevent the local symbols from being included in the object file, thereby reducing the file size.

4.3 HI-TECH C Assembly Language

The source language accepted by the macro assembler, `ASPIC`, is described below. All opcode mnemonics and operand syntax are strictly PIC assembly language. Additional mnemonics and assembler directives are documented in this section.

4.3.1 Assembler Format Deviations

The HI-TECH PICC assembler uses a slightly modified form of assembly language to that specified by the Microchip datahseets. Certain PIC instructions used by Microchip assembler use the operands "`,0`" or "`,1`" to specify the destination for the result of that operation. The HI-TECH PICC assembler uses the more-readable operands "`,w`" and "`,f`" to specify the destination register. The W register is selected as the destination when using the "`,w`" operand, and the file register is selected when using the "`,f`" operand or if no destination operand is specified. The case of the letter in the destination operand in not important. The numerical destination operands cannot be used with the HI-TECH PICC assembler.

The assembler also recognizes seveal mnemonics which expand into regular PIC assembly instruction. The mnemonics are `fcall` and `ljmp`. These instructions expand into `call` and `goto`

Table 4.2: ASPICstatement 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>			

instructions respecitively, but also include the instructions necessary to set the bits in `PCLATH` for a call or jump to another page of program memory. These additional mnemonics should be used where possible as they make assembler code independent of the final position of the routines that are to be executed. If the call or jump is determined to be within the current page, the additional code to set the `PCLATH` bits may be optimized away.

4.3.2 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 2.6.11, then it may also contain lines that form valid preprocessor directives. See Section 3.11.1 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.

4.3.3 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*.

4.3.3.1 Delimiters

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

Table 4.3: ASPIC numbers and bases

Radix	Format
Binary	digits 0 and 1 followed by `B`
Octal	digits 0 to 7 followed by `O`, `Q`, `o` or `q`
Decimal	digits 0 to 9 followed by `D`, `d` or nothing
Hexadecimal	digits 0 to 9, A to F preceded by `0x` or followed by `H` *or* `h`

4.3.3.2 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.

4.3.4 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 2.6.11, then it may also contain C or C++ style comments using the standard `/* ... */` and `//` syntax.

4.3.4.1 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 achieve the same effect for locations defined and accessed in assembly code.

4.3.5 Constants

4.3.5.1 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.

4.3.5.2 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.

4.3.6 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
```

4.3.6.1 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.

4.3.6.2 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.

4.3.6.3 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.

4.3.6.4 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 <aspic.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 2.6.11. This header file contains appropriate commands to ensure that the header file specific for the target device is included into the source file.

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

4.3.6.5 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 and optionally 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.

Note that the colon following the label is optional, therefore symbols which are not interpreted in any other way are assumed to be labels. Thus the code:

```
mowlv 23h
bananas
movf 37h
```

defined a symbol called bananas. Mis-typed assembler instructions can sometimes be treated as labels without an error message being issued. Labels may be used (and are preferred) 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 accessible in other modules, use the GLOBAL directive. See Section 4.3.9.1 for more information.

4.3.7 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.

Table 4.4: ASPIC operators

Operator	Purpose	Example
*	Multiplication	`movlw 4*33,W`
+	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 77mod4`
&	Bitwise AND	`clrf inp&0ffh`
^	Bitwise XOR (exclusive or)	`movf inp^80,W`
|	Bitwise OR	`movf inp!1,W`
not	Bitwise complement	`movlw not 055h,W`
< < or shl	Shift left	`DB inp><8`
> > or shr	Shift right	`movlw inp shr 2,W`
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	

4.3.8 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 4.3.9.3 for full information on psect definitions. Chapter 5 has more information on the operation of the linker and on options 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.

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 text0,class=CODE,delta=2
adjust
   goto  clear_fred
increment
   incf _fred
PSECT rbss_0,class=BANK0,space=1
fred
   DS 2
PSECT text0,class=CODE,delta=2
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 `rbss` 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 output. The actual location in memory of the `text` and `rbss` 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.

4.3.9 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.

4.3.9.1 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
```

4.3.9.2 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
```

4.3.9.3 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.

Table 4.5: ASPIC 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
DABS	Define absolute 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 graph
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

- 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.

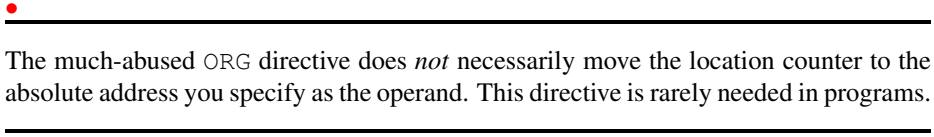
- The `reloc` flag allows specification of a requirement for alignment of the psect on a particular boundary, e.g. `reloc=100h` would specify that this psect must start on an address that is a multiple of 100h.
- The `size` flag allows a maximum size to be specified for the psect, e.g. `size=100h`. This 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
```

4.3.9.4 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
```

4.3.9.5 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 4.3.9.6.

4.3.9.6 SET

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

```
thomas SET 0h
```

4.3.9.7 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.

4.3.9.8 DW

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

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

4.3.9.9 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
```

4.3.9.10 DABS

This directive allows one or more bytes of memory to be reserved at the specified address. The general form of the directive is:

```
DABS memory_space,address,bytes
```

where *memory_space* is a number representing the memory space in which the reservation will take place, *address* is the address at which the reservation will take place, and *bytes* is the number of bytes that is to be reserved. This directive differs to the DS directive in that it does not allocate space at the curernt location in the current psect, but instead can be used to reserve memory at any location.

The memory space number is the same as the number specified with the space flag option to psects. Devices with a single flat memory space will typically always use 0 as the space value; devices with separate code and data spaces typically use 0 for the code space and 1 for the data space.

The code generator issues a DABS directive for every user-defined absolute C variable, or for variables that have been allocated an address by the code generator.

4.3.9.11 FNADDR

This directive tells the linker that a function has its address taken, and thus could be called indirectly through a function pointer. For example

```
FNADDR _func1
```

tells the linker that func1() has its address taken.

4.3.9.12 FNARG

The directive

```
FNARG fun1,fun2
```

tells the linker that evaluation of the arguments to function fun1 involves a call to fun2, thus the memory argument memory allocated for the two functions should not overlap. For example, the C function calls

```
fred(var1, bill(), 2);
```

will generate the assembler directive

```
FNARG _fred,_bill
```

thereby telling the linker that bill() is called while evaluating the arguments for a call to fred().

4.3.9.13 FNBREAK

This directive is used to break links in the call graph information. The form of this directive is as follows:

```
FNBREAK fun1,fun2
```

and is automatically generated when the interrupt_level pragma is used. It states that any calls to fun1 in trees other than the one rooted at fun2 should not be considered when checking for functions that appear in multiple call graphs. Fun2() is typically intlevel0 or intlevel1 in compiler-generated code when the interrupt_level pragma is used. Memory for the auto/parameter area for a fun1 will only be assigned in the tree rooted at fun2.

4.3.9.14 FNCALL

This directive takes the form:

```
FNCALL fun1,fun2
```

`FNCALL` is usually used in compiler generated code. It tells the linker that function `fun1` calls function `fun2`. This information is used by the linker when performing call graph analysis. If you write assembler code which calls a C function, use the `FNCALL` directive to ensure that your assembler function is taken into account. For example, if you have an assembler routine called _fred which calls a C routine called foo(), in your assembler code you should write:

```
FNCALL _fred,_foo
```

4.3.9.15 FNCONF

The FNCONF directive is used to supply the linker with configuration information for a call graph. FNCONF is written as follows:

```
FNCONF psect,auto,args
```

where psect is the psect containing the call graph, auto is the prefix on all auto variable symbol names and args is the prefix on all function argument symbol names. This directive normally appears in only one place: the runtime startup code used by C compiler generated code. For the HI-TECH C PRO for the PIC10/12/16 MCU Family the startup routine will include the directive:

```
FNCONF rbss,??,?
```

telling the linker that the call graph is in the rbss psect, auto variable blocks start with ?? and function argument blocks start with ?.

4.3.9.16 FNINDIR

This directive tells the linker that a function performs an indirect call to another function with a particular signature (see the SIGNAT directive). The linker must assume worst case that the function could call any other function which has the same signature and has had its address taken (see the FNADDR directive). For example, if a function called fred() performs an indirect call to a function with signature 8249, the compiler will produce the directive:

```
FNINDIR _fred,8249
```

4.3.9.17 FNSIZE

The FNSIZE directive informs the linker of the size of the local variable and argument area associated with a function. These values are used by the linker when building the call graph and assigning addresses to the variable and argument areas. This directive takes the form:

```
FNSIZE func,local,args
```

The named function has a local variable area and argument area as specified, for example

```
FNSIZE _fred, 10, 5
```

means the function fred() has 10 bytes of local variables and 5 bytes of arguments. The function name arguments to any of the call graph associated directives may be local or global. Local functions are of course defined in the current module, but most be used in the call graph construction in the same manner as global names.

4.3.9.18 FNROOT

This directive tells the assembler that a function is a root function and thus forms the root of a call graph. It could either be the C main() function or an interrupt function. For example, the C main module produce the directive:

```
FNROOT _main
```

4.3.9.19 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.

4.3.9.20 MACRO and ENDM

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

For example:

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

```
;descr: Move a literal value into a nominated file register:
movlf   MACRO   arg1,arg2
   movlw arg1
   movwf arg2 mod 080h
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:

```
movlf 2,tempvar
```

expands to:

```
movlw 2
movwf tempvar mod 080h
```

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 4.3.10.2,

4.3.9.21 LOCAL

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

```
down MACRO count
   LOCAL more
   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.

4.3.9.22 ALIGN

The ALIGN directive aligns whatever is following, data storage or code etc., to the specified boundary in the psect in which the directive is found. The boundary is specified by a number following the directive and 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.

4.3.9.23 REPT

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

```
REPT 3
addwf fred,w
ENDM
```

will expand to

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

4.3.9.24 IRP and IRPC

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

For example:

```
PSECT idata_0
    IRP number,4865h,6C6Ch,6F00h
    DW number
ENDM
PSECT text0
```

would expand to:

```
PSECT idata_0
    DW 4865h
    DW 6C6Ch
    DW 6F00h
PSECT text0
```

Note that you can use local labels and *angle brackets* in the same manner as with conventional macros.

The `IRPC` directive is similar, except it substitutes one character at a time from a string of non-space characters.

For example:

```
PSECT 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
```

4.3.9.25 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 PICC, see Section 2.6.22, or using the assembler `-P` option, see Table 4.1, but can also be set with this directive, e.g.

```
PROCESSOR 16F877
```

4.3.9.26 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.

4.3.10 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: ASPIC 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*>"

4.3.10.1 COND

Any conditional code will be included in the listing output. See also the NOCOND control in Section 4.3.10.5.

4.3.10.2 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 4.3.10.6.

4.3.10.3 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 2.6.11 which forces the C preprocessor to preprocess assembly file, thus allowing use of preprocessor directives, such as #include (see Section 3.11.1).

4.3.10.4 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 4.3.10.7.

4.3.10.5 NOCOND

Using this control will prevent conditional code from being included in the listing output. See also the COND control in Section 4.3.10.1.

4.3.10.6 NOEXPAND

NOEXPAND disables macro expansion in the listing file. The macro call will be listed instead. See also the EXPAND control in Section 4.3.10.2. Assembly macro are discussed in Section 4.3.9.20.

4.3.10.7 NOLIST

This control turns the listing output off from this point onward. See also the LIST control in Section 4.3.10.4.

4.3.10.8 NOXREF

NOXREF will disable generation of the *raw* cross reference file. See also the XREF control in Section 4.3.10.13.

4.3.10.9 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.

4.3.10.10 SPACE

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

4.3.10.11 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 4.3.10.12.

4.3.10.12 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 4.3.10.11.

4.3.10.13 XREF

`XREF` is equivalent to the driver command line option `--CR` (see Section 2.6.25). 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 4.3.9.3 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 4.3.9.14 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 4.3.9.17).

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 2.6.8.

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 2.6.30 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\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 \
-pintsave_0=07Fh -ppowerup=CODE \
-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\HI-TECH Software\pic\std\9.60\lib\pic412-c.lib \
C:\Program\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 2.6.7.

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 4.3.9.18 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 4.3.9.14 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 4.3.9.17 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 2.6.52) 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 4.3.9.1 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 3.12.3. 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 2.6.18 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

Table 5.2: Librarian command-line options

Option	Effect
`-Pwidth`	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
`o`	Re-order modules

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:

```
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.

The o key takes a list of module names and re-orders the matching modules in the library file so they have the same order as that listed on the command line. Modules which are not listed are left in their existing order, and will appear after the re-ordered modules.

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.

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.

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 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.

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*

5.13.1 -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.

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 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

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

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

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

additional 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.

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

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

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:

```
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*][w*Width*][t*Code*][g*Algorithm*]

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.

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
7	Fletcher's checksum (8 bit)
8	Fletcher's checksum (16 bit)

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.
- *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*[m*Mask*]@*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.

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.

- *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.

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 -MASK

Use this option to logically AND a memory range with a particular bitmask. This is used to ensure that the unimplemented bits in program words (if any) are left blank. The usage of this option is as follows:

```
-MASK=hexcode@start-end
```

Where *hexcode* is a hexadecimal value that will be ANDed with data within the *start-end* address range. Multibyte mask values can be entered in little endian byte order.

5.15.1.14 -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.

5.15.1.15 -SERIAL

This option will store a particular hex value at a fixed address. The usage of this option is:

```
-SERIAL=Code[+/-Increment]@Address[+/-Interval][rRepetitions]
```

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`.

5.15.1.16 -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.

5.15.1.17 -STRING

The -STRING option will embed an ASCII string at a fixed address. The usage of this option is:

-STRING@*Address*[t*Code*]=”*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`.

5.15.1.18 -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 decomposed into the following categories.

Synopsis the C declaration of the function, and the header file in which it is declared.

Description a narrative description of the function and its purpose.

Example an example of the use of the function. It is usually a complete small program that illustrates the function.

Data types any special data types (structures etc.) defined for use with the function. These data types will be defined in the header file named under **Synopsis**.

See also any allied functions.

Return value the type and nature of the return value of the function, if any. Information on error returns is also included

Only those categories which are relevant to each function are used.

__CONFIG

Synopsis

```
#include <htc.h>

__CONFIG(data)
```

Description

This macro is used to program the configuration fuses that set the device into various modes of operation.

The macro accepts 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(RC & UNPROTECT)

void
main (void)
{
}
```

See also

__EEPROM_DATA(), __IDLOC(), __IDLOC7()

__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 4 nibbles of data to the 4 locations reserved for ID purposes.

Example

```
#include <htc.h>

__IDLOC(15F0);
/* will store 1, 5, F and 0 in the ID registers*/

void
main (void)
{
}
```

See also

__IDLOC7(), __CONFIG()

__IDLOC7

Synopsis

```
#include <htc.h>

__IDLOC7(a,b,c,d)
```

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 7 bits of data to each of the 4 locations reserved for ID purposes.

Example

```
#include <htc.h>

__IDLOC(0x7F,70,1,0x5A);
/* will store 7Fh, 70, 1 and 5Ah in the ID registers */

void
main (void)
{
}
```

Note

Not all devices permit 7 bit programming of the ID locations. Refer to the device datasheet to see whether this macro can be used on your particular device.

See also

__IDLOC(), __CONFIG()

_DELAY, __DELAY_MS, __DELAY_US

Synopsis

```
__delay_ms(x)  // request a delay in milliseconds
__delay_us(x)  // request a delay in microseconds

// request a delay for a number of instruction cycles
void _delay_ms(unsigned long n)
```

Description

The when code calls `_delay(n)`, the code generator will customize and in-lined sequence of code to facilitate a delay of *n* instruction cycles. As this routine is customized for the parameter given, the resultant code produced may differ significantly based on the magnitude of the requested delay.

As it is often more convenient request a delay in time-based terms rather than in cycle counts, the macros `__delay_ms(x)` and `__delay_us(x)` are provided. These macros simply wrap around `_delay(n)` and convert the time based request into instruction cycles based on the system frequency. In order to achieve this, these macros require the prior definition of preprocessor symbol `_XTAL_FREQ`. This symbol should be defined as the oscillator frequency (in Hertz) used by the system.

An error will result if these macros are used without defining this symbol or if the delay period requested is too large.

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();
}
```

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.

DI, EI

Synopsis

```
#include <htc.h>

void ei (void)
void di (void)
```

Description

The **di()** and **ei()** routines disable and re-enable interrupts respectively. These are implemented as macros defined in **PIC.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;
}
```

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);
}
```

See Also

udiv(), ldiv(), uldiv()

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 char addr);
void eeprom_write (unsigned char addr, unsigned char value);
```

Description

These function 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 the device. Writing a value to the eeprom is a slow process and the **eeprom_write()** function polls the appropriate registers to ensure that any previous writes have completed before writing the next 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 char address;

    address = 0x10;
    data = eeprom_read(address);
}
```

Note

It may be necessary to poll the eeprom registers to ensure that the write has completed if an **eeprom_write()** call is immediately followed by an **eeprom_read()**. The global interrupt enable bit (GIE) is now restored by the **eeprom_write()** routine. The EEIF interrupt flag is not reset by this function.

EVAL_POLY

Synopsis

```
#include <math.h>

double eval_poly (double x, const double * d, int n)
```

Description

The **eval_poly()** function evaluates a polynomial, whose coefficients are contained in the array **d**, at **x**, for example:

```
y = x*x*d2 + x*d1 + d0.
```

The order of the polynomial is passed in **n**.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    double x, y;
    double d[3] = {1.1, 3.5, 2.7};

    x = 2.2;
    y = eval_poly(x, d, 2);
    printf("The polynomial evaluated at %f is %f\n", x, y);
}
```

Return Value

A double value, being the polynomial evaluated at **x**.

EXP

Synopsis

```
#include <math.h>

double exp (double f)
```

Description

The **exp()** routine returns the exponential function of its argument, i.e. e to the power of **f**.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    for(f = 0.0 ; f <= 5 ; f += 1.0)
        printf("e to %1.0f = %f\n", f, exp(f));
}
```

See Also

log(), log10(), pow()

FABS

Synopsis

```
#include <math.h>

double fabs (double f)
```

Description

This routine returns the absolute value of its double argument.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f %f\n", fabs(1.5), fabs(-1.5));
}
```

See Also

abs(), labs()

FLASH_COPY

Synopsis

```
#include <htc.h>

void flash_copy(const unsigned char * source_addr,
     unsigned char length, unsigned short dest_addr);
```

Description

This utility function is useful for copying a large section of memory to a new location in flash memory.

Note it is only applicable to those devices which have an internal set of flash buffer registers.

When the function is called, it needs to be supplied with a **const pointer** to the source address of the data to copy. The pointer may point to a valid address in either RAM or flash memory.

A length parameter must be specified to indicate the number of words of the data to be copied.

Finally the flash address where this data is destined must be specified.

Example

```
#include <htc.h>

const unsigned char ROMSTRING[] = "0123456789ABCDEF";

void
main (void){
    const unsigned char * ptr = &ROMSTRING[0];
    flash_copy( ptr, 5, 0x70 );
}
```

See Also

EEPROM_READ, EEPROM_WRITE, FLASH_READ, FLASH_WRITE

Note

This function is only applicable to those devices which use internal buffer registers when writing to flash.

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.

FLASH_ERASE(), FLASH_READ()

Synopsis

```
#include <htc.h>

void flash_erase (unsigned short addr);
unsigned int flash_read (unsigned short addr);
```

Description

These functions allow access to the flash memory of the microcontroller (if supported).

Reading from the flash memory can be done one word at a time with use of the **flash_read()** function. **flash_read()** returns the data value found at the specified word address in flash memory.

Entire sectors of 32 words can be restored to an unprogrammed state (value=**FF**) with use of the **flash_erase()** function. Specifying an address to the **flash_erase()** function, will erase all 32 words in the sector that contains the given address.

Example

```
#include <htc.h>

void
main (void)
{
    unsigned int data;
    unsigned short address=0x1000;

    data = flash_read(address);

    flash_erase(address);

}
```

Return Value

flash_read() returns the data found at the given address, as an unsigned int.

Note

The functions **flash_erase()** and **flash_read()** are only available on those devices that support such functionality.

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.

GET_CAL_DATA

Synopsis

```
#include <htc.h>

double get_cal_data (const unsigned char * code_ptr)
```

Description

This function returns the 32-bit floating point calibration data from the PIC 14000 calibration space. Only use this function to access KREF, KBG, VHTHERM and KTC (that is, the 32-bit floating point parameters). FOSC and TWDT can be accessed directly as they are bytes.

Example

```
#include <htc.h>

void
main (void)
{
    double x;
    unsigned char y;

        /* Get the slope reference ratio. */
    x = get_cal_data(KREF);

        /* Get the WDT time-out. */
    y =TWDT;
}
```

Return Value

The value of the calibration parameter

Note

This function can only be used on the PIC 14000.

GMTIME

Synopsis

```
#include <time.h>

struct tm * gmtime (time_t * t)
```

Description

This function converts the time pointed to by **t** which is in seconds since 00:00:00 on Jan 1, 1970, into a broken down time stored in a structure as defined in **time.h**. The structure is defined in the 'Data Types' section.

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;
    struct tm * tp;

    time(&clock);
    tp = gmtime(&clock);
    printf("It's %d in London\n", tp->tm_year+1900);
}
```

See Also

ctime(), asctime(), time(), localtime()

Return Value

Returns a structure of type **tm**.

Note

The example will require the user to provide the time() routine as one cannot be supplied with the compiler. See time() for more detail.

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(), uldiv(), udiv()

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.

MEMCHR

Synopsis

```
#include <string.h>

/* For baseline and midrange processors */
const void * memchr (const void * block, int val, size_t length)

/* For high-end processors */
void * memchr (const void * block, int val, size_t length)
```

Description

The **memchr()** function is similar to strchr() except that instead of searching null terminated strings, it searches a block of memory specified by length for a particular byte. Its arguments are a pointer to the memory to be searched, the value of the byte to be searched for, and the length of the block. A pointer to the first occurrence of that byte in the block is returned.

Example

```
#include <string.h>
#include <stdio.h>

unsigned int ary[] = {1, 5, 0x6789, 0x23};

void
main (void)
{
    char * cp;

    cp = memchr(ary, 0x89, sizeof ary);
    if(!cp)
        printf("not found\n");
    else
        printf("Found at offset %u\n", cp - (char *)ary);
}
```

See Also

strchr()

Return Value

A pointer to the first byte matching the argument if one exists; NULL otherwise.

MEMCMP

Synopsis

```
#include <string.h>

int memcmp (const void * s1, const void * s2, size_t n)
```

Description

The **memcmp()** function compares two blocks of memory, of length **n**, and returns a signed value similar to strncmp(). Unlike strncmp() the comparison does not stop on a null character.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    int buf[10], cow[10], i;

    buf[0] = 1;
    buf[2] = 4;
    cow[0] = 1;
    cow[2] = 5;
    buf[1] = 3;
    cow[1] = 3;
    i = memcmp(buf, cow, 3*sizeof(int));
    if(i < 0)
        printf("less than\n");
    else if(i > 0)
        printf("Greater than\n");
    else
        printf("Equal\n");
}
```

See Also

strncpy(), strncmp(), strchr(), memset(), memchr()

Return Value

Returns negative one, zero or one, depending on whether **s1** points to string which is less than, equal to or greater than the string pointed to by **s2** in the collating sequence.

MEMCPY

Synopsis

```
#include <string.h>

/* For baseline and midrange processors */
void * memcpy (void * d, const void * s, size_t n)

/* For high-end processors */
far void * memcpy (far void * d, const void * s, size_t n)
```

Description

The **memcpy()** function copies **n** bytes of memory starting from the location pointed to by **s** to the block of memory pointed to by **d**. The result of copying overlapping blocks is undefined. The **memcpy()** function differs from strcpy() in that it copies a specified number of bytes, rather than all bytes up to a null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];

    memset(buf, 0, sizeof buf);
    memcpy(buf, "a partial string", 10);
    printf("buf = '%s'\n", buf);
}
```

See Also

strncpy(), strncmp(), strchr(), memset()

Return Value

The **memcpy()** routine returns its first argument.

MEMMOVE

Synopsis

```
#include <string.h>

/* For baseline and midrange processors */
void * memmove (void * s1, const void * s2, size_t n)

/* For high-end processors */
far void * memmove (far void * s1, const void * s2, size_t n)
```

Description

The **memmove()** function is similar to the function memcpy() except copying of overlapping blocks is handled correctly. That is, it will copy forwards or backwards as appropriate to correctly copy one block to another that overlaps it.

See Also

strncpy(), strncmp(), strchr(), memcpy()

Return Value

The function **memmove()** returns its first argument.

MEMSET

Synopsis

```
#include <string.h>

/* For baseline and midrange processors */
void * memset (void * s, int c, size_t n)

/* For high-end processors */
far void * memset (far void * s, int c, size_t n)
```

Description

The **memset()** function fills **n** bytes of memory starting at the location pointed to by **s** with the byte **c**.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char abuf[20];

    strcpy(abuf, "This is a string");
    memset(abuf, 'x', 5);
    printf("buf = '%s'\n", abuf);
}
```

See Also

strncpy(), strncmp(), strchr(), memcpy(), memchr()

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**.

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. Field widths and precision are only supported on the midrange and high-end processors, with the precision specification only applicable to **%s**.

If the character ***** is used in place of a decimal constant, e.g. in the format **%*d**, then one integer argument will be taken from the list to provide that value. The types of conversion for the Baseline series are:

o x X u d

Integer conversion - in radices 8, 16, 16, 10 and 10 respectively. The conversion is signed in the case of **d**, unsigned otherwise. The precision value is the total number of digits to print, and may be used to force leading zeroes. E.g. **%8.4x** will print at least 4 hex digits in an 8 wide field. The letter **X** prints out hexadecimal numbers using the upper case letters *A-F* rather than *a-f* as would be printed when using **x**. When the alternate format is specified, a leading zero will be supplied for the octal format, and a leading 0x or 0X for the hex format.

s

Print a string - the value argument is assumed to be a character pointer. At most **n** characters from the string will be printed, in a field **m** characters wide.

c

The argument is assumed to be a single character and is printed literally.

Any other characters used as conversion specifications will be printed. Thus **%** will produce a single percent sign.

For the Midrange and High-end series, the types of conversions are as for the Baseline with the addition of:

l

Long integer conversion - Preceding the integer conversion key letter with an **l** indicates that the argument list is long.

f

Floating point - **m** is the total width and **n** is the number of digits after the decimal point. If **n** is omitted it defaults to 6. If the precision is zero, the decimal point will be omitted unless the alternate format is specified.

Example

```
printf("Total = %4d%", 23)
       yields 'Total =   23%'

printf("Size is %lx" , size)
       where size is a long, prints size
       as hexadecimal.

Note: precision number is only available when using Midrange
and High-end processors when using the %s  placeholder.
printf("Name = %.8s", "a1234567890")
       yields 'Name = a1234567'

Note: the variable width number is only available when using
Midrange and High-end processors placeholder.
printf("xx%*d", 3, 4)
       yields 'xx  4'

/* vprintf example */

#include <stdio.h>

int
error (char * s, ...)
{
    va_list ap;

    va_start(ap, s);
    printf("Error: ");
    vprintf(s, ap);
```

```
        putchar('\n');
        va_end(ap);
    }

    void
    main (void)
    {
        int i;

        i = 3;
        error("testing 1 2 %d", i);
    }
```

See Also

sprintf()

Return Value

The **printf()** routine returns the number of characters written to stdout.
NB The return value is a char, NOT an int.

Note

Certain features of printf are only available for the midrange and high-end processors. Read the description for details. Printing floating point numbers requires that the float to be printed be no larger than the largest possible long integer. In order to use long or float formats, the appropriate supplemental library must be included. See the description on the PICC -L option and the HPDPIC Options/Long formats in printf menu for more details.

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.

Upon entry to this function, the working register contains an error code, the address that failed can be determined from the FSR register and IRP bit. The failed value will still be accessable through the INDF 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.

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\n", i, sin(i*C));
        printf("cos(%3.0f) = %f\n", i, cos(i*C));
}
```

See Also

cos(), tan(), asin(), acos(), atan(), atan2()

Return Value

Sine vale of **f**.

SPRINTF

Synopsis

```
#include <stdio.h>

/* For baseline and midrange processors */
unsigned char sprintf (char *buf, const char * fmt, ...)

/* For high-end processors */
unsigned char sprintf (far char *buf, const char * fmt, ...)
```

Description

The **sprintf()** function operates in a similar fashion to printf(), except that instead of placing the converted output on the *stdout stream*, the characters are placed in the buffer at **buf**. The resultant string will be null terminated, and the number of characters in the buffer will be returned.

See Also

printf()

Return Value

The **sprintf()** routine returns the number of characters placed into the buffer.
NB: The return value is a char not an int.

Note

For High-end processors the buffer is accessed via a far pointer.

SQRT

Synopsis

```
#include <math.h>

double sqrt (double f)
```

Description

The function **sqrt()**, implements a square root routine using Newton's approximation.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double i;

    for(i = 0 ; i <= 20.0 ; i += 1.0)
        printf("square root of %.1f = %f\n", i, sqrt(i));
}
```

See Also

exp()

Return Value

Returns the value of the square root.

Note

A domain error occurs if the argument is negative.

SRAND

Synopsis

```
#include <stdlib.h>

void srand (unsigned int seed)
```

Description

The **srand()** function initializes the random number generator accessed by rand() with the given **seed**. This provides a mechanism for varying the starting point of the pseudo-random sequence yielded by rand(). On the Z80, a good place to get a truly random seed is from the refresh register. Otherwise timing a response from the console will do, or just using the system time.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t toc;
    int i;

    time(&toc);
    srand((int)toc);
    for(i = 0 ; i != 10 ; i++)
        printf("%d\t", rand());
    putchar('\n');
}
```

See Also

rand()

STRCAT

Synopsis

```
#include <string.h>

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.

STRCAT

Synopsis

```
#include <string.h>

/* For baseline and midrange processors */
char * strcat (char * s1, const char * s2)

/* For high-end processors */
far char * strcat (far char * s1, const char * s2)
```

Description

This function appends (contcatenates) string **s2** to the end of string **s1**. The result will be null terminated. The argument **s1** must point to a character array big enough to hold the resultant string.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = " ... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

See Also

strcpy(), strcmp(), strncat(), strlen()

Return Value

The value of **s1** is returned.

STRCHR, STRICHR

Synopsis

```
#include <string.h>

char * strchr (const char * s, int c)
char * strichr (const char * s, int c)
```

Description

The **strchr()** function searches the string **s** for an occurrence of the character **c**. If one is found, a pointer to that character is returned, otherwise NULL is returned.

The **strichr()** function is the case-insensitive version of this function.

Example

```
#include <strings.h>
#include <stdio.h>

void
main (void)
{
    static char temp[] = "Here it is...";
    char c = 's';

    if(strchr(temp, c))
        printf("Character %c was found in string\n", c);
    else
        printf("No character was found in string");
}
```

See Also

strrchr(), strlen(), strcmp()

Return Value

A pointer to the first match found, or NULL if the character does not exist in the string.

Note

Although the function takes an integer argument for the character, only the lower 8 bits of the value are used.

STRCHR, STRICHR

Synopsis

```
#include <string.h>

/* For baseline and midrange processors */
const char * strchr (const char * s, int c)
const char * strichr (const char * s, int c)

/* For high-end processors */
char * strchr (const char * s, int c)
char * strichr (const char * s, int c)
```

Description

The **strchr()** function searches the string **s** for an occurrence of the character **c**. If one is found, a pointer to that character is returned, otherwise NULL is returned.

The **strichr()** function is the case-insensitive version of this function.

Example

```
#include <strings.h>
#include <stdio.h>

void
main (void)
{
    static char temp[] = "Here it is...";
    char c = 's';

    if(strchr(temp, c))
        printf("Character %c was found in string\n", c);
    else
        printf("No character was found in string");
}
```

See Also

strrchr(), strlen(), strcmp()

Return Value

A pointer to the first match found, or NULL if the character does not exist in the string.

Note

The functions takes an integer argument for the character, only the lower 8 bits of the value are used.

STRCMP, STRICMP

Synopsis

```
#include <string.h>

int strcmp (const char * s1, const char * s2)
int stricmp (const char * s1, const char * s2)
```

Description

The **strcmp()** function compares its two, null terminated, string arguments and returns a signed integer to indicate whether **s1** is less than, equal to or greater than **s2**. The comparison is done with the standard collating sequence, which is that of the ASCII character set.

The **stricmp()** function is the case-insensitive version of this function.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    int i;

    if((i = strcmp("ABC", "ABc")) < 0)
        printf("ABC is less than ABc\n");
    else if(i > 0)
        printf("ABC is greater than ABc\n");
    else
        printf("ABC is equal to ABc\n");
}
```

See Also

strlen(), strncmp(), strcpy(), strcat()

Return Value

A signed integer less than, equal to or greater than zero.

Note

Other C implementations may use a different collating sequence; the return value is negative, zero or positive, i.e. do not test explicitly for negative one (-1) or one (1).

STRCPY

Synopsis

```
#include <string.h>

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.

STRCPY

Synopsis

```
#include <string.h>

/* For baseline and midrange processors */
char * strcpy (char * s1, const char * s2)

/* For high-end processors */
far char * strcpy (far char * s1, const char * s2)
```

Description

This function copies a null terminated string **s2** to a character array pointed to by **s1**. The destination array must be large enough to hold the entire string, including the null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = " ... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

See Also

strncpy(), strlen(), strcat(), strlen()

Return Value

The destination buffer pointer **s1** is returned.

STRCSPN

Synopsis

```
#include <string.h>

size_t strcspn (const char * s1, const char * s2)
```

Description

The **strcspn()** function returns the length of the initial segment of the string pointed to by **s1** which consists of characters NOT from the string pointed to by **s2**.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    static char set[] = "xyz";

    printf("%d\n", strcspn( "abcdevwxyz", set));
    printf("%d\n", strcspn( "xxxbcadefs", set));
    printf("%d\n", strcspn( "1234567890", set));
}
```

See Also

strspn()

Return Value

Returns the length of the segment.

STRLEN

Synopsis

```
#include <string.h>

size_t strlen (const char * s)
```

Description

The **strlen()** function returns the number of characters in the string **s**, not including the null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = " ... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

Return Value

The number of characters preceding the null terminator.

STRNCAT

Synopsis

```
#include <string.h>

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.

STRNCAT

Synopsis

```
#include <string.h>

/* For baseline and midrange processors */
char * strncat (char * s1, const char * s2, size_t n)

/* For high-end processors */
far char * strncat (far char * s1, const char * s2, size_t n)
```

Description

This function appends (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.

STRNCPY

Synopsis

```
#include <string.h>

/* For baseline and midrange processors */
char * strncpy (char * s1, const char * s2, size_t n)

/* For high-end processors */
far char * strncpy (far char * s1, const char * s2, size_t n)
```

Description

This function copies a null terminated string **s2** to a character array pointed to by **s1**. At most **n** characters are copied. If string **s2** is longer than **n** then the destination string will not be null terminated. The destination array must be large enough to hold the entire string, including the null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strncpy(buffer, "Start of line", 6);
    s1 = buffer;
    s2 = " ... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

See Also

strcpy(), strcat(), strlen(), strcmp()

Return Value

The destination buffer pointer **s1** is returned.

STRPBRK

Synopsis

```
#include <string.h>

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.

STRPBRK

Synopsis

```
#include <string.h>

/* For baseline and midrange processors */
const char * strpbrk (const char * s1, const char * s2)

/* For high-end processors */
char * strpbrk (const char * s1, const char * s2)
```

Description

The **strpbrk()** function returns a pointer to the first occurrence in string **s1** of any character from string **s2**, or a null pointer if no character from **s2** exists in **s1**.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * str = "This is a string.";

    while(str != NULL) {
        printf( "%s\n", str );
        str = strpbrk( str+1, "aeiou" );
    }
}
```

Return Value

Pointer to the first matching character, or NULL if no character found.

STRRCHR, STRRICHR

Synopsis

```
#include <string.h>

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.

STRRCHR, STRRICHR

Synopsis

```
#include <string.h>

/* For baseline and midrange processors */
const char * strrchr (char * s, int c)
const char * strrichr (char * s, int c)

/* For high-end processors */
char * strrchr (char * s, int c)
char * strrichr (char * s, int c)
```

Description

The **strrchr()** function is similar to the **strchr()** function, but searches from the end of the string rather than the beginning, i.e. it locates the *last* occurrence of the character **c** in the null terminated string **s**. If successful it returns a pointer to that occurrence, otherwise it returns NULL.

The **strrichr()** function is the case-insensitive version of this function.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * str = "This is a string.";

    while(str != NULL) {
        printf( "%s\n", str );
        str = strrchr( str+1, 's');
    }
}
```

See Also

strchr(), strlen(), strcmp(), strcpy(), strcat()

Return Value

A pointer to the character, or NULL if none is found.

STRSPN

Synopsis

```
#include <string.h>

size_t strspn (const char * s1, const char * s2)
```

Description

The **strspn()** function returns the length of the initial segment of the string pointed to by **s1** which consists entirely of characters from the string pointed to by **s2**.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    printf("%d\n", strspn("This is a string", "This"));
    printf("%d\n", strspn("This is a string", "this"));
}
```

See Also

strcspn()

Return Value

The length of the segment.

STRSTR, STRISTR

Synopsis

```
#include <string.h>

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.

STRSTR, STRISTR

Synopsis

```
#include <string.h>

/* For baseline and midrange processors */
const char * strstr (const char * s1, const char * s2)
const char * stristr (const char * s1, const char * s2)

/* For high-end processors */
char * strstr (const char * s1, const char * s2)
char * stristr (const char * s1, const char * s2)
```

Description

The **strstr()** function locates the first occurrence of the sequence of characters in the string pointed to by **s2** in the string pointed to by **s1**.

The **stristr()** routine is the case-insensitive version of this function.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    printf("%d\n", strstr("This is a string", "str"));
}
```

Return Value

Pointer to the located string or a null pointer if the string was not found.

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.

STRTOK

Synopsis

```
#include <string.h>

/* For baseline and midrange processors */
char * strtok (char * s1, const char * s2)

/* For high-end processors */
far char * strtok (far char * s1, const char * s2)
```

Description

A number of calls to **strtok()** breaks the string **s1** (which consists of a sequence of zero or more text tokens separated by one or more characters from the separator string **s2**) into its separate tokens.

The first call must have the string **s1**. This call returns a pointer to the first character of the first token, or NULL if no tokens were found. The inter-token separator character is overwritten by a null character, which terminates the current token.

For subsequent calls to **strtok()**, **s1** should be set to a null pointer. These calls start searching from the end of the last token found, and again return a pointer to the first character of the next token, or NULL if no further tokens were found.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * ptr;
    char * buf = "This is a string of words.";
    char * sep_tok = ".,?! ";

    ptr = strtok(buf, sep_tok);
    while(ptr != NULL) {
        printf("%s\n", ptr);
```

```
            ptr = strtok(NULL, sep_tok);
        }
    }
```

Return Value

Returns a pointer to the first character of a token, or a null pointer if no token was found.

Note

The separator string **s2** may be different from call to call.

TAN

Synopsis

```
#include <math.h>

double tan (double f)
```

Description

The **tan()** function calculates the tangent of **f**.

Example

```
#include <math.h>
#include <stdio.h>

#define C 3.141592/180.0

void
main (void)
{
    double i;

    for(i = 0 ; i <= 180.0 ; i += 10)
        printf("tan(%3.0f) = %f\n", i, tan(i*C));
}
```

See Also

sin(), cos(), asin(), acos(), atan(), atan2()

Return Value

The tangent of **f**.

TIME

Synopsis

```
#include <time.h>

time_t time (time_t * t)
```

Description

This function is not provided as it is dependant on the target system supplying the current time. This function will be user implemented. When implemented, this function should return the current time in seconds since 00:00:00 on Jan 1, 1970. If the argument **t** is not equal to NULL, the same value is stored into the object pointed to by **t**.

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;

    time(&clock);
    printf("%s", ctime(&clock));
}
```

See Also

ctime(), gmtime(), localtime(), asctime()

Return Value

This routine when implemented will return the current time in seconds since 00:00:00 on Jan 1, 1970.

Note

The **time()** routine is not supplied, if required the user will have to implement this routine to the specifications outlined above.

TOLOWER, TOUPPER, TOASCII

Synopsis

```
#include <ctype.h>

char toupper (int c)
char tolower (int c)
char toascii (int c)
```

Description

The **toupper()** function converts its lower case alphabetic argument to upper case, the **tolower()** routine performs the reverse conversion and the **toascii()** macro returns a result that is guaranteed in the range 0-0177. The functions **toupper()** and **tolower()** return their arguments if it is not an alphabetic character.

Example

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

void
main (void)
{
    char * array1 = "aBcDE";
    int i;

    for(i=0;i < strlen(array1); ++i) {
        printf("%c", tolower(array1[i]));
    }
    printf("\n");
}
```

See Also

islower(), isupper(), isascii(), et. al.

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()

UDIV

Synopsis

```
#include <stdlib.h>

int udiv (unsigned num, unsigned demon)
```

Description

The **udiv()** function calculate the quotient and remainder of the division of `number` and `denom`, storing the results into a udiv_t structure which is returned.

Example

```
#include <stdlib.h>

void
main (void)
{
    udiv_t result;
    unsigned  num = 1234, den = 7;

    result = udiv(num, den);
}
```

See Also

uldiv(), div(), ldiv()

Return Value

Returns the the quotient and remainder as a udiv_t structure.

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(), udiv(), 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");
}
```

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.

(1) too many errors (*) ***(all applications)***

The executing compiler application has encountered too many errors and will exit immediately. Other uncompiled source files will be processed, but the compiler applications that would normally be executed in due course will not be run. The number of errors that can be accepted can be controlled using the `--ERRORS` option, See Section 2.6.30.

(2) error/warning (*) generated, but no description available ***(all applications)***

The executing compiler application has emitted a message (advisory/warning/error), but there is no description available in the message description file (MDF) to print. This may be because the MDF is out of date, or the message issue has not been tranlated into the selected language.

(3) malformed error information on line *, in file * ***(all applications)***

The compiler has attempted to load the messages for the selected language, but the message description file (MDF) was corrupted and could not be read correctly.

(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 #assert 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  /* in-line assembler ends here,
            only where did it begin? */
}
```

(106) nested #asm directives ***(Preprocessor)***

It is not legal to nest `#asm` directives. Check for a missing or misspelt `#endasm` directive, e.g.:

```
#asm
  move r0, #0aah
#asm      ; 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  /* oops -- that should be #undef DEBUG */
```

(108) #if[n][def] 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             /* oops -- 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  /* oops -- 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 preprocessor 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: */
/* Is this correct? It will overwrite the first definition. */
#define ONE one
```

(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 preprocessor 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 preprocessor 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 preprocessor 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;    /* oops -- 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  /* oops -- did you mean: #if FOO == BAR ? */
```

(118) stack overflow processing #if expression ***(Preprocessor)***

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

(119) invalid expression in #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 operator "*" ***(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 parenthesis at operator "*" ***(Preprocessor)***

The evaluation of a `#if` expression found mismatched parentheses. Check the expression for correct parenthesisation, e.g.:

```
#if ((A) + (B)  /* oops -- 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) strings can't be used in #if ***(Preprocessor)***

The preprocessor does not allow the use of strings in `#if` expressions, e.g.:

```
/* no string operations allowed by the preprocessor */
#if MESSAGE > "hello"
#define DEBUG
#endif
```

(127) bad syntax for defined() in #[el]if ***(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.:

```
/* oops -- defined expects a name, not an expression */
#if defined(a&b)
  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  /* oops -- 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) unknown type "*" in #[el]if sizeof() ***(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  /* should be: #if sizeof(int) == 2 */
  i = 0xFFFF;
#endif
```

(131) illegal type combination in #[el]if sizeof() ***(Preprocessor)***

The preprocessor found an illegal type combination in the argument to `sizeof()` in a `#if` expression, e.g.

```
/* To sign, or not to sign, that is the error. */
#if sizeof(signed unsigned int) == 2
  i = 0xFFFF;
#endif
```

(132) no type specified in #[el]if sizeof() ***(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()  /* oops -- size of what? */
  i = 0;
#endif
```

(133) unknown type code (0x*) in #[el]if sizeof() ***(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) syntax error in #[el]if sizeof() ***(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)  // oops - should be: #if sizeof(int) == 2
  i = 0xFFFF;
#endif
```

(135) unknown operator (*) in #if ***(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.:

```
/* the ' character will not lead to a valid token */
#define cc(a, b) a ## 'b
```

(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.:

```
/* the ' character will not lead to a valid token */
#define cc(a, b) a ## 'b
```

(139) end of file in comment ***(Preprocessor)***

End of file was encountered inside a comment. Check for a missing closing comment flag, e.g.:

```
/* Here the comment begins. I'm not sure where I end, though
}
```

(140) can't open * file "*": * ***(Driver, Preprocessor, Code Generator, Assembler)***

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 * file "*": * ***(Any)***

An output file could not be created. Confirm the spelling and path of the file specified on the command line.

(144) too many nested #if blocks ***(Preprocessor)***

`#if`, `#ifdef` etc. blocks may only be nested to a maximum of 32.

(146) #include filename too long ***(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 specified ***(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 preprocessor macro ***(Preprocessor)***

A macro may only have up to 31 parameters, as per the C Standard.

(149) preprocessor macro work area overflow ***(Preprocessor)***

The total length of a macro expansion has exceeded the size of an internal table. This table is normally 32768 bytes long. Thus any macro expansion must not expand into a total of more than 32K bytes.

(150) illegal "__" preprocessor macro "*" ***(Preprocessor)***

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

(151) too many arguments in preprocessor 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 preprocessor macro "*" argument expansion ***(Preprocessor)***

A macro argument has exceeded the length of an internal buffer. This buffer is normally 4096 bytes long.

(155) work buffer overflow conatenating "*" ***(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) can't allocate * bytes of memory ***(Code Generator, Assembler, Optimiser)***

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

(158) invalid disable in preprocessor macro "*" ***(Preprocessor)***

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

(159) too many calls to unget() ***(Preprocessor)***

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

(161) control line "*" within preprocessor macro expansion ***(Preprocessor)***

A preprocessor control line (one starting with a #) has been encountered while expanding a macro. This should not happen.

(162) #warning: * ***(Preprocessor, Driver)***

This warning is either the result of user-defined `#warning` preprocessor directive or the driver encountered a problem reading the the map file. If the latter then please HI-TECH Software technical support with details

(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) #include filename "*" was converted to lower case ***(Preprocessor)***

The `#include` file name had to be converted to lowercase before it could be opened, e.g.:

```
#include <STDIO.H>  /* oops -- should be: #include <stdio.h> */
```

(165) #include filename "*" does not match actual name (check upper/lower case) ***(Preprocessor)***

In Windows versions this means the file to be included actually exists and is spelt the same way as the `#include` filename, however the case of each does not exactly match. For example, specifying `#include "code.c"` will include `Code.c` if it is found. In Linux versions this warning could occur if the file wasn't found.

(166) too few values specified with option "*" ***(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) too many values specified with -S option; "*" 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 366.

(168) unknown option "*" ***(Any)***

This option given to the component which caused the error is not recognized.

(169) strange character (*) after ## ***(Preprocessor)***

There is an unexpected character after `#`.

(170) symbol "*" in undef was never defined ***(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 preprocessor 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)        /* oops -- 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  /* oops -- 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) symbol "*" in -U option was never 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; /* next 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 to other types ***(Parser)***

You can't convert a structure, union or array to another type, e.g.:

```
struct TEST test;
struct TEST * sp;
sp = test;       /* oops -- did you mean: sp = &test; ? */
```

(182) illegal conversion between types ***(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;          /* int cannot be converted to 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) calling an interrupt function is illegal ***(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);  /* oops --
                            parameter should not be here */
}
```

(186) too many function arguments ***(Parser)***

This function does not accept as many arguments as there are here.

```
void add(int a, int b);
add(5, 7, input);      /* call has too many arguments */
```

(187) too few function 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:  /* oops!
              can't use 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];  /* oops -- 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];   /* oops --
                     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;  /* oops -- 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;  /* oops -- 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;  /* oops -- 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) taking the address of a register variable is illegal ***(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;
  /* oops -- in may not have an address to take */
  return ip;
}
```

(201) taking the address of this object is illegal *(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;  /* oops -- 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 isn't a variable,
                 it can't be written to */
```

A typecast does not yield an lvalue, e.g.:

```
/* the contents of c cast to int
   is only a intermediate value */
(int)c = 1;
```

However you can write this using pointers:

```
*(int *)&c = 1
```

(203) illegal operation on 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;  /* oops --
             cannot take the address of a bit object */
```

(204) void function can't return a 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 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.

(211) taking sizeof bit is illegal ***(Parser)***

It is illegal to use the `sizeof` operator with the HI-TECH C `bit` type. When used against a type the `sizeof` operator gives the number of bytes required to store an object that type. Therefore its usage with the `bit` type make no sense and is an illegal operation.

(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  /* oops -- 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  /* oops -- 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;  /* oops -- what is the exponent? */
```

(221) hexadecimal 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;  /* oops -- 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;
/* leading 0 implies octal which has digits 0 - 7 */
```

(224) illegal "#" directive ***(Parser)***

An illegal # preprocessor has been detected. Likely a directive has been misspelt in your code somewhere.

(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';  /* oops -- 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`;  /* oops -- 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 argument 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 argument 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 ***(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;
}
/* only one prototype & definition of rv can exist */
long twice(long a)
{
  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.:

```
/* oops -- uint is a type, not a variable */
typedef unsigned int uint = 99;
```

(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;  /* attempting to define another of 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.:

```
/* three elements, but four initializers */
int ivals[3] = { 2, 4, 6, 8};
```

(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'};
/* oops -- 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) {  /* oops -- 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;
    } /* oops! 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" case 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 381.

(246) case label 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;
    }          /* oops -- 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];        /* oops -- 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.:

```
/* this is supposedly calc's prototype */
extern int calc(int, int);
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)    /* oops --
                 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;
/* oops -- this results in two volatile qualifiers */
volatile vint very_vol;
```

(259) can't be qualifed both far and near ***(Parser)***

It is illegal to qualify a type as both `far` and `near`, e.g.:

```
far near int spooky;  /* oops -- choose 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) struct/union 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;  /* oops -- 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 can't 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);  /* 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;   /* 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:  /* oops -- 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 {
  /* no additional qualifiers may be present with members */
  static int first;
} ;
```

(267) bad storage class ***(Code Generator)***

The code generator has encountered a variable definition whose storage class is invalid, e.g.:

```
auto int foo; /* auto not permitted with global variables */
int power(static int a)  /* parameters 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) variable can't have storage class "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) type can't be long ***(Parser)***

Only `int` and `float` can be qualified with `long`.

```
long char lc;  /* what? */
```

(272) type can't be short ***(Parser)***

Only `int` can be modified with `short`, e.g.:

```
short float sf;  /* what? */
```

(273) type can't be both signed and unsigned ***(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? */
```

(274) type can't be unsigned ***(Parser)***

A floating point type cannot be made `unsigned`, e.g.:

```
unsigned float uf;  /* what? */
```

(275) "..." illegal in non-prototype argument 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:

```
/* K&R-style non-prototyped function definition */
int kandr(a, b, ...)
  int a, b;
{
```

(276) type specifier required for prototyped argument ***(Parser)***

A type specifier is required for a prototyped argument. It is not acceptable to just have an identifier.

(277) can't mix prototyped and non-prototyped arguments ***(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)  /* oops -- 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.

```
/* can't have two parameters called "a" */
int calc(int a, int a)
```

(279) initialization of function arguments is illegal ***(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.:

```
/* oops -- a is initialized when proc is called */
extern int proc(int a = 9);
```

(280) arrays of functions are illegal ***(Parser)***

You can't define an array of functions. You can however define an array of pointers to functions, e.g.:

```
int * farray[]();  /* oops -- 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.:

```
/* This should be, e.g.: int arr[][7] */
int get_element(int arr[2][])
{
  return array[1][6];
}
```

(284) invalid dimension ***(Parser)***

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

(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) arrays of bits or pointers to bit are illegal *(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.:

```
/* variables cannot be qualified interrupt */
interrupt int input;
```

(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) /* const * char ccrv(void) perhaps? */
{                     /* error flagged here */
  return ccip;
}
```

(291) K&R identifier "*" not an argument ***(Parser)***

This identifier that has appeared in a K&R style argument declarator is not listed inside the parentheses after the function name, e.g.:

```
int process(input)
int unput;        /* oops -- that should be int input; */
{
}
```

(292) function 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.

(296) out of memory ***(Objtohex)***

This could be an internal compiler error. Contact HI-TECH Software technical support with details.

(297) bad argument (*) to tysize() ***(Parser)***

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

(298) end of file 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
}                /* oops -- where is the #endasm */
```

(300) unexpected end of file *(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) end of file 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) can't allocate * bytes of memory (line *) *(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.

(306) can't allocate * bytes of memory for * *(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 case labels 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);  /* oops --
                         should be: process(carray[idx]); */
```

(311) closing quote expected ***(Parser)***

A closing quote was expected for the indicated string.

(312) "*" expected ***(Parser)***

The indicated token was expected by the parser.

(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.:

```
/* the function block must follow, not a semicolon */
int get_value(a, b);
```

(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--  /* oops -- where is the semicolon? */
}          /* error is flagged here */
```

Note: Omitting a semicolon from statements not preceding a close brace or keyword typically results in some other error being issued for the following code which the parser assumes 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.:

```
/* oops! no opening brace after the prototype */
void process(char c)
  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;  /* oops -- 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;          /* oops -- 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 printf-style format string ***(Parser)***

There are too many arguments for this format string. This is harmless, but may represent an incorrect format string, e.g.:

```
/* oops -- missed a placeholder? */
printf("%d - %d", low, high, median);
```

(325) error in printf-style 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);  /* oops -- maybe: printf("%ld", lll); */
```

(326) long int argument required in printf-style format string ***(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);  // maybe you meant: printf("%lx", 2L);
```

(327) long long int argument required in printf-style format string ***(Parser)***

A long long argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments, e.g.:

```
printf("%llx", 2);  // maybe you meant: printf("%llx", 2LL);
```

Note that not all HI-TECH C compilers provide support for a long long integer type.

(328) int argument required in printf-style format string ***(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); /* wrong number or wrong placeholder */
```

(329) double argument required in printf-style format string ***(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 in printf-style format string ***(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 printf-style 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);
  /* oops! 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  /* oops -- 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.:

```
/* oops -- should that be #pragma strings const ? */
#pragma strings cinst
```

(334) unrecognized qualifier name after "printf_check" ***(Parser)***

The `#pragma printf_check` was passed a qualifier that could not be identified, e.g.:

```
/* oops -- should that be const not cinst? */
#pragma printf_check(printf) cinst
```

(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 "*" ***(Any)***

The application tried to create or open the named file, but it could not be created. Check that all file pathnames are correct.

(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? */
```

(340) string not terminated by null character. ***(Parser)***

A char array is being initialized with a string literal larger than the array. Hence there is insufficient space in the array to safely append a null terminating character, e.g.:

```
char foo[5] = "12345"; /* the string stored in foo won't have
                          a null terminating, i.e.
                          foo = ['1', '2', '3', '4', '5'] */
```

(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)
{
  /* this would be better outside the function */
  extern int away;
  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 for "*" ***(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 parameter 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)  /* f will be converted to double */
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;    /* oops -- 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;     /* oops -- 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;
/* yes, but what sort of characters? */
cp = "I am a string 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];           /* oops -- 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:

```
/* I may prevent an error arising from calls below */
void set(long a, int b);
void main(void)
{
  /* by here a prototype for set should have seen */
  set(10L, 6);
}
```

(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;
/* array is a constant, not a variable; the & is redundant. */
ip = &array;
```

(363) redundant "&" or "*" applied to function address *(Parser)*

The address operator "&" has been applied to a function. Since using the name of a function gives its address anyway, this is unnecessary and has been ignored, e.g.:

```
extern void foo(void);
void main(void)
{
    void(*bar)(void);
    /* both assignments are equivalent */
    bar = &foo;
    bar = foo;  /* the & is redundant */
}
```

(364) attempt to modify object qualified * *(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;               /* oops --
                          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;
  /* returning this is dangerous;
     the pointer could be dereferenced */
  return &c;
}
```

(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) identifier 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)
{
  /* at this point the compiler assumes set is extern... */
  set(10L, 6);
}
/* now it finds out otherwise */
static void set(long a, int b)
{
  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.:

```
/* param should be: "int array[]" or "int *" */
int get_first(int array[10])
{                               /* 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;
} ;
```

(370) illegal basic type; int assumed ***(Parser)***

The basic type of a cast to a qualified basic type couldn't not be recognised and the basic type was assumed to be `int`, e.g.:

```
/* here ling is assumed to be int */
unsigned char bar = (unsigned ling) 'a';
```

(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;
/* thinks: chat & b are unsigned, but where is the comma? */
unsigned chat b;
```

(373) implicit signed to unsigned conversion ***(Parser)***

An `unsigned` type was expected where a `signed` type was given and was implicitly cast to `unsigned`, e.g.:

```
unsigned int foo = -1;
/* the above initialization is implicitly treated as:
   unsigned int foo = (unsigned) -1; */
```

(374) missing basic type; int assumed ***(Parser)***

The basic type of a cast to a qualified basic type was missing and assumed to be `int`., e.g.:

```
int i = (signed) 2; /* (signed) assumed to be (signed int) */
```

(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 *, expression * ***(Linker, Parser)***

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

(387) illegal or too many -G options ***(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 option ***(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 options ***(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) missing argument to -P ***(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 argument to -Q ***(Linker)***

The `-Q` linker option requires the machine type for an argument.

(392) missing argument to -U ***(Linker)***

The `-U` (undefine) option needs an argument.

(393) missing argument to -W ***(Linker)***

The `-W` option (listing width) needs a numeric argument.

(394) duplicate -D or -H option ***(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 argument to -J ***(Linker)***

The maximum number of errors before aborting must be specified following the `-j` linker option.

(397) usage: hlink [-options] files.obj files.lib ***(Linker)***

Improper usage of the command-line linker. If you are invoking the linker directly then please refer to Section 5.7 for more details. Otherwise this may be an internal compiler error and you should contact HI-TECH Software technical support with details.

(398) output file can't 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.

(402) bad argument to -F ***(Objtohex)***

The `-F` option for `objtohex` has been supplied an invalid argument. If you are invoking this command-line tool directly then please refer to Section 5.12 for more details. Otherwise this may be an internal compiler error and you should contact HI-TECH Software technical support with details.

(403) bad -E option: "*" ***(Objtohex)***

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.

(406) bad argument to -A ***(Objtohex)***

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

(407) bad argument to -U ***(Objtohex)***

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

(408) bad argument to -B *(Objtohex)*

This option requires an integer argument in either base 8, 10 or 16. If you are invoking `objtohex` directly then see Section 5.12 for more details. Otherwise this may be an internal compiler error and you should contact HI-TECH Software technical support with details.

(409) bad argument to -P *(Objtohex)*

This option requires an integer argument in either base 8, 10 or 16. If you are invoking `objtohex` directly then see Section 5.12 for more details. Otherwise this may be an internal compiler error and you should contact HI-TECH Software technical support with details.

(410) bad combination of options *(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 in COFF file *(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 in "*" *(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.

(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 syntactically incorrect.

(431) bad argument to -E ***(Objtoexe)***

This option requires an integer argument in either base 8, 10 or 16. If you are invoking `objtoexe` directly then check this argument. Otherwise this may be an internal compiler error and you should contact HI-TECH Software technical support with details.

(432) usage: objtohex [-ssymfile] [object-file [exe-file]] ***(Objtohex)***

Improper usage of the command-line tool `objtohex`. If you are invoking `objtohex` directly then please refer to Section 5.12 for more details. Otherwise this may be an internal compiler error and you should 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 segment selector "*" ***(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 information.

(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
```

(439) bad -D spec: "*" ***(Linker)***

The format of a `-D` specification, giving a *delta* value to a class, is invalid, e.g.:

```
-DCODE
```

What is the *delta* value for this class? Maybe you meant something like:

```
-DCODE=2
```

(440) bad delta value in -D spec ***(Linker)***

The *delta* value supplied to a `-D` specification is invalid. This value should an integer of base 8, 10 or 16.

(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
```

(442) missing address in -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 information.

(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) psect "*" was never defined ***(Linker, Optimiser)***

This psect has been listed in a `-P` option, but is not defined in any module within the program.

(451) bad psect origin format in -P option ***(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 "+" (minimum address) format in -P option ***(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.

(454) link and load address can't both be set to "." in -P option ***(Linker)***

The link and load address of a psect have both been specified with a *dot* character. Only one of these addresses may be specified in this manner, e.g.:

```
-Pmypsect=1000h/.
-Pmypsect=./1000h
```

Both of these options are valid and equivalent, however the following usage is ambiguous:

```
-Pmypsect=./.
```

What is the link or load address of this psect?

(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.

(456) 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.

(459) remove failed, error: *, * ***(xstrip)***

The creation of the output file failed when removing an interemediate file.

(460) rename failed, error: *, * ***(xstrip)***

The creation of the output file failed when renaming an interemediate file.

(461) can't create * file "*" ***(Assembler, Code Generator)***

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) {
      /* recursion may not be supported by some compilers */
      return test(a++);
   }
   return 0;
}
```

(472) non-reentrant 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 */
}
```

(473) function "*" is not called from specified interrupt_level ***(Linker)***

The indicated function is never called from an interrupt function of the same interrupt level, e.g.:

```
#pragma interrupt_level 1
void foo(void)
{
    ...
}
#pragma interrupt_level 1
void interrupt bar(void)
{
    // this function never calls foo()
}
```

(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 startup 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 * * (location 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 (location 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.

(478) * range check failed (location 0x* (0x*+*), value 0x* > limit 0x*) ***(Linker)***

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

(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) function 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: */
/* this is different to the declaration */
int get_value(int in, char type)
{
```

(481) common symbol "*" psect conflict ***(Linker)***

A common symbol has been defined to be in more than one psect.

(482) symbol "*" is defined more than once in "*" ***(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:           ; oops -- 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 "*" can't be global ***(Linker)***

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

(484) psect "*" can't 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 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 0x* 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 accommodate 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 3.8.1 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 program'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 accommodate 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 2.6.8 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 restructuring the program. If a code psect must be split into two or more small psects, this requires 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 graph:` 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 accommodate the psect `text` which is 0x34 bytes long. Notice, however, that the total amount of memory available is larger than 0x34 bytes.

(492) attempt to position absolute psect "*" is illegal *(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) origin of psect "*" is defined more than once ***(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
```

(495) use of both "with=" and "INCLASS/INCLASS" allocation is illegal ***(Linker)***

It is not legal to specify both the link and location of a psect as within a class, when that psect was also defined using a `with` psect flag.

(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) program entry point is defined more than once ***(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 inobject file ***(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) object file record too long: *+* ***(Linker)***

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

(507) unexpected end of file in object 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.

(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 function "*" ***(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 to -W option (*) illegal and 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 (memory) 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.

(529) usage: objtohex [-Ssymfile] [object-file [hex-file]] ***(Objtohex)***

Improper usage of the command-line tool `objtohex`. If you are invoking `objtohex` directly then please refer to Section 5.12 for more details. Otherwise this may be an internal compiler error and you should contact HI-TECH Software technical support with details.

(593) can't find 0x* words (0x* withtotal) for psect "*" in segment "*" ***(Linker)***

See error (491) on Page 424.

(594) undefined symbol: ***(Linker)***

The symbol following is undefined at link time. This could be due to spelling error, or failure to link an appropriate module.

(595) 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.

(596) segment "*" (*-*) overlaps segment "*" (*-*) ***(Linker)***

The named segments have overlapping code or data. Check the addresses being assigned by the `-P` linker option.

(599) No psect classes given for COFF write ***(Cromwell)***

Cromwell requires that the program memory psect classes be specified to produce a COFF file. Ensure that you are using the `-N` option as per Section 5.14.2.

(600) No chip arch given for COFF write ***(Cromwell)***

Cromwell requires that the chip architecture be specified to produce a COFF file. Ensure that you are using the -P option as per Section 5.14.1.

(601) Unknown chip arch "*" for COFF write ***(Cromwell)***

The chip architecture specified for producing a COFF file isn't recognised by Cromwell. Ensure that you are using the -P option as per Section 5.14.1 and that the architecture specified matches one of those in Table 5.8.

(602) null file format name ***(Cromwell)***

The `-I` or `-O` option to Cromwell must specify a file format.

(603) ambiguous file 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 file 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.

(608) bad path specification ***(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.

(610) missing psect classes after -N ***(Cromwell)***

Cromwell requires that the `-N` option be given a list of the names of psect classes.

(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.

(616) option -Cbaseaddr is illegal with options -R or -L ***(Linker)***

The linker option `-Cbaseaddr` cannot be used in conjunction with either the `-R` or `-L` linker options.

(618) error reading COD file data ***(Cromwell)***

An error occurred reading the input COD file. Confirm the spelling and path of the file specified on the command line.

(619) I/O error reading symbol table ***(Cromwell)***

The COD file has an invalid format in the specified record.

(620) filename index out of range in line number record ***(Cromwell)***

The COD file has an invalid value in the specified record.

(621) error writing ELF/DWARF section "*" on "*" ***(Cromwell)***

An error occurred writing the indicated section to the given file. Confirm the spelling and path of the file specified on the command line.

(622) too many type entries ***(Cromwell)***

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

(623) bad class in type hashing ***(Cromwell)***

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

(624) bad class in type compare ***(Cromwell)***

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

(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.

(627) missing "*" in SDB file "*" line * column * ***(Cromwell)***

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

(629) bad storage class "*" in SDB file "*" line * column * ***(Cromwell)***

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

(630) invalid syntax for prefix list in SDB file "*" ***(Cromwell)***

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

(631) syntax error at token "*" in SDB file "*" line * column * ***(Cromwell)***

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

(632) can't handle address size (*) ***(Cromwell)***

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

(633) unknown symbol class (*) ***(Cromwell)***

Cromwell has encountered a symbol class in the symbol table of a COFF, Microchip COFF, or ICOFF file which it can't identify.

(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 "*" on 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 "*" on line * *(Cromwell, Hexmate)*

A checksum error was found at the specified line in the specified Intel hex file. The HEX file may be corrupt.

(637) unknown prefix "*" in SDB file "*" *(Cromwell)*

This is an internal compiler warning. Contact HI-TECH Software technical support with details.

(638) version mismatch: 0x* expected *(Cromwell)*

The input Microchip COFF file wasn't produced using Cromwell.

(639) zero bit width in Microchip optional header *(Cromwell)*

The optional header in the input Microchip COFF file indicates that the program or data memory spaces are zero bits wide.

(668) prefix list did not match any SDB types *(Cromwell)*

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

(669) prefix list matched more than one SDB type *(Cromwell)*

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

(670) bad argument to -T *(Clist)*

The argument to the -T option to specify tab size was not present or correctly formed. The option expects a decimal interger argument.

(671) argument to -T should be in range 1 to 64 ***(Clist)***

The argument to the `-T` option to specify tab size was not in the expected range. The option expects a decimal interger argument ranging from 1 to 64 inclusive.

(673) missing filename after * option ***(Objtohex)***

The indicated option requires a valid file name. Ensure that the filename argument supplied to this option exists and is spelt correctly.

(674) too many references to "*" ***(Cref)***

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

(677) set_fact_bit on pic17! ***(Code Generator)***

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

(678) case 55 on pic17! ***(Code Generator)***

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

(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.

(681) bad common spec in -P option ***(Code Generator)***

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

(682) this architecture is not supported by the PICC Lite compiler ***(Code Generator)***

A target device other than baseline, midrange or highend was specified. This compiler only supports devices from these architecture families.

(683) bank 1 variables are not supported by the PICC Lite compiler *(Code Generator)*

A variable with an absolute address located in bank 1 was detected. This compiler does not support code generation of variables in this bank.

(684) bank 2 and 3 variables are not supported by the PICC Lite compiler *(Code Generator)*

A variable with an absolute address located in bank 2 or 3 was detected. This compiler does not support code generation of variables in these banks.

(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 information.

(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.

(690) interrupt function requires address *(Code Generator)*

The Highend PIC devices support multiple interrupts. An `@ address` is required with the interrupt definition to indicate with which vector this routine is associated, e.g.:

```
void interrupt isr(void) @ 0x10
{
  /* isr code goes here */
}
```

This construct is not required for midrange PIC devices.

(691) interrupt functions not implemented for 12 bit PIC ***(Code Generator)***

The 12-bit range of PIC processors do not support interrupts.

(692) interrupt function "*" may only have one interrupt level ***(Code Generator)***

Only one interrupt level may be associated with an `interrupt` function. Check to ensure that only one `interrupt_level` pragma has been used with the function specified. This pragma may be used more than once on main-line functions that are called from `interrupt` functions. For example:

```
#pragma interrupt_level 0
#pragma interrupt_level 1  /* which is it to be: 0 or 1? */
void interrupt isr(void)
{
```

(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  /* oops -- only 0 or 1 */
void interrupt isr(void)
{
  /* isr code goes here */
}
```

(694) no interrupt strategy available ***(Code Generator)***

The processor does not support saving and subsequent restoring of registers during an interrupt service routine.

(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;         /* oops --  this should be: static bit bb; */
  bb = (a > 66);
  return bb;
}
```

(699) no case labels in switch *(Code Generator)*

There are no `case` labels in this `switch` statement, e.g.:

```
switch(input) {
}               /* there is nothing to match the value of input */
```

(700) truncation of enumerated value *(Code Generator)*

An enumerated value larger than the maximum value supported by this compiler was detected and has been truncated, e.g.:

```
enum { ZERO, ONE, BIG=0x99999999 } test_case;
```

(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 information.

(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) bad which in expand() *(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.

(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) bad how in expand() *(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.

(713) bad initialization list *(Code Generator)*

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

(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 argument to -M option "*" *(Code Generator)*

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

(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) element count must be constant expression ***(Code Generator)***

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

(722) bad variable syntax in intermediate code ***(Code Generator)***

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

(723) function definitions 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 (*) in revlog() ***(Code Generator)***

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

(726) bad op "*" in uconval() ***(Code Generator)***

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

(727) bad op "*" in bconfloat() ***(Code Generator)***

This is an internal code generator error. Contact HI-TECH technical support with details.

(728) bad op "*" in confloat() ***(Code Generator)***

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

(729) bad op "*" in 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) initialization of bit types is illegal *(Code Generator)*

Variables of type `bit` cannot be initialised, e.g.:

```
bit b1 = 1; /* oops!
               b1 must be assigned after its definition */
```

(733) bad string "*" in pragma "psect" *(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.

(735) bad string "*" in pragma "stack_size" *(Code Generator)*

The argument to the stack_size pragma is malformed. This pragma must be followed by a number representing the maximum allowed stack size.

(737) unknown argument "*" to pragma "switch" *(Code Generator)*

The `#pragma switch` directive has been used with an invalid switch code generation method. Possible arguments are: `auto`, `simple` and `direct`.

(739) error closing output file *(Code Generator, Optimiser)*

The compiler detected an error when closing a file. Contact HI-TECH Support with details.

(740) zero dimension array is illegal *(Code Generator)*

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

(741) bitfield 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;   /* oops -- that's 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.

(743) divide by zero ***(Code Generator)***

An expression involving a division by zero has been detected in your code.

(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.

(746) object "*" qualified const, but not initialized *(Code Generator)*

An object has been qualified as `const`, but there is no initial value supplied at the definition. As this object cannot be written by the C program, this may imply the intial value was accidently omitted.

(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)          /* oops -- a has never been assigned a value */
    process();
}
```

(749) unknown register name "*" used with pragma *(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 reveals 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;  /* int to char conversion
               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;  /* oops -- that shifts the entire value out */
```

(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) divide 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;
  /* this can never be false;
     always perform the true statement */
  if(a == 4)
    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;
  /* this loop must iterate at least once */
  for(a=0; a!=10; a++)
    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;
/* this can never be false;
   always perform the true statement */
if(a = 4)
  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:

```
/* this can never be false;
   always perform the true statement */
if(a == 4)
    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.

(762) constant truncated when assigned to bitfield ***(Code Generator)***

A constant value is too large for a bitfield structure member to which it is being assigned, e.g.

```
struct INPUT {
  unsigned a : 3;
  unsigned b : 5;
} input_grp;
input_grp.a = 0x12;
/* 12h cannot fit into a 3-bit wide object */
```

(763) constant left operand to "? :" operator ***(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)      /* oops -- 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.

(767) constant truncated to bitfield width *(Code Generator)*

A constant value is too large for a bitfield structure member on which it is operating, e.g.

```
struct INPUT {
  unsigned a : 3;
  unsigned b : 5;
} input_grp;
input_grp.a |= 0x13;
/* 13h to large for 3-bit wide object */
```

(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.

(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.

(776) can't allocate space for object "*" (offs: *) *(Assembler)*

The assembler has run out of memory.

(777) can't allocate space for opnd structure within object "*", (offs: *) ***(Assembler)***

The assembler has run out of memory.

(780) too many psects defined ***(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" psects 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) can't handle "v_rtype" of * in copyexpr ***(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) end of file inside conditional *(Assembler)*

END-of-FILE was encountered while scanning for an "endif" to match a previous "if".

(793) unterminated macro argument *(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) use of LOCAL outside macros is illegal *(Assembler)*

The `LOCAL` directive is only legal inside macros. It defines local labels that will be unique for each invocation of the macro.

(797) syntax error in LOCAL argument *(Assembler)*

A symbol defined using the `LOCAL` assembler directive in an assembler macro is syntactically incorrect. Ensure that all symbols and all other assembler identifiers conform with the assembly language of the target device.

(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  ; oops --
            ; 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.

(804) write error on object file *(Assembler)*

The assembler failed to write to an object file. This may be an internal compiler error. Contact HI-TECH Software technical support with details.

(806) attempted to get an undefined object (*) *(Assembler)*

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

(807) attempted to set an undefined object (*) *(Assembler)*

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

(808) bad size in add_reloc() *(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.

(811) "cnt" too large (*) in display() *(Assembler)*

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

(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.

(816) duplicate ARCH specification 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.

(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.

(818) 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.

(819) 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.

(820) 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.

(821) duplicate INTSAVE for "*" in chipinfo file at line * ***(Assembler)***

The chipinfo file has a processor section with multiple INTSAVE values. Only one INTSAVE value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(822) 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.

(823) duplicate START for "*" in chipinfo file at line * ***(Assembler)***

The chipinfo file has a processor section with multiple START values. Only one START value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(824) 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.

(825) too many RAMBANK lines in chipinfo file for "*" ***(Assembler)***

The chipinfo file contains a processor section with too many RAMBANK fields. Reduce the number of values.

(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.

(827) too many COMMON lines in chipinfo file for "*" ***(Assembler)***

There are too many lines specifying common (access bank) memory in the chip configuration file.

(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.

(830) 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.

(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.

(841) bad source/destination for movfp/movpf instruction *(Assembler)*

The absolute address specified with the `movfp`/`movpf` instruction is too large.

(842) bad bit number *(Assembler, Optimiser)*

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

(843) a macro name can't 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  ; oops -- choose a different name to the macro
```

(844) lexical error *(Assembler, Optimiser)*

An unrecognized character or token has been seen in the input.

(845) symbol "*" defined more than once *(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:           ; oops -- 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.

(848) symbol has been declared EXTERN *(Assembler)*

An assembly label uses the same name as a symbol that has already been declared as `EXTERN`.

(849) illegal instruction for this processor *(Assembler)*

The instruction is not supported by this processor.

(850) PAGESEL not usable with this processor *(Assembler)*

The PAGESEL pseudo-instruction is not usable with the device selected.

(851) illegal destination *(Assembler)*

The destination (either `,f` or `,w`) is not correct for this instruction.

(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 */
```

(856) ALIGN argument must be a positive constant ***(Assembler)***

The `align` assembler directive requires a non-zero positive integer argument.

(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) argument to C option 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
```

(860) page width must be >= 49 ***(Assembler)***

The page width suboption to the `LIST` assembler directive must specify a with of at least 49.

(861) argument to N option 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 can't 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) argument to "size" psect flag 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 flag "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) argument to "reloc" psect flag 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 flag "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) argument to "delta" psect flag 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  ; negative delta value doesn't make sense
```

(869) psect flag "delta" redefined ***(Assembler)***

The 'DELTA' option of a psect has been redefined more than once in the same module.

(870) argument to "pad" psect flag must specify a positive constant ***(Assembler)***

The parameter to the PSECT assembler directive's 'PAD' option must be a non-zero positive integer.

(871) argument to "space" psect flag 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 flag "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.

(877) yacc stack overflow *(Assembler)*

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

(878) -S option used: "*" ignored *(Driver)*

The indicated assembly file has been supplied to the driver in conjunction with the `-S` option. The driver really has nothing to do since the file is already an assembly file.

(880) invalid number of parameters. Use "* –HELP" for help *(Driver)*

Improper command-line usage of the of the compiler's driver.

(881) setup succeeded *(Driver)*

The compiler has been successfully setup using the `--setup` driver option.

(883) setup failed *(Driver)*

The compiler was not successfully setup using the `--setup` driver option. Ensure that the directory argument to this option is spelt correctly, is syntactically correct for your host operating system and it exists.

(884) please ensure you have write permissions to the configuration file *(Driver)*

The compiler was not successfully setup using the `--setup` driver option because the driver was unable to access the XML configuration file. Ensure that you have write permission to this file. The driver will search the following configuration files in order:

- the file specified by the environment variable `HTC_XML`

- the file `/etc/htsoft.xml` if the directory '`/etc`' is writable and there is no `.htsoft.xml` file in your home directory
- the file `.htsoft.xml` file in your home directory

If none of the files can be located then the above error will occur.

(889) this * compiler has expired ***(Driver)***

The demo period for this compiler has concluded.

(890) contact HI-TECH Software to purchase and re-activate this compiler ***(Driver)***

The evaluation period of this demo installation of the compiler has expired. You will need to purchase the compiler to re-activate it. If however you sincerely believe the evaluation period has ended prematurely please contact HI-TECH technical support.

(891) can't open psect usage map file "*": * ***(Driver)***

The driver was unable to open the indicated file. The psect usage map file is generated by the driver when the driver option `--summary=file` is used. Ensure that the file is not open in another application.

(892) can't open memory usage map file "*": * ***(Driver)***

The driver was unable to open the indicated file. The memory usage map file is generated by the driver when the driver option `--summary=file` is used. Ensure that the file is not open in another application.

(893) can't open HEX usage map file "*": * ***(Driver)***

The driver was unable to open the indicated file. The HEX usage map file is generated by the driver when the driver option `--summary=file` is used. Ensure that the file is not open in another application.

(894) unknown source file type "*" ***(Driver)***

The extension of the indicated input file could not be determined. Only files with the extensions `as`, `c`, `obj`, `usb`, `p1`, `lib` or `hex` are identified by the driver.

(895) can't request and specify options in the one command ***(Driver)***

The usage of the driver options `--getoption` and `--setoption` is mutually exclusive.

(896) no memory ranges specified for data space ***(Driver)***

No on-chip or external memory ranges have been specified for the data space memory for the device specified.

(897) no memory ranges specified for program space ***(Driver)***

No on-chip or external memory ranges have been specified for the program space memory for the device specified.

(899) can't open option file "*" for application "*": * ***(Driver)***

An option file specified by a `--getoption` or `--setoption` driver option could not be opened. If you are using the `--setoption` option ensure that the name of the file is spelt correctly and that it exists. If you are using the `--getoption` option ensure that this file can be created at the given location or that it is not in use by any other application.

(900) exec failed: * ***(Driver)***

The subcomponent listed failed to execute. Does the file exist? Try re-installing the compiler.

(902) no chip name specified; use "* –CHIPINFO" to see available chip names ***(Driver)***

The driver was invoked without selecting what chip to build for. Running the driver with the –CHIPINFO option will display a list of all chips that could be selected to build for.

(904) illegal format specified in "*" option ***(Driver)***

The usage of this option was incorrect. Confirm correct usage with –HELP or refer to the part of the manual that discusses this option.

(905) illegal application specified in "*" option ***(Driver)***

The application given to this option is not understood or does not belong to the compiler.

(907) unknown memory space tag "*" in "*" option specification ***(Driver)***

A parameter to this memory option was a string but did not match any valid *tags*. Refer to the section of this manual that describes this option to see what tags (if any) are valid for this device.

(908) exit status = * ***(Driver)***

One of the subcomponents being executed encountered a problem and returned an error code. Other messages should have been reported by the subcomponent to explain the problem that was encountered.

(913) "*" option may cause compiler errors in some standard header files ***(Driver)***

Using this option will invalidate some of the qualifiers used in the standard header files resulting in errors. This issue and its solution are detailed in the section of this manual that specifically discusses this option.

(915) no room for arguments ***(Preprocessor, Parser, Code Generator, Linker, Objtohex)***

The code generator could not allocate any more memory.

(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.

(919) * in chipinfo file "*" at line * ***(Driver)***

The specified parameter in the chip configuration file is illegal.

(920) empty chipinfo file ***(Driver, Assembler)***

The chip configuration file was able to be opened but it was empty. Try re-installing the compiler.

(922) chip "*" not present in chipinfo file "*" ***(Driver)***

The chip selected does not appear in the compiler's chip configuration file. You may need to contact HI-TECH Software to see if support for this device is available or upgrade the version of your compiler.

(923) unknown suboption "*" ***(Driver)***

This option can take suboptions, but this suboption is not understood. This may just be a simple spelling error. If not, –HELP to look up what suboptions are permitted here.

(924) missing argument to "*" option ***(Driver)***

This option expects more data but none was given. Check the usage of this option.

(925) extraneous argument to "*" option ***(Driver)***

This option does not accept additional data, yet additional data was given. Check the usage of this option.

(926) duplicate "*" option ***(Driver)***

This option can only appear once, but appeared more than once.

(928) bad "*" option value ***(Driver, Assembler)***

The indicated option was expecting a valid hexadecimal integer argument.

(929) bad "*" option ranges ***(Driver)***

This option was expecting a parameter in a range format (*start_of_range-end_of_range*), but the parameter did not conform to this syntax.

(930) bad "*" option specification ***(Driver)***

The parameters to this option were not specified correctly. Run the driver with –HELP or refer to the driver's chapter in this manual to verify the correct usage of this option.

(931) command file not specified ***(Driver)***

Command file to this application, expected to be found after `'@'` or `'<'` on the command line was not found.

(939) no file arguments ***(Driver)***

The driver has been invoked with no input files listed on its command line. If you are getting this message while building through a third party IDE, perhaps the IDE could not verify the source files to compile or object files to link and withheld them from the command line.

(940) *-bit checksum * placed at * ***(Objtohex)***

Presenting the result of the requested checksum calculation.

(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.

(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.

(964) unpaired nibble in -FILL value will be truncated ***(Hexmate)***

The hexadecimal code given to the FILL option contained an incomplete byte. The incomplete byte (nibble) will be disregarded.

(965) -STRPACK option not yet implemented, option will be ignored ***(Hexmate)***

This option currently is not available and will be ignored.

(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 implies 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 identifier is already an enum tag ***(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 {             /* oops -- IN is already defined */
  int a, b;
};
```

(979) this identifier is already a struct tag ***(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};  /* oops -- IN is already defined */
```

(980) this identifier is already a union tag ***(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};  /* oops -- 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) unknown op "*" in nxtuse() ***(Optimiser,Assembler)***

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

(983) storage class redeclared ***(Parser)***

A variable previously declared as being *static*, has now be redeclared as *extern*.

(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;  /* oops -- what is the correct type? */
```

(985) qualifiers redeclared ***(Parser)***

This function or variable has different qualifiers in different declarations.

(986) enum member redeclared ***(Parser)***

A member of an enumeration is defined twice or more with differing values. Does the member appear twice in the same list or does the name of the member appear in more than one enum list?

(987) arguments redeclared ***(Parser)***

The data types of the parameters passed to this function do not match its prototype.

(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.

(993) some command-line options are disabled ***(Driver)***

The compiler is operating in demo mode. Some command-line options are disabled.

(994) some command-line options are disabled and compilation is delayed ***(Driver)***

The compiler is operating in demo mode. Some command-line options are disabled, the compilation speed will be slower.

(995) some command-line options are disabled, code size is limited to 16kB, compilation is delayed ***(Driver)***

The compiler is operating in demo mode. Some command-line options are disabled, the compilation speed will be slower, and the maximum allowed code size is limited to 16kB.

(1015) missing "*" specification in chipinfo file "*" at line * ***(Driver)***

This attribute was expected to appear at least once but was not defined for this chip.

(1016) missing argument* to "*" specification in chipinfo file "*" at line * ***(Driver)***

This value of this attribute is blank in the chip configuration file.

(1017) extraneous argument* to "*" specification in chipinfo file "*" at line * ***(Driver)***

There are too many attributes for the the listed specification in the chip configuration file.

(1018) illegal number of "*" specification* (* found; * expected) in chipinfo file "*" at line * ***(Driver)***

This attribute was expected to appear a certain number of times but it did not for this chip.

(1019) duplicate "*" specification in chipinfo file "*" at line * ***(Driver)***

This attribute can only be defined once but has been defined more than once for this chip.

(1020) unknown attribute "*" in chipinfo file "*" at line * ***(Driver)***

The chip configuration file contains an attribute that is not understood by this version of the compiler. Has the chip configuration file or the driver been replaced with an equivalent component from another version of this compiler?

(1021) syntax error reading "*" value in chipinfo file "*" at line * ***(Driver)***

The chip configuration file incorrectly defines the specified value for this device. If you are modifying this file yourself, take care and refer to the comments at the beginning of this file for a description on what type of values are expected here.

(1022) syntax error reading "*" range in chipinfo file "*" at line * ***(Driver)***

The chip configuration file incorrectly defines the specified range for this device. If you are modifying this file yourself, take care and refer to the comments at the beginning of this file for a description on what type of values are expected here.

(1024) syntax error in chipinfo file "*" at line * ***(Driver)***

The chip configuration file contains a syntax error at the line specified.

(1025) unknown architecture in chipinfo file "*" at line * ***(Driver)***

The attribute at the line indicated defines an architecture that is unknown to this compiler.

(1026) missing architecture in chipinfo file "*" at line * ***(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.

(1027) activation was successful ***(Driver)***

The compiler was successfully activated.

(1028) activation was not successful - error code (*) ***(Driver)***

The compiler did not activated successfully.

(1029) compiler not installed correctly - error code (*) ***(Driver)***

This compiler has failed to find any activation information and cannot proceed to execute. The compiler may have been installed incorrectly or incompletely. The error code quoted can help diagnose the reason for this failure. You may be asked for this failure code if contacting HI-TECH Software for assistance with this problem.

(1030) HEXMATE - Intel hex editing utility (Build 1.%i) ***(Hexmate)***

Indicating the version number of the Hexmate being executed.

(1031) USAGE: * [input1.hex] [input2.hex]... [inputN.hex] [options] ***(Hexmate)***

The suggested usage of Hexmate.

(1032) use –HELP=<option> for usage of these command line options ***(Hexmate)***

More detailed information is available for a specific option by passing that option to the HELP option.

(1033) available command-line options: ***(Hexmate)***

This is a simple heading that appears before the list of available options for this application.

(1034) type "*" for available options ***(Hexmate)***

It looks like you need help. This advisory suggests how to get more information about the options available to this application or the usage of these options.

(1035) bad argument count (*) ***(Parser)***

The number of arguments to a function is unreasonable. This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1036) bad "*" optional header length (0x* expected) ***(Cromwell)***

The length of the optional header in this COFF file was of an incorrect length.

(1037) short read on * ***(Cromwell)***

When reading the type of data indicated in this message, it terminated before reaching its specified length.

(1038) string table length too short ***(Cromwell)***

The specified length of the COFF string table is less than the minimum.

(1039) inconsistent symbol count ***(Cromwell)***

The number of symbols in the symbol table has exceeded the number indicated in the COFF header.

(1040) bad checksum: record 0x*, checksum 0x* ***(Cromwell)***

A record of the type specified failed to match its own checksum value.

(1041) short record ***(Cromwell)***

While reading a file, one of the file's records ended short of its specified length.

(1042) unknown * record type 0x* ***(Cromwell)***

The type indicator of this record did not match any valid types for this file format.

(1043) unknown optional header ***(Cromwell)***

When reading this Microchip COFF file, the optional header within the file header was of an incorrect length.

(1044) end of file encountered ***(Cromwell, Linker)***

The end of the file was found while more data was expected. Has this input file been truncated?

(1045) short read on block of * bytes ***(Cromwell)***

A while reading a block of byte data from a UBROF record, the block ended before the expected length.

(1046) short string read ***(Cromwell)***

A while reading a string from a UBROF record, the string ended before the specified length.

(1047) bad type byte for UBROF file ***(Cromwell)***

This UBROF file did not begin with the correct record.

(1048) bad time/date stamp ***(Cromwell)***

This UBROF file has a bad time/date stamp.

(1049) wrong CRC on 0x* bytes; should be * ***(Cromwell)***

An end record has a mismatching CRC value in this UBROF file.

(1050) bad date in 0x52 record ***(Cromwell)***

A debug record has a bad date component in this UBROF file.

(1051) bad date in 0x01 record ***(Cromwell)***

A start of program record or segment record has a bad date component in this UBROF file.

(1052) unknown record type ***(Cromwell)***

A record type could not be determined when reading this UBROF file.

(1053) additional RAM ranges larger than bank size ***(Driver)***

A block of additional RAM being requested exceeds the size of a bank. Try breaking the block into multiple ranges that do not cross bank boundaries.

(1054) additional RAM range out of bounds ***(Driver)***

The RAM memory range as defined through custom RAM configuration is out of range.

(1055) RAM range out of bounds (*) ***(Driver)***

The RAM memory range as defined in the chip configuration file or through custom configuration is out of range.

(1056) unknown chip architecture ***(Driver)***

The compiler is attempting to compile for a device of an architecture that is either unsupported or disabled.

(1057) fast double option only available on 17 series processors ***(Driver)***

The fast double library cannot be selected for this device. These routines are only available for PIC17 devices.

(1058) assertion ***(Code Generator)***

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

(1059) rewrite loop ***(Code Generator)***

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

(1081) static initialization of persistent variable "*" ***(Parser, Code Generator)***

A persistent variable has been assigned an initial value. This is somewhat contradictory as the initial value will be assigned to the variable during execution of the compiler's startup code, however the *persistent* qualifier requests that this variable shall be unchanged by the compiler's startup code.

(1082) size of initialized array element is zero ***(Code Generator)***

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

(1088) function pointer "*" is used but never assigned a value ***(Code Generator)***

A function call involving a function pointer was made, but the pointer was never assigned a target address, e.g.:

```
void (*fp)(int);
fp(23);     /* oops -- what function does fp point to? */
```

(1089) recursive function call to "*" ***(Code Generator)***

A recursive call to the specified function has been found. The call may be direct or indirect (using function pointers) and may be either a function calling itself, or calling another function whose call graph includes the function under consideration.

(1090) variable "*" is not used ***(Code Generator)***

This variable is declared but has not been used by the program. Consider removing it from the program.

(1091) main function "*" not defined ***(Code Generator)***

The *main* function has not been defined. Every C program must have a function called *main*.

(1094) bad derived type ***(Code Generator)***

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

(1095) bad call to typeSub() ***(Code Generator)***

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

(1096) type should be unqualified ***(Code Generator)***

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

(1097) unknown type string "*" ***(Code Generator)***

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

(1098) conflicting declarations for variable "*" (*:*) ***(Parser, Code Generator)***

Differing type information has been detected in the declarations for a variable, or between a declaratin and the definition of a variable, e.g.:

```
extern long int test;
int test;    /* oops -- which is right? int or long int ? */
```

(1104) unqualified error ***(Code Generator)***

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

(1118) bad string "*" in getexpr(J) ***(Code Generator)***

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

(1119) bad string "*" in getexpr(LRN) ***(Code Generator)***

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

(1121) expression error ***(Code Generator)***

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

(1137) match() error: * ***(Code Generator)***

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

(1157) W register must be W9 ***(Assembler)***

The working register required here has to be W9, but an other working register was selected.

(1159) W register must be W11 ***(Assembler)***

The working register required here has to be W11, but an other working register was selected.

(1178) the "*" option has been removed and has no effect ***(Driver)***

This option no longer exists in this version of the compiler and has been ignored. Use the compiler's *–help* option or refer to the manual to find a replacement option.

(1179) interrupt level for function "*" may not exceed * ***(Code Generator)***

The interrupt level for the function specified is too high. Each interrupt function is assigned a unique interrupt level. This level is considered when analysing the call graph and re-entrantly called functions. If using the `interrupt_level` pragma, check the value specified.

(1180) directory "*" does not exist ***(Driver)***

The directory specified in the setup option does not exist. Create the directory and try again.

(1182) near variables must be global or static ***(Code Generator)***

A variable qualified as *near* must also be qualified with *static* or made global. An auto variable cannot be qualified as *near*.

(1183) invalid version number ***(Activation)***

During activation, no matching version number was found on the HI-TECH activation server database for the serial number specified.

(1184) activation limit reached ***(Activation)***

The number of activations of the serial number specified has exceeded the maximum number allowed for the license.

(1185) invalid serial number ***(Activation)***

During activation, no matching serial number was found on the HI-TECH activation server database.

(1186) licence has expired ***(Driver)***

The time-limited license for this compiler has expired.

(1187) invalid activation request ***(Driver)***

The compiler has not been correctly activated.

(1188) network error * ***(Activation)***

The compiler activation software was unable to connect to the HI-TECH activation server via the network.

(1190) FAE license only - not for use in commercial applications ***(Driver)***

Indicates that this compiler has been activated with an FAE licence. This licence does not permit the product to be used for the development of commercial applications.

(1191) licensed for educational use only ***(Driver)***

Indicates that this compiler has been activated with an education licence. The educational licence is only available to educational facilities and does not permit the product to be used for the development of commercial applications.

(1192) licensed for evaluation purposes only ***(Driver)***

Indicates that this compiler has been activated with an evaluation licence.

(1193) this licence will expire on * ***(Driver)***

The compiler has been installed as a time-limited trial. This trial will end on the date specified.

(1195) invalid syntax for "*" option ***(Driver)***

A command line option that accepts additional parameters was given inappropriate data or insufficient data. For example an option may expect two parameters with both being integers. Passing a string as one of these parameters or supplying only one parameter could result in this error.

(1198) too many "*" specifications; * maximum ***(Hexmate)***

This option has been specified too many times. If possible, try performing these operations over several command lines.

(1199) compiler has not been activated ***(Driver)***

The trial period for this compiler has expired. The compiler is now inoperable until activated with a valid serial number. Contact HI-TECH Software to purchase this software and obtain a serial number.

(1200) Found %0*lXh at address *h ***(Hexmate)***

The code sequence specified in a -FIND option has been found at this address.

(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 length of data was given to this option. The value provided to this option exceeds the maximum or minimum bounds required by this option.

(1205) using the configuration file *; you may override this with the environment variable HTC_XML ***(Driver)***

This is the compiler configuration file selected during compiler setup. This can be changed via the HTC_XML environment variable. This file is used to determine where the compiler has been installed.

(1207) some of the command line options you are using are now obsolete ***(Driver)***

Some of the command line options passed to the driver have now been discontinued in this version of the compiler, however during a grace period these old options will still be processed by the driver.

(1208) use –help option or refer to the user manual for option details ***(Driver)***

An obsolete option was detected. Use –help or refer to the manual to find a replacement option that will not result in this advisory message.

(1209) An old MPLAB tool suite plug-in was detected. ***(Driver)***

The options passed to the driver resemble those that the Microchip MPLAB IDE would pass to a previous version of this compiler. Some of these options are now obsolete, however they were still interpreted. It is recommended that you install an updated HI-TECH options plug-in for the MPLAB IDE.

(1210) Visit the HI-TECH Software website (www.htsoft.com) for a possible update ***(Driver)***

Visit our website to see if an update is available to address the issue(s) listed in the previous compiler message. Please refer to the on-line self-help facilities such as the *Frequently asked Questions* or search the *On-line forums*. In the event of no details being found here, contact HI-TECH Software for further information.

(1212) Found * (%0*lXh) at address *h ***(Hexmate)***

The code sequence specified in a -FIND option has been found at this address.

(1213) 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.

(1218) 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.

(1228) unable to locate installation directory ***(Driver)***

The compiler cannot determine the directory where it has been installed.

(1230) dereferencing uninitialized pointer "*" ***(Code Generator)***

A pointer that has not yet been assigned a value has been dereferenced. This can result in erroneous behaviour at runtime.

(1235) unknown keyword * ***(Driver)***

The token contained in the USB descriptor file was not recognised.

(1236) invalid argument to *: * ***(Driver)***

An option that can take additional parameters was given an invalid parameter value. Check the usage of the option or the syntax or range of the expected parameter.

(1237) endpoint 0 is pre-defined ***(Driver)***

An attempt has been made to define endpoint 0 in a USB file. This channel c

(1238) FNALIGN failure on * ***(Linker)***

Two functions have their auto/parameter blocks aligned using the `FNALIGN` directive, but one function calls the other, which implies that must not be aligned. This will occur if a function pointer is assigned the address of each function, but one function calls the other. For example:

```
int one(int a) { return a; }
int two(int a) { return two(a)+2; }  /* ! */
int (*ip)(int);
ip = one;
ip(23);
ip = two;      /* ip references one and two; two calls one */
ip(67);
```

(1239) pointer * has no valid targets ***(Code Generator)***

A function call involving a function pointer was made, but the pointer was never assigned a target address, e.g.:

```
void (*fp)(int);
fp(23);     /* oops -- what function does fp point to? */
```

(1240) unknown checksum algorithm type (%i) ***(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.

(1241) bad start address in * ***(Driver)***

The start of range address for the `--CHECKSUM` option could not be read. This value must be a hexadecimal number.

(1242) bad end address in * *(Driver)*

The end of range address for the `--CHECKSUM` option could not be read. This value must be a hexadecimal number.

(1243) bad destination address in * *(Driver)*

The destination address for the `--CHECKSUM` option could not be read. This value must be a hexadecimal number.

(1245) value greater than zero required for * *(Hexmate)*

The *align* operand to the HEXMATE `-FIND` option must be positive.

(1246) no RAM defined for variable placement *(Code Generator)*

No memory has been specified to cover the banked RAM memory.

(1247) no access RAM defined for variable placement *(Code Generator)*

No memory has been specified to cover the access bank memory.

(1248) symbol (*) encountered with undefined type size *(Code Generator)*

The code generator was asked to position a variable, but the size of the variable is not known. This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1250) could not find space (* byte*) for variable * *(Code Generator)*

The code generator could not find space in the banked RAM for the variable specified.

(1253) could not find space (* byte*) for auto/param block *(Code Generator)*

The code generator could not find space in RAM for the psect that holds `auto` and parameter variables.

(1254) could not find space (* byte*) for data block *(Code Generator)*

The code generator could not find space in RAM for the data psect that holds initialised variables.

(1255) conflicting paths for output directory *(Driver)*

The compiler has been given contradictory paths for the output directory via any of the `-O` or `--OUTDIR` options, e.g.

```
--outdir=../../   -o../main.hex
```

(1256) undefined symbol "*" treated as hex constant *(Assembler)*

A token which could either be interpreted as a symbol or a hexadecimal value does not match any previously defined symbol and so will be interpreted as the latter. Use a leading *zero* to avoid the ambiguity, or use an alternate radix sepcifier such as `0x`. For example:

```
mov  a, F7h  ; is this the symbol F7h, or the hex number 0xF7?
```

(1257) local variable "*" is used but never given a value *(Code Generator)*

An `auto` variable has been defined and used in an expression, but it has not been assigned a value in the C code before its first use. Auto variables are not cleared on startup and their initial value is undefined. For example:

```
void main(void) {
  double src, out;
  out = sin(src);   /* oops -- what value was in src? */
```

(1258) possible stack overflow when calling function "*" *(Code Generator)*

The call tree analysis by the code generator indicates that the hardware stack may overflow. This should be treated as a guide only. Interrupts, the assembler optimizer and the program structure may affect the stack usage. The stack usuage is based on the C program and does not include any call tree derived from assembly code.

(1259) can't optimize for both speed and space *(Driver)*

The driver has been given contradictory options of compile for speed and compile for space, e.g.

```
--opt=speed,space
```

(1260) macro "*" redefined ***(Assembler)***

More than one definition for a macro with the same name has been encountered, e.g.

```
MACRO fin
  ret
ENDM
MACRO fin   ; oops -- was this meant to be a different macro?
  reti
ENDM
```

(1261) string constant required ***(Assembler)***

A string argument is required with the `DS` or `DSU` directive, e.g.

```
DS ONE     ; oops -- did you mean DS "ONE"?
```

(1264) unsafe pointer conversion ***(Code Generator)***

A pointer to one kind of structure has been converted to another kind of structure and the structures do not have a similar definition, e.g.

```
struct ONE {
  unsigned a;
  long b;          /* ! */
} one;
struct TWO {
  unsigned a;
  unsigned b;      /* ! */
} two;
struct ONE * oneptr;
oneptr = & two;    /* oops --
                   was ONE meant to be same struct as TWO? */
```

(1267) fixup overflow referencing * into * bytes at 0x* ***(Linker)***

See the following error message (1268) for more information..

(1268) fixup overflow storing 0x* in * bytes at * ***(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.

(0) 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
```

(0) incomplete ident record ***(Libr)***

The IDENT record in the object file was incomplete. Contact HI-TECH Support with details.

(0) incomplete symbol record ***(Libr)***

The SYM record in the object file was incomplete. Contact HI-TECH Support with details.

(0) library file names should have .lib extension: * ***(Libr)***

Use the `.lib` extension when specifying a library filename.

(0) 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.

(0) 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.

Appendix C

Chip Information

The following table lists all devices currently supported by HI-TECH C PRO for the PIC10/12/16 MCU Family.

Table C.1: Devices supported by HI-TECH C PRO for the PIC10/12/16 MCU Family

DEVICE	ARCH	ROMSIZE	RAMBANK	EEPROMSIZE
10F200	PIC12	100	10-1F	
10F202	PIC12	200	08-1F	
10F204	PIC12	100	10-1F	
10F206	PIC12	200	08-1F	
10F220	PIC12	100	10-1F	
10F222	PIC12	200	09-1F	
12C508	PIC12	200	07-1F	
12F508	PIC12	200	07-1F	
12C509	PIC12	400	07-1F,30-3F	
12F509	PIC12	400	07-1F,30-3F	
12F510	PIC12	400	0A-1F,30-3F	
12F519	PIC12	400	07-1F,30-3F	
12C508A	PIC12	200	07-1F	
12C509A	PIC12	400	07-1F,30-3F	
12C509AG	PIC12	400	07-1F,30-3F	
RF509AG	PIC12	400	07-1F,30-3F	
12C509AF	PIC12	400	07-1F,30-3F	
RF509AF	PIC12	400	07-1F,30-3F	
12CR509A	PIC12	400	07-1F,30-3F	
12CE518	PIC12	200	07-1F	
12CE519	PIC12	400	07-1F,30-3F	
16C505	PIC12	400	08-1F,30-3F,50-5F,70-7F	
16F505	PIC12	400	08-1F,30-3F,50-5F,70-7F	

continued...

Table C.1: Devices supported by HI-TECH C PRO for the PIC10/12/16 MCU Family

DEVICE	ARCH	ROMSIZE	RAMBANK	EEPROMSIZE
16F506	PIC12	400	0D-1F,30-3F,50-5F,70-7F	
16F526	PIC12	400	0D-1F,30-3F,50-5F,70-7F	
16C52	PIC12	180	07-1F	
16C54	PIC12	200	07-1F	
16CR54A	PIC12	200	07-1F	
16CR54B	PIC12	200	07-1F	
16CR54C	PIC12	200	07-1F	
16HV540	PIC12	200	08-1F	
16C54A	PIC12	200	07-1F	
16C54B	PIC12	200	07-1F	
16C54C	PIC12	200	07-1F	
16F54	PIC12	200	07-1F	
16C55	PIC12	200	08-1F	
16C55A	PIC12	200	08-1F	
16C56	PIC12	400	07-1F	
16C56A	PIC12	400	07-1F	
16CR56A	PIC12	400	07-1F	
16C57	PIC12	800	08-1F,30-3F,50-5F,70-7F	
16C57C	PIC12	800	08-1F,30-3F,50-5F,70-7F	
16CR57B	PIC12	800	08-1F,30-3F,50-5F,70-7F	
16CR57C	PIC12	800	08-1F,30-3F,50-5F,70-7F	
16F57	PIC12	800	08-1F,30-3F,50-5F,70-7F	
16C58	PIC12	800	07-1F,30-3F,50-5F,70-7F	
16C58A	PIC12	800	07-1F,30-3F,50-5F,70-7F	
16C58B	PIC12	800	07-1F,30-3F,50-5F,70-7F	
16CR58A	PIC12	800	07-1F,30-3F,50-5F,70-7F	
16CR58B	PIC12	800	07-1F,30-3F,50-5F,70-7F	
16F59	PIC12	800	90-9F,B0-BF,D0-DF,E0-EF	
MCV08A	PIC12	400	0A-1F,30-3F	
MCV14A	PIC12	400	0D-1F,30-3F,50-5F,70-7F	
MCV18A	PIC12	200	07-1F	
MCV28A	PIC12	800	08-1F,30-3F,50-5F,70-7F	
12F609	PIC14	400	40-7F	
12HV609	PIC14	400	40-7F	
12F615	PIC14	400	40-7F	
12HV615	PIC14	400	40-7F	
12F629	PIC14	3FF	20-5F	80
12F635	PIC14	400	40-7F	80
12C671	PIC14	3FF	20-7F,A0-BF	
12C672	PIC14	7FF	20-7F,A0-BF	
12CE673	PIC14	3FF	20-7F,A0-BF	
12CE674	PIC14	7FF	20-7F,A0-BF	
12F675	PIC14	3FF	20-5F	80
12F675F	PIC14	3FF	20-5F	80
12F675H	PIC14	3FF	20-5F	80

continued...

Table C.1: Devices supported by HI-TECH C PRO for the PIC10/12/16 MCU Family

DEVICE	ARCH	ROMSIZE	RAMBANK	EEPROMSIZE
12F675K	PIC14	3FF	20-5F	80
12F683	PIC14	800	20-7F,A0-BF	100
14000	PIC14	FC0	20-7F,A0-FF	
16C432	PIC14	800	20-7F,A0-BF	
16C433	PIC14	800	20-7F,A0-BF	
16C554	PIC14	200	20-6F	
16C554A	PIC14	200	20-6F	
16C556	PIC14	400	20-6F	
16C556A	PIC14	400	20-6F	
16C557	PIC14	800	20-7F,A0-BF	
16C558	PIC14	800	20-7F,A0-BF	
16C558A	PIC14	800	20-7F,A0-BF	
16C61	PIC14	400	0C-2F	
16C62	PIC14	800	20-7F,A0-BF	
16C62A	PIC14	800	20-7F,A0-BF	
16C62B	PIC14	800	20-7F,A0-BF	
16CR62	PIC14	800	20-7F,A0-BF	
16C63	PIC14	1000	20-7F,A0-FF	
16C63A	PIC14	1000	20-7F,A0-FF	
16CR63	PIC14	1000	20-7F,A0-FF	
16C64	PIC14	800	20-7F,A0-BF	
16C64A	PIC14	800	20-7F,A0-BF	
16CR64	PIC14	800	20-7F,A0-BF	
16C65	PIC14	1000	20-7F,A0-FF	
16CR65	PIC14	1000	20-7F,A0-FF	
16C65A	PIC14	1000	20-7F,A0-FF	
16C65B	PIC14	1000	20-7F,A0-FF	
16C66	PIC14	2000	20-7F,A0-EF,110-16F,190-1EF	
16C67	PIC14	2000	20-7F,A0-EF,110-16F,190-1EF	
16C620	PIC14	200	20-6F	
16C620A	PIC14	200	20-7F	
16CR620A	PIC14	200	20-7F	
16C621	PIC14	400	20-6F	
16C621A	PIC14	400	20-7F	
16C622	PIC14	800	20-7F,A0-BF	
16C622A	PIC14	800	20-7F,A0-BF	
16CE623	PIC14	200	20-7F	
16CE624	PIC14	400	20-7F	
16CE625	PIC14	800	20-7F,A0-BF	
16F610	PIC14	400	40-7F	
16HV610	PIC14	400	40-7F	
16F616	PIC14	800	A0-BF	
16HV616	PIC14	800	A0-BF	
16F630	PIC14	3FF	20-5F	80
16F631	PIC14	400	40-7F	80

continued...

Table C.1: Devices supported by HI-TECH C PRO for the PIC10/12/16 MCU Family

DEVICE	ARCH	ROMSIZE	RAMBANK	EEPROMSIZE
16F636	PIC14	800	20-7F,A0-BF	100
16F639	PIC14	800	20-7F,A0-BF	100
16C641	PIC14	800	20-7F,A0-BF	
16C642	PIC14	1000	20-7F,A0-EF	
16C661	PIC14	800	20-7F,A0-BF	
16C662	PIC14	1000	20-7F,A0-EF	
16F676	PIC14	3FF	20-5F	80
16F677	PIC14	800	20-7F,A0-BF	100
16F684	PIC14	800	20-7F,A0-BF	100
16F685	PIC14	1000	20-7F,A0-EF,120-16F	100
16F687	PIC14	800	20-7F,A0-BF	100
16F688	PIC14	1000	20-7F,A0-EF,120-16F	100
16F689	PIC14	1000	20-7F,A0-EF,120-16F	100
16F690	PIC14	1000	20-7F,A0-EF,120-16F	100
16C710	PIC14	200	0C-2F	
16C71	PIC14	400	0C-2F	
16C711	PIC14	400	0C-4F	
16C712	PIC14	400	20-7F,A0-BF	
16C715	PIC14	800	20-7F,A0-BF	
16C716	PIC14	800	20-7F,A0-BF	
16C717	PIC14	800	20-7F,A0-EF,120-16F	
16C72	PIC14	800	20-7F,A0-BF	
16C72A	PIC14	800	20-7F,A0-BF	
16CR72	PIC14	800	20-7F,A0-BF	
16F72	PIC14	800	20-7F,A0-BF	
16C73	PIC14	1000	20-7F,A0-FF	
16F73	PIC14	1000	20-7F,A0-FF	
16F722	PIC14	800	20-7F,A0-BF	
16LF722	PIC14	800	20-7F,A0-BF	
16F723	PIC14	1000	20-7F,A0-EF,120-12F	
16LF723	PIC14	1000	20-7F,A0-EF,120-12F	
16F724	PIC14	1000	20-7F,A0-EF,120-12F	
16LF724	PIC14	1000	20-7F,A0-EF,120-12F	
16F726	PIC14	2000	20-7F,A0-EF,110-16F,190-1EF	
16LF726	PIC14	2000	20-7F,A0-EF,110-16F,190-1EF	
16F727	PIC14	2000	20-7F,A0-EF,110-16F,190-1EF	
16LF727	PIC14	2000	20-7F,A0-EF,110-16F,190-1EF	
16F737	PIC14	1000	20-7F,A0-EF,110-16F,190-1EF	
16C73A	PIC14	1000	20-7F,A0-FF	
16C73B	PIC14	1000	20-7F,A0-FF	
16C74	PIC14	1000	20-7F,A0-FF	
16F74	PIC14	1000	20-7F,A0-FF	
16F747	PIC14	1000	20-7F,A0-EF,110-16F,190-1EF	
16C74A	PIC14	1000	20-7F,A0-FF	
16C74B	PIC14	1000	20-7F,A0-FF	

continued...

Table C.1: Devices supported by HI-TECH C PRO for the PIC10/12/16 MCU Family

DEVICE	ARCH	ROMSIZE	RAMBANK	EEPROMSIZE
16LC74B	PIC14	1000	20-7F,A0-FF	
16C76	PIC14	2000	20-7F,A0-EF,110-16F,190-1EF	
16F76	PIC14	2000	20-7F,A0-EF,110-16F,190-1EF	
16F767	PIC14	2000	20-7F,A0-EF,110-16F,190-1EF	
16C77	PIC14	2000	20-7F,A0-EF,110-16F,190-1EF	
16F77	PIC14	2000	20-7F,A0-EF,110-16F,190-1EF	
16F777	PIC14	2000	20-7F,A0-EF,110-16F,190-1EF	
16C770	PIC14	800	20-7F,A0-EF,120-16F	
16C771	PIC14	1000	20-7F,A0-EF,120-16F	
16C773	PIC14	1000	20-7F,A0-EF,120-16F	
16C774	PIC14	1000	20-7F,A0-EF,120-16F	
16C745	PIC14	2000	20-7F,A0-EF,120-16F	
16C765	PIC14	2000	20-7F,A0-EF,120-16F	
16C781	PIC14	400	20-7F,A0-BF	
16C782	PIC14	800	20-7F,A0-BF	
16F785	PIC14	800	20-7F,A0-BF	100
16HV785	PIC14	800	20-7F,A0-BF	100
16F818	PIC14	400	20-7F,A0-BF	80
16F819	PIC14	800	20-7F,A0-EF,120-16F	100
16F83	PIC14	200	0C-2F	40
16CR83	PIC14	200	0C-2F	40
16C84	PIC14	400	0C-2F	40
16F84	PIC14	400	0C-4F	40
16F84A	PIC14	400	0C-4F	40
16CR84	PIC14	400	0C-4F	40
16F627	PIC14	400	20-7F,A0-EF,120-14F	80
16F627A	PIC14	400	20-7F,A0-EF,120-14F	80
16F628	PIC14	800	20-7F,A0-EF,120-14F	80
16F628A	PIC14	800	20-7F,A0-EF,120-14F	80
16F648A	PIC14	1000	20-7F,A0-EF,120-16F	100
16F716	PIC14	800	20-7F,A0-BF	
16F87	PIC14	1000	20-7F,A0-EF,110-16F,190-1EF	100
16F870	PIC14	800	20-7F,A0-BF	40
16F871	PIC14	800	20-7F,A0-BF	40
16F872	PIC14	800	20-7F,A0-BF	40
16F873	PIC14	1000	20-7F,A0-FF	80
16F873A	PIC14	1000	20-7F,A0-FF	80
16F874	PIC14	1000	20-7F,A0-FF	80
16F874A	PIC14	1000	20-7F,A0-FF	80
16F876	PIC14	2000	20-7F,A0-EF,110-16F,190-1EF	100
16F876A	PIC14	2000	20-7F,A0-EF,110-16F,190-1EF	100
16F877	PIC14	2000	20-7F,A0-EF,110-16F,190-1EF	100
16F877A	PIC14	2000	20-7F,A0-EF,110-16F,190-1EF	100
16F88	PIC14	1000	20-7F,A0-EF,110-16F,190-1EF	100
16F882	PIC14	800	20-7F,A0-BF	80

continued...

Table C.1: Devices supported by HI-TECH C PRO for the PIC10/12/16 MCU Family

DEVICE	ARCH	ROMSIZE	RAMBANK	EEPROMSIZE
16F883	PIC14	1000	20-7F,A0-EF,120-16F	100
16F884	PIC14	1000	20-7F,A0-EF,120-16F	100
16F886	PIC14	2000	20-7F,A0-EF,110-16F,190-1EF	100
16F887	PIC14	2000	20-7F,A0-EF,110-16F,190-1EF	100
16F913	PIC14	1000	20-7F,A0-EF,120-16F	100
16F914	PIC14	1000	20-7F,A0-EF,120-16F	100
16F916	PIC14	2000	20-7F,A0-EF,120-16F,190-1EF	100
16F917	PIC14	2000	20-7F,A0-EF,120-16F,190-1EF	100
16C923	PIC14	1000	20-7F,A0-EF	
16C924	PIC14	1000	20-7F,A0-EF	
16C925	PIC14	1000	20-7F,A0-EF	
16C926	PIC14	2000	20-7F,A0-EF,120-16F,1A0-1EF	
16F946	PIC14	2000	20-7F,A0-EF,120-16F,1A0-1EF	100
16F1933	PIC14E	1000	20-7F,A0-EF,120-16F	100
16LF1933	PIC14E	1000	20-7F,A0-EF,120-16F	100
16F1934	PIC14E	1000	20-7F,A0-EF,120-16F	100
16LF1934	PIC14E	1000	20-7F,A0-EF,120-16F	100
16F1936	PIC14E	2000	20-7F,A0-EF,120-16F,1A0-1EF,220-26F,2A0-2EF,320-32F	100
16LF1936	PIC14E	2000	20-7F,A0-EF,120-16F,1A0-1EF,220-26F,2A0-2EF,320-32F	100
16F1937	PIC14E	2000	20-7F,A0-EF,120-16F,1A0-1EF,220-26F,2A0-2EF,320-32F	100
16LF1937	PIC14E	2000	20-7F,A0-EF,120-16F,1A0-1EF,220-26F,2A0-2EF,320-32F	100

Index

PICC 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
--APBANK	Specify Bank for Autos and Parameters
--ASMLIST	Generate assembler .LST file for each compilation
--ADDRQUAL=*action*	Select compiler response to bank qualifiers in source
--CALLGRAPH<=*argument*>	Style of call graph listing in map file
--CHECKSUM=*start-end@dest*	Calculate a checksum over an address range
--CHIP=*processor*	Selects which processor to compile for
--CHIPINFO	Displays a list of supported processors
--CODEOFFSET=*address*	Offset program code to address
--CR=*file*	Generate cross-reference listing
--DEBUGGER=*type*	Select the debugger that will be used
--DOUBLE=*type*	Selects size/kind of double types
--ECHO	Echo command line
--ERRFORMAT<=*format*>	Format error message strings to the given style
--ERRORS=*number*	Sets the maximum number of errors displayed
--FILL=*opcode*	Fill unused program locations with this hexadecimal code
--FLOAT<=*argument*>	Size of float type
continued...	

PICC 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
`--MEMMAP=`*`file`*	Display memory summary information for the map file
`--MODE=`*`mode`*	Specify compiler operating mode
`--MSGDISABLE<=`*`argument`*`>`	Disable these warning or advisory messages
`--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
`--OBJDIR=`*`argument`*	Object and intermediate files directory
`--OPT<=`*`type`*`>`	Enable general compiler optimizations
`--OUTDIR`	Specify output files directory
`--OUTPUT=`*`type`*	Generate output file type
`--PASS1`	Stop after .p1 file generation
`--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
`--TIME`	Show execution time in each stage of build process
`--VER`	Display the compiler's version number
`--WARN=`*`level`*	Set the compiler's warning level
continued...	

PICC Command-line Options

Option	**Meaning**
`--WARNFORMAT=`*`format`*	Format warning message strings to given style