

2

Using the Compiler

This chapter describes how to use the High C/C++ driver and how to customize the compilation process. It contains the following sections:

§2.1: *The Build Process*

§2.2: *High C/C++ Driver Overview*

§2.3: *Using the High C/C++ Driver*

§2.4: *Linking Run-Time Libraries*

§2.5: *Customizing the Compilation*

2.1 The Build Process

To create a load module or an executable program, you link one or more object modules generated by the compiler and the ELF assembler with MetaWare run-time libraries.

The process of compiling and linking a program is sometimes referred to as *building* a program. The build process is illustrated in Figure 2.1. This chapter describes that process and explains how to build a High C/C++ program.

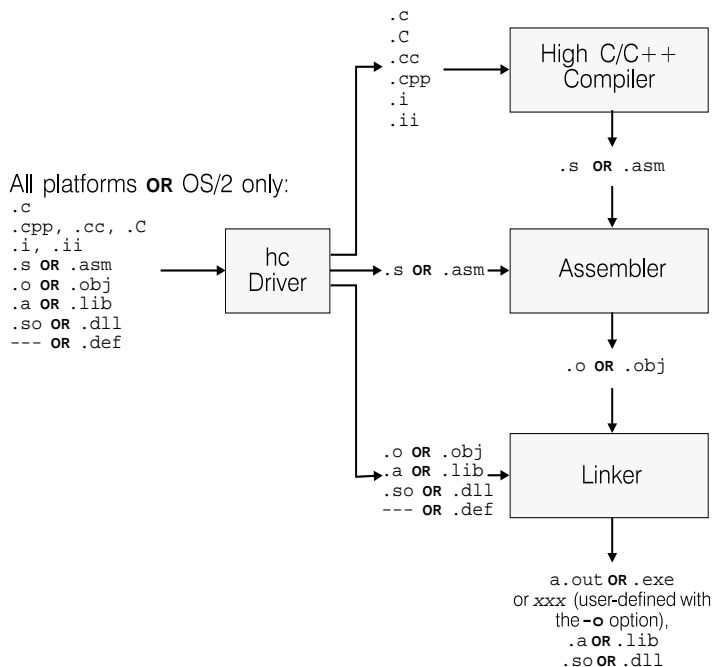


Figure 2.1 Building a Program

As Figure 2.1 illustrates, any assembly-language files (`.s` or `.asm` files) that appear on the command line go directly to the assembler. The assembler, like the compiler, produces object modules and sends them to the linker.

If you suppress linking, your program is compiled into one or more relocatable object modules. By default, all object files generated during the build process — including the load module generated by the linker — are placed in the current working directory. (For information on how to direct object files to another directory, see the description of `-Hobjdir` in Chapter 3: *Using Compiler Options*.)

2.2 High C/C++ Driver Overview

The High C/C++ compiler is distributed with a *driver program* that you can use to compile and link your program in one step. You can also use the driver to separately invoke the compiler or linker.

Advantages of using the driver

Using the driver to compile and link your program in one command has several advantages. Based on default settings and any (optional) compiler controls you use, the driver knows exactly which libraries to include in the link. This driver-controlled linking significantly decreases the possibility of generating undefined or multiply defined symbol errors. The driver also handles default linker settings, duplicate compiler/linker options, and linker options.

The following sections describe the driver syntax and its various elements.

2.2.1 Driver Syntax

This is the syntax of the High C/C++ driver command-line:

```
driver_cmd [options] scfile... [@arg_file...]
```

where the components of the command line have the following meanings:

<i>driver_cmd</i>	High C/C++ driver command
<i>options</i>	Compiler options
<i>scfile</i>	Source-file name
<i>arg_file</i>	ASCII file containing more command-line arguments

The following sections discuss each of these elements in more detail.

2.2.2 Driver Command

- Native compiler* If you are using High C/C++ as a native compiler (that is compiling and running your program on the same type of processor), the driver command takes the form **hc**.
- Cross compiler* If you are using High C/C++ as a cross compiler, the driver command takes the form **hc***suffix*, where *suffix* represents the target machine.

For compilers crossing to the PowerPC processor, use the command **hcppc**.

Note: In the examples throughout this manual, we use the command **hc**.

2.2.3 Compiler Options

You can specify any number of compiler options on the command line. Each option applies to all the files on the command line (for which it makes sense).

You can also set options in an argument file — an ASCII file read by the driver at compile time. See §2.2.5: *Driver Argument Files* for more information.

You can also set options on the `ARGS=` line in the driver configuration (`.cnf`) file. See 2.5.5, *Using the Driver Configuration (.cnf) File* for more information.

Any option not recognized by the driver is assumed to be a linker option and is passed to the linker. To explicitly pass options to the linker, use compiler option **-Hldopt**. If an option passed to the linker has arguments, you must use option **-Hldopt** to pass it.

In order to pass options from the driver to the assembler, you must use option **-Hasopt**.

Note: If an argument to an option passed to the linker or assembler contains embedded whitespaces, you must enclose the entire argument in quotes.

2.2.4 Driver Input Files

The High C/C++ driver can process a variety input file types, including those containing assembly source code, C or C++ source code, preprocessed source code, and object code.

The driver recognizes file names with the extensions shown in Table 2.1. If you specify a file name with no extension on the command line, the driver

assumes that it is an object file. For example, if you type **hc** alpha on the command line, the driver invokes the linker *without first invoking the compiler*, because it assumes alpha is an object file.

If the name of your source file has extension **.cpp**, **.ii**, **.cc**, or **.C** (UNIX hosts only), the driver invokes the C++ compiler. (See option **-Hcppext** for exceptions.)

If the extension is **.c** or **.i**, the driver invokes the C compiler, unless you specify Incremental C++ with command-line option **-Hcpp1v1=1**. (See *Migrating from C to C++* in the **High C/C++ Language Reference** and Chapter 3: *Using Compiler Options*.)

Table 2.1 *Default File-Name Extensions*

Extension	Meaning
.asm	Assembly source module.
.c	C source file. See Note 1 (following this table).
.C .cc .cpp	C++ source file (uppercase .C is for UNIX hosts only). See Note 1 (following this table).
.i .ii	Preprocessed source files. See Note 2 (following this table).
.s	Assembly-language file; the assembler is invoked to assemble it.
.obj .lib .dll	Object module Library file Dynamic link library
anything except the preceding, but usually:	
.o .a .so	Object module Archive library Shared library

Note 1 To change the default C and C++ file-name extensions, edit the following variables defined in the configuration (**.cnf**) file:

CEXT Default file-name extension for C source files (usually **.c**)

CPPEXT Default file-name extension for C++ source files (usually .cpp, .cc, and .C)

For a discussion of the driver configuration file, see §2.5.5: *Using the Driver Configuration (.cnf) File*

You can also use command-line option **-Hcppext** to change the C++ file-name extensions; see the description of **-Hcppext** in Chapter 3: *Using Compiler Options*.

Caution: The same extensions must not appear in both **CEXT** and **CPPEXT**.

Note 2 (.i and .ii) By convention, the compiler assigns extension .i (C) or .ii (C++) to source files that have been preprocessed by a macro preprocessor but have not been compiled.

Use option **-P** to generate .i and .ii files, as explained in Chapter 3: *Using Compiler Options*. See §2.5.3: *Using the Macro Preprocessor* for more information.

2.2.5 Driver Argument Files

You can package driver arguments (compiler options and file names) in a *driver argument file*, an ASCII file that contains a list of arguments. (This is sometimes called a *response file* or *@ file*.) In the argument file, line boundaries are interpreted as whitespace.

You specify the driver argument file name, prefixed by @, on the driver command line. The arguments in the driver argument file are interpreted as if they appeared on the command line.

For example, the following command compiles `my_prog.c` with the options and arguments in `my_args.lst`:

```
hc @my_args.lst my_prog.c
```

where file `my_args.lst` contains the following:

```
-Hon=Int_function_warnings,Downshift_file_names
-Hoff=Cross_jump
-g -v -Hansi
```

The following conventions apply to argument files:

- You can list arguments on multiple lines.
- A newline is treated as whitespace.
- The argument file cannot contain user comments.
- You can specify more than one argument file on the command line.

2.2.6 Executable File Names

The linker combines object files and library functions into an executable file named `a.out`.

You can specify a different executable name with option `-o`. For more information, see Chapter 3: *Using Compiler Options*.

2.3 Using the High C/C++ Driver

This section contains examples of using the High C/C++ driver. The examples demonstrate how to perform some common compiler operations, using default driver settings. For information on techniques available for customizing the compilation process, see §2.5: *Customizing the Compilation*.

2.3.1 Getting Help

High C/C++ includes an on-line help utility that displays information about run-time library functions, compiler options, pragmas, and toggles. To access this on-line help, enter this at the command line:

```
hc -hkeyword
```

where the following apply:

- `-h` must be lowercase
- `keyword` is all or part of a control or function name, or a topic

For example, to get help on all controls for include-directory specification, enter this command:

```
hc -hinclude
```

To see a listing of *all* the High C/C++ command-line options (and what they do), enter this at the command line:

```
hc -h
```

2.3.2 Compiling and Linking in One Step

Preprocessing and compiling The driver invokes the macro preprocessor and (if needed) reads the profile. It then invokes the High C/C++ compiler, which converts C or C++ source programs into object files. (See §2.5: *Customizing the Compilation* for information about the macro preprocessor and profiles.)

Linking After invoking the compiler, the driver calls the linker if both of the following are true:

1. The compiler has not detected any compilation errors.
2. You have not suppressed linking with a compiler option (for example, option **-c**, **-S**, **-E**, or **-Hnoobject**).

The linker links the object files into an executable file.

Any arguments not recognized by the driver as compiler arguments are passed to the linker. Any file name with an unrecognized suffix is passed directly to the linker. (For a list of recognized file-name suffixes, see Table 2.1, *Default File-Name Extensions*.)

2.3.2.1 Sample Compile-and-Link Invocations

C programs To compile and link a C program in file `my_prog.c` with optimization level 5, enter this command, where **-O5** is the option that sets the optimization level:

```
hc -O5 my_prog.c
```

To compile the C program in file `my_sort.c`, link it, generate an executable file named `my_sort`, and send an assembly listing to `stdout`, enter this command:

```
hc -Hanno -S my_sort.c -o my_sort
```


C++ programs To compile and link the C++ program in file `c_plus.cpp`, enter this command:

```
hc c_plus.cpp
```

2.3.3 Compiling without Linking

You can use the driver to compile and link your program in two separate steps, as follows:

Step 1 Invoke the driver to run the compiler alone by specifying option `-c` to suppress linking:

```
hc -c myprog.cpp
```

When you suppress linking, the compiler creates one or more relocatable object files (`filename.o`) and places them in your working directory.

Step 2 Invoke the driver to run the linker by specifying the object file:

```
hc myprog.o
```

or by specifying no extension at all:

```
hc myprog
```

Passing linker options directly If linker options clash with compiler options, use compiler option `-Hldopt` to bypass the compiler and send linker options directly to the linker.

Note: You can use options `-c`, `-S`, `-E`, or `-Hnoobject` to direct the driver to compile your program but suppress the linking step. For information about these options, see Chapter 3: *Using Compiler Options*.

For platform-specific information about how to link a program's object files independently (not using the driver to invoke the linker), see the **ELF Linker/Locator and Archiver User's Guide**.

2.3.4 Generating Source and Assembly Listings

Use these command-line options to generate source and assembly listings:

`-Hlist` Write a source listing to standard output.

-Hanno and **-S** Generate an assembly-language file annotated with lines from the original source code.

For more information about these options, see §3.2: *Compiler Option Reference*. See also Appendix A: *Generating List Files*.

2.3.5 Preparing Your Program for Debugging

If you want to use a debugger on your program, specify **-g** on the driver command line. This option causes the compiler to include debugging information in the object files.

For example:

```
hc -g my_prog.c
```

See Chapter 13: *Debugging and Diagnostic Tools* for platform-specific information about debugger support. For information about stripping debugging information from object files, see the **ELF Linker/Locator and Archiver User's Guide**.

2.4 Linking Run-Time Libraries

When you invoke the driver to compile and link your program, several factors determine which MetaWare libraries the driver links into your program. For example:

- whether the target code is big-endian or little-endian
- whether you have specified option **-fsoft** to indicate that the application will use software floating-point emulation (rather than a floating-point unit on the target processor)

The C and C++ libraries are named `libc.a` and `libcc.a`, respectively. The variants of the libraries are distinguished by their location in subdirectories below `$HCDIR/lib`.

For information about the types of libraries available with your distribution, including their names and locations, consult the file `readme`, in your main High C/C++ directory.

2.4.1 Linking MetaWare Libraries and System Libraries

Some libraries provided with operating systems do not conform to the ANSI C libraries. MetaWare supplies a library and a set of header files that do conform to the ANSI C Standard. When you link a program, you can specify which of these libraries to use with the **-Hmwlib** and **-Hsyslib** compiler options.

- The **-Hsyslib** option specifies the system library and header files.
- The **-Hmwlib** option directs the compiler to use the MetaWare-supplied library and header files.

When you use **-Hmwlib**, the system library is placed immediately after the MetaWare library when the linker is invoked. In this way, any esoteric system functions are available even when you specify the **-Hmwlib** option.

2.4.2 Including Additional Libraries

By default, the driver links the appropriate and necessary C++, C, and operating-system libraries and other files to produce your executable file.

To include specific libraries and tell the linker where to find libraries, use these options:

- Ldir** Search a directory for a library specified with option **-l**.
- lstring** Include library *libstring.a*, or *libstring.so*, *string.dll*, or *string.lib* in the link.

Linker search strategies The linker uses the following search strategies when you use options **-Ldir** and **-lstring**.

For statically linked libraries:

1. The linker picks *libstring.a* as the first choice, if that file is present in the library directory.
2. If *libstring.a* is not present, the linker attempts to link *string.lib*.

For shared object libraries (or DLLs):

1. The linker picks `libstring.so` as the first choice, if that file is present in the library directory.
2. If `libstring.so` is not present, the linker attempts to link `libstring.dll`.
3. If `libstring.dll` is not present, the linker attempts to link `string.dll`.
4. If none of the above are present, the linker attempts to link `libstring.a`.
5. If `libstring.a` is not present, the linker attempts to link `string.lib`.

For additional information on the strategy the linker uses to link libraries, consult the **ELF Linker/Locator and Archiver User's Guide**.

2.4.3 Locating Libraries

To find the installed location of MetaWare run-time libraries, invoke the driver with command-line option **-Hbatch** or **-v**.

- Option **-Hbatch** creates a batch file that includes the paths to all the run-time libraries needed and specified.
- Option **-v** displays on your monitor the full pathnames of the libraries being linked.

2.5 Customizing the Compilation

You can run the driver as installed, without modification, to compile and link your applications. You can also use special features and utilities included in the installation to fine-tune your compilation process. This section covers the following topics:

§2.5.1: *Setting Compiler Controls*

§2.5.2: *Setting Compile-Time Environment Variables*

§2.5.3: *Using the Macro Preprocessor*

§2.5.4: *Using Profiles*

§2.5.5: *Using the Driver Configuration (.cnf) File*

§2.5.6: *Using Makefiles*

2.5.1 Setting Compiler Controls

Compiler controls consist of options, toggles, and pragmas. They modify the compilation process and let you control many aspects of compilation, from the format of a listing to subtle variations in the kind of code generated.

Command-line options Options are typically placed on the command line. You can specify any number of options on the command line. Each option applies to all the files on the command line (for which it is applicable).

Options can also be set on the `ARGS=` line in the driver configuration (`.cnf`) file. See 2.5.5, *Using the Driver Configuration (.cnf) File* for more information.

You can also set options in an argument file — an ASCII file read by the driver at compile time. See §2.2.5: *Driver Argument Files* for more information.

Toggles and pragmas You can set toggles and pragmas either on the command line or within a source-code module. (Target processor and floating-point toggles are an exception; they must be set on the command line.)

DOS hosts only **Note:** DOS imposes a limit of 128 characters on the command line. If you must use a command line longer than 128 characters, see §2.2.5: *Driver Argument Files*.

Compiler controls are listed alphabetically and described fully in the following chapters:

- Chapter 3: *Using Compiler Options*
- Chapter 4: *Using Compiler Toggles*
- Chapter 5: *Using Compiler Pragmas*

To get a listing of all the controls being passed to the compiler and linker, use option `-v`.

2.5.1.1 Scope of Controls

You can set some toggles and pragmas more than once in a given module. If a control can vary *within* a module, each time the control changes, the new value takes effect. The default setting stays in effect until the compiler encounters a user-specified setting.

If a control *cannot* vary within a module, and it is set more than once, the compiler uses the last value encountered before the information is required.

2.5.1.2 Precedence of Controls

In general, for those controls that can be specified more than one way, the following precedence rules apply:

1. Default values in the compiler can be overruled by values set in the driver configuration file.
2. Configuration-file values can be overruled by options.
3. Options can be overruled by pragmas and toggles in a profile. (See §2.5.4: *Using Profiles*.)
4. Pragmas and toggles in a profile can be overruled by pragmas and toggles in regular source files.
5. Pragmas and toggles in regular source files can be overruled by subsequent pragmas and toggles in source.

Pragmas and toggles set in your program's source files are the final user-specified values encountered. These controls are processed in the order in which they appear in the source code.

Some controls cannot be overridden There are exceptions to the rules of precedence. Some command-line controls cannot be overridden. Also, some options (such as **-g** for debugging) have no corresponding pragmas or toggles.

In some cases, a control requires the compiler to make a decision about the control's setting (value) before processing any source code. For instance, if a control modifies the size of an object, that control must be set before the object is declared.

For controls that fall into this category, the control setting used is always the last setting specified before the object's first declaration in the program

being compiled; the compiler ignores any subsequent settings. This characteristic is noted in the description of each such control.

2.5.2 Setting Compile-Time Environment Variables

The environment variables listed in Table 2.2 control certain aspects of compiler behavior. If you are running on DOS, you can set these variables from within your `autoexec.bat` file. On UNIX, you can set them from within your `.login` and `.cshrc` files.

Table 2.2 *Compile-Time Environment Variables*

Variable	Meaning
HCDIR	Pathname of the directory where the High C/C++ distribution was installed
IPATH	Pathname of the directory where the include files are located
TMPPREFIX	Alternate directory for temporary files required by the compiler

The following requirements apply to these environment variables:

- You must either set HCDIR or modify the compiler configuration file to explicitly define the symbol HCDIR.
- You must either set TOOLSDIR or modify the compiler configuration file to explicitly define the symbol TOOLSDIR.
- For the DOS-hosted compiler, temporary files required by the compiler are ordinarily created in your current working directory. For the UNIX-hosted compiler, the default directory for temporary files is typically `/tmp` or `/var/tmp`.

To designate an alternate directory for temporary files on DOS, set the TMP or TMPPREFIX environment variable in the compiler configuration file. To designate the directory on UNIX, set TMPDIR or TMPPREFIX in the configuration file. The compiler searches TMP or TMPDIR first, then TMPPREFIX.

Examples of setting these environment variables:

```
UNIX hosts  setenv HCDIR /u/hcpcp
            setenv TOOLSDIR /hc/mytools
            setenv TMPDIR /mytemp
            setenv TMPPREFIX /mytemp2

DOS hosts   set HCDIR=\util\hcpcp
            set TOOLSDIR=c:\hc\tools
            set TMP=d:\temp
            set TMPPREFIX=d:\temp2
```

(For additional information about variables you can set to customize the compilation process, see §B.2: *User-Modifiable Configuration Variables*.)

2.5.3 Using the Macro Preprocessor

Before a source file is compiled, it is preprocessed by the macro preprocessor. The preprocessor performs the following operations:

- includes in a compilation the contents of source files specified in **#include** statements
- performs macro expansion on tokens that appear in source files
- directs the compiler to skip over portions of source code
- inserts **#line** directives in source code

2.5.3.1 Invoking the Preprocessor

Use option **-P** (see Chapter 3: *Using Compiler Options*) to preprocess your source file, suppress compilation, and save the preprocessed source code to an intermediate **.i** (C) or **.ii** (C++) file. See also §2.2.4: *Driver Input Files*.

2.5.3.2 Choosing the Internal or External Preprocessor

UNIX hosts Two preprocessors are available on UNIX host platforms:

- an internal (*inboard*) preprocessor that is an integral part of the High C/C++ compiler and that conforms to the ANSI C standard
- an external (*outboard*) preprocessor (if it is provided with the UNIX host operating system)

ANSI-C conforming preprocessor MetaWare provides the internal preprocessor because the external preprocessor supplied by UNIX prior to the introduction of SVR4 did not conform to the ANSI C Standard.

The MetaWare preprocessor can emulate most of the behavior of the AT&T Portable C Compiler, and can therefore be used to compile what is commonly known as “K&R C” (the version of the C language documented in the first edition of **The C Programming Language** by Brian W. Kernighan and Dennis M. Ritchie, before the ANSI C Standard was defined).

Options for specifying the preprocessor The following options specify which preprocessor to use:

-Hcpp Invokes the external (outboard) macro preprocessor on the source files.

When you compile with option **-Hcpp**, the output of the preprocessor is sent to a temporary `.i` file, which then serves as input to the compiler.

-Hnocpp Invokes the MetaWare internal (inboard) preprocessor.

Note: If you use option **-Hcpp**, the driver reads the profile after preprocessing the source files.

2.5.3.3 Using Pre-Defined Macros

The High C/C++ compiler provides several predefined macro symbols that you can use with the **#if**, **#ifdef**, or **#ifndef** preprocessor directives to include code written for a specific compilation.

For example, if the `__STDC__` macro symbol is defined, as it is when you compile with option **-Hansi**, the compiler accepts code that complies with the ANSI C Standard. If the `__HIGHC__` symbol is defined, High C/C++ extensions are supported.

Table 2.3 lists the macros predefined in High C/C++.

Table 2.3 *Predefined Macro Symbols*

	Macro Symbol	Meaning
	<code>__BE_PPC</code>	Defined when you compile in big-endian mode
<i>C++ only</i>	<code>__cplusplus</code>	Defined when you compile C++ modules; check one of these flags to ensure that compilation proceeds in C++ mode
<i>C++ only</i>	<code>__EXCEPTIONS__</code>	Defined if your program is aware of C++ exception handling (that is, if you compiled with command-line option -Hexcept)
	<code>__HIGHC__</code>	Defined if option -Hansi is <i>not</i> specified; that is, if the system executing your application accepts non-ANSI-Standard High C/C++ extensions
	<code>__LE_PPC</code>	Defined when you compile in little-endian mode
<i>OS/2 only</i>	<code>__OS2</code>	Defined when the compiler is generating code for OS/2
	<code>__SOL_PPC</code>	Defined when the compiler is generating code for the PowerPC based on the SVR4 ABI PowerPC Supplement .
	<code>__STDC__</code>	Defined if option -Hpcc is <i>not</i> specified; that is, defined if the system executing your application is not running in PCC mode.

For more information about **-Hpcc** and PCC mode, see Chapter 3: *Using Compiler Options* and Chapter 4: *Using Compiler Toggles*.

2.5.4 Using Profiles

A *profile* is an ASCII file that is treated as a prefix to the source file. Its purpose is to customize the compiler to particular needs. The compiler reads the profile immediately before reading the source files used in compilation.

The default profile name is `hc.pro`, but you can specify an alternate name with option **-Hpro**, as explained in Chapter 3: *Using Compiler Options*. A default `hc.pro` file resides in the same directory as the compiler executables.

Note: A local `hc.pro` overrides the default profile.

You can use a profile to initialize toggles, specify aliasing conventions, define macros, invoke pragmas, define global types, and so on. These new “defaults” can then be applied to all source modules in a global fashion. In other words, you can use the profile to alter compiler defaults.

For example, a profile containing these lines:

```
#pragma On(List)
typedef long int integer;
```

causes the compiler to do the following:

- turn on toggle **List** (with pragma **On**) to produce a source listing
- create a **typedef** named **integer**, synonymous with **long int**

Searching for the profile

When you compile a program, the compiler searches for a profile following the same paths it uses to search for other include files. If the compiler finds a profile, it reads the profile before it reads your application’s source files (unless you use command-line option **-Hnopro**).

Caution: If the compiler does not find a profile, it assumes there is none and immediately starts reading your application’s source file — even if you use the **-Hpro** option on the **hc** command line. This happens because the **-Hpro** option specifies the name of a profile but does not require that the file be found.

Note: If you use the **-Hcpp** option to invoke the external macro preprocessor, the compiler does not read any available profile until preprocessing is complete.

If you use option **-Hnocpp** to invoke the internal macro preprocessor, the compiler reads the profile before preprocessing takes place.

Using conditional compilation directives

You can use **#ifdef/#else** compiler directives in your profile to generate different code, depending on whether a symbolic constant is defined on the command line. For example, if profile **my.pro** contains conditional code depending on **#ifdef DEF**, you can include the code in your program by entering the following arguments on the command line:

```
hc myprog.c -Hpro=my.pro -DDEF
```

This works even though option **-DDEF** follows the **-Hpro** option.

You can also use the `-D` option with command-line arguments in a driver input file:

```
hc myprog.c @hc.arg -DDEF
```

See §2.2.5: *Driver Argument Files* for how to set up a driver input file.

2.5.5 Using the Driver Configuration (`.cnf`) File

The High C/C++ driver configuration (`.cnf`) file is an ASCII file the driver reads to configure the compilation process. This configuration file is written in a “driver language” that is processed by the driver when you invoke the compiler.

The `.cnf` file names are defined as follows:

- `hcppc.cnf` if you are using a PowerPC cross compiler
- `hc.cnf` if you are using a PowerPC native compiler

For most installations, there is no need to change the contents of the configuration file. However, it can be customized as required.

See Appendix B: *Configuring the Driver* for more information about the driver and its associated configuration file.

2.5.6 Using Makefiles

To build a High C/C++ application, it is often more convenient to write and execute a *makefile* than to place all the information needed for the compilation on the driver command line. A makefile is an ASCII file that specifies which target files (object files) depend on your program's source and header files.

To write a makefile, you must know how your source files depend upon each other during the compiling and linking process. The following High C/C++ options can help you determine the file dependencies needed to write makefiles:

-Hmake	Directs the compiler to write the dependencies needed to create a makefile to standard output. The list contains the names of files included in the source file, as well as the name of the source file itself.
-Hmakeof=<i>fname</i>	Directs the compiler to write the dependencies needed to create a makefile to the named output file (<i>fname</i>). The list of dependencies is the same as for -Hmake .
-Hmake	Also generates makefile dependencies, but writes them to a file with the file-name extension <code>.u</code> instead of to standard output.
-Hrel	Instructs the compiler not to place absolute include files (those included with <code>< ></code>) into the list of dependencies produced by the -Hmake , -Hmakeof , or -Hmake option.

For more information about these options, see Chapter 3: *Using Compiler Options*.

Note: The High C/C++ distribution does not include a make utility. You must use a third-party make utility.
