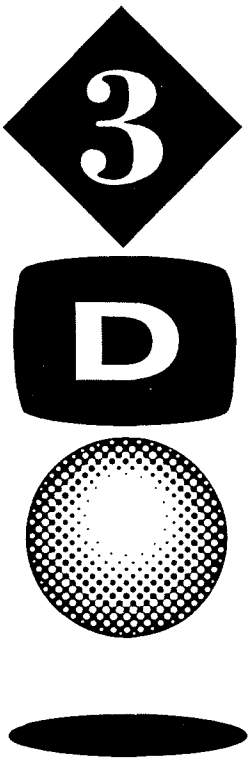


3DO M2 RELEASE ◆ VERSION 2.0

Volume 5

- *DIAB Data D-C++ Language User's Manual*
- *DIAB Data PowerPC Target User's Manual*
- *DIAB Data D-AS/PowerPC Assembler User's Manual*
- *DIAB Data Utilities User's Manual*

02572-003

D-C++

Language User's Manual

DIAB DATA

Defining Compiler Performance

Copyright Notice

Disclaimer

Diab Data makes no representations or warranties with respect to the contents of this publication, and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Diab Data reserves the right to revise this publication and make changes from time to time in the content hereof without obligation on the part of Diab Data to notify any person or company of such revision or changes.

In no event shall Diab Data, or others from whom Diab Data has a licensing right, be liable for any indirect, special, incidental, or consequential damages arising out of or connected with a customers possession or use of this product, even if Diab Data or such others has advance notice of the possibility of such damages.

Trademarks

Diab Data, alone and in combination with D-AS, D-C++, D-CC, D-F77, and D-LD are trademarks of Diab Data, Inc. All other trademarks used in this document are the property of their respective owners.

Diab Data, Inc.
323 Vintage Park Drive
Foster City, CA 94404
USA

Tel **415-571-1700**
Fax **415-571-9068**

Email **support@ddi.com**

CONTENTS

List of Tables

List of Figures

1 Introduction

This manual describes the **Diab Data D-CC Optimizing C++ compiler**.

It is written for the professional programmer and contains detailed information about how to use the compiler, including a description of all command line options, and hints for porting existing code to the compiler.

For information describing back-end features and optimizations for the target processor, see the separate *Compiler Target User's Manual.*

Important features and extensions

- Many compiler controls and options to for great flexibility over compiler operation and code generation. See Chapter 3, "Invoking the Compiler," beginning on page 11.
- A flexible configuration language to let the user change the default target processor and the default command line options. See Chapter 4, "Configuration File," beginning on page 35.
- Many features and extensions targeted for the embedded systems programmer. See Chapter 8, "Use in an Embedded Environment," in the *Compiler Target User's Manual.*
- Optimizations and features tailored individually for each processor type within a family. See "Using D-C++ for different targets" on page 9 for information on how to specify the target processor.
- Extensive compile time checking to detect suspicious and non-portable constructs. See Chapter 7, "The Lint Facility," beginning on page 63.
- Powerful profiling capabilities to locate bottle necks in the code. The profiling information can also automatically be used as feedback to the compiler, enabling even more aggressive optimizations. See Chapter 6, "Optimization Hints," beginning on page 59 and also the discussion of D-BCNT in the *Utilities User's Manual.*
- C++ Templates, Exceptions, and Run-time Type information.

High performance optimizations

A wide range of optimizations, some of which are unique to Diab Data compilers, produces very fast and compact code as measured by independent benchmarks. Special optimizations include superior inter-procedural register allocations, inlining, and reaching analysis.

Optimizations fall into three categories: local, function-level, and program-level, as listed next. See Chapter 7, "Optimizations," in the *Compiler Target User's Manual* for details.

- Local optimizations within a block of code:

  Constant folding
  Delete TST
  Local common sub-expression elimination

Local strength reduction
Minor transformations
Peep-hole optimizations
Switch optimizations

- Function global optimizations within each function:

  Auto increment/decrement optimizations
  Automatic register allocation
  Branch to small code
  Complex branch optimization
  Condition code optimization
  Constant propagation
  Dead code elimination
  Delayed branches optimization
  Delayed register saving
  Entry/exit code removal
  Extend optimization
  Global common sub-expression elimination
  Lifetime analysis (coloring)
  Link register optimization
  Loop invariant code motion
  Loop statics optimization
  Loop strength reduction
  Loop unrolling
  Memory read/write optimizations
  Reordering code scheduling
  Restart optimization
  Branch-chain optimization
  Space optimization
  Split optimization
  Structure member to registers
  Tail recursion
  Tail jump optimization
  Undefined variable propagation
  Unused assignment deletion
  Variable location optimization
  Variable propagation

- Program global optimizations across multiple functions

  Argument address optimization
  Function inlining
  Glue function optimization
  Inter-procedural optimizations
  Literal synthesis optimization
  Profiling feedback optimization

Professional programming tools

D-C++ is an integral part of Diab Data's suite of software tools for the professional developer. These include the D-AS Assembler, the D-LD Linker, and the D-AR

Archiver, as well as front-ends for additional languages and back-ends for different target processors.

D-C++ is a high performance programming tool, designed specially for professional programmers. Besides the benefits of state-of-the-art optimization, D-C++ reduces time spent creating reliable code because the compiler and other tools are themselves fast, and because the compiler includes many built-in, customizable, checking features which help you find problems earlier.

The compiler is often particularly helpful for speeding up and/or reducing the size of existing programs developed with other tools.

Ease-of-use

D-C++ is both easy to use and flexible. With a few short commands, D-C++ is loaded onto your system with its corresponding files and is ready for use. With over 100 command-line options available, you have a rich command language to work with, and can easily and quickly customize D-C++ to be compatible with existing systems or compilers and to meet individual needs.

Portability

D-C++ conforms fully to the ANSI X3.159-1989 standard (called "ANSI C"), with extensions for compatibility with other compilers to ease conversion from existing code.

Standard C programs can be compiled with a strict ANSI option that turns off the extensions and reduces the language to the standard core. Alternatively, such programs can be gradually upgraded by using the extensions as desired.

Diab Data tracks the evolving ANSI C++ standard. Both exceptions and templates are implemented.

This manual

This guide contains all information necessary to use the compiler effectively, including chapters on installing and invoking the compiler, running it with non-default options, additions to ANSI C and C++ especially for embedded systems applications, hints on optimizing code and using the compiler with code developed with other tools, and appendices describing limits, implementation-defined behavior, and errors. Please see the table of contents for a detailed overview.

This manual does not explain the C++ or C++ language, nor does it attempt to teach C++ programming. See "Additional documentation" next for references to standard works.

Additional documentation

- *Compiler Target User's Manual*
- *Assembler User's Manual*
- *Linker User's Manual*
- *C Library Manual*

- *C++ Class Reference Manual*
- *Utilities User's Manual*
- `relnote.txt` in the *version_path* directory (see Table 2-1, "Example Default Installation Path Names," on page 5).

The C Programming Language, Second Edition by Brian Kernighan and Dennis Ritchie and the ANSI C standard *X3.159-1989* is recommended as a C reference.

The C++ Programming Language, Second Edition by Bjarne Stroustrup or *The Annotated C++ Reference Manual* are recommended as C++ references. There is also a draft proposal for an ANSI C++ standard which is updated as the standardization work progresses.

Document conventions

This manual uses the following typographic conventions:

Table 1-1 Document Conventions

Example	Description
`dcc -o test.c`	This font is used for file and program names, examples, user input, and program output.
if, **main( )**, **#pragma**, **__pack__**	Bold type is used for keywords, operators and other tokens of the language, and library routines. Some names begin or end with underscores. These underscores and special characters such as # shown in bold are required.
variable, *filename*	Italic type is used for placeholders for information which you must supply. Italics are also used for emphasis, to introduce new terms, and for titles.
[optional text]	An item enclosed in brackets is optional.
{ item1 | item2 }	Two or more items enclosed in braces and separated by vertical bars means that you *must* choose exactly one of the items.
item ... item ,...	An item followed by "..." means that items of that form may be repeated separated by whitespace (spaces or tabs). A character preceding the "..." means that the items are separated by the character, shown here as a comma, and optional whitespace. The item may be a single token, an optional item enclosed in [ ] brackets (meaning that the item may appear not at all, once, or multiple times), or a set of choices enclosed in { } braces (meaning that a choice must be made from the enclosed items one or more times).

2 Installing the Compiler

Installation and compiler components

Brief installation procedures are shipped with the media.

All files are found in subdirectories of a single root directory. The following terminology is used throughout this manual to refer to that root and related subdirectories:

- ***install_path*** represents the full path name of the root directory. The root directory contains *version* subdirectories, each acting as a sub-root for all files related to a single version of the compiler. This allows multiple versions of the tools to reside on the same file system.
- ***version_path*** is the name given to the complete path name for a single version of the compiler.
- ***host_dir*** is the name of a subdirectory under *version_path* containing directories specific to a single type of host, e.g., `MSDOS` or `SUNS` (for Sun Solaris). This permits tools for different types of systems to reside on a single networked file system.

These names for a default installation depend on the host file system as shown in the following table. Assume that the version number is 3.7a.

Table 2-1 Example Default Installation Path Names

System	Default Root *install_path*	Default *version_path*
UNIX	`/usr/lib/diab`	`/usr/lib/diab/3.7a`
DOS	`C:\DIAB`	`C:\DIAB\3.7a`
MPW	`{MPW}diab`	`{MPW}diab:3.7a`

➤ Instructions and examples for DOS also apply to Window95, Windows NT, and OS/2.

The table on the next two pages lists the subdirectories of *version_path* and important files contained in them.

Table 2-2 *version_path* **Subdirectories and Important Files**

Subdirectory or File	Contents or Use
host_dir/bin *host_dir*\bin (DOS)	**Programs intended for direct use by the user.**
dplus	D-C++ main driver for C++ source files. Uses dtools.conf (see lib below).
dcc	D-CC compiler driver for C source files. Uses dtools.conf (see lib below).
das	D-AS assembler. Uses dtools.conf (see lib below).
dld	D-LD linker. Generates executable files from one or more object files and object libraries (archives).
dar	D-AR archiver. Creates an object library (archive) from one or more object files.
dbcnt	D-BCNT basic block counter. Generates profiling information from files compiled with -Xblock-count.
ddump	D-DUMP object file dumper. Examines or converts object files, e.g. COFF to IEEE 695 or Motorola S-Records.
reorder	This program is started from dplus. It reschedules the instruction sequence to avoid stalls in the processor pipeline and does some peep-hole optimizations. See Chapter 7, "Optimizations," in the *Compiler Target User's Manual* and option -Xkill-reorder on page 26.
host_dir\lib *host_dir*/lib (DOS)	Programs and files used by programs in bin.
dtoa	Generic C++ compiler. Reads a target.cd file to direct code generation (see below).
ctoa	Generic C compiler. Reads a target.cd file to direct code generation (see below).
conf dtools.conf default.conf user.conf	Configuration files read by the compiler at startup, primarily to supply command line options. See Chapter 4, "Configuration File," beginning on page 35 for details. (**Note:** on DOS, extension .conf is .CON)
include	Standard include files for use in user programs.
src	**Source code for replacement routines for system calls.** These functions must be modified before they can be used in an embedded environment. See Chapter 8, "Use in an Embedded Environment," in the *Compiler Target User's Manual.*

Table 2-2 *version_path* **Subdirectories and Important Files** *(continued)*

Subdirectory or File	Contents or Use
Targets	Files, programs, and subdirectories specific to each target. See the applicable *Compiler Target User's Manual* for more information on a specific target.
Example target subdirectories:	
`MC00ES`	MC68000, Embedded mnemonics, Software floating point
`MC100B`	MC88100, BCS/OCS
`PPC`	Generic PowerPC
`PPCEH`	PowerPC, EABI, Hardware floating point
Files in target subdirectories:	
`target.cd` `target.ad`	Compiled description file used by the compiler and assembler to direct code generation (no user-modifiable code).
`crt0.o`	Startup code called before which in turn calls `main`.
`mcrt0.o`	Version of `crt0` used during profiling.
`libc.a`	General library containing all ANSI standard C functions as documented in the *Library Reference Manual* (except those in `libm.a`) plus additional functions required to support code generated by the compiler for some constructs. Required for all languages.
`libcomplex.a`	C++ complex math class library.
`libd.a`	C++ iostream class library.
`libm.a`	Math library as documented in the *Library Reference Manual.*

Accessing current and other versions of D-C++

The D-C++ tools do not require that any environment variables be set (although may be, see the next section). Once started, each version of the `dplus` main driver "knows" where to find the programs it calls (the main driver program is modified with the selected directory during installation). Thus, running the compiler requires only that the `dplus` main driver program be accessed in any of the usual ways:

- Add *version_path*`/`*host_dir*`/bin` to your path (*version_path*`\`*host_dir*`\bin` for DOS).
- Create an alias or batch file that includes the complete path directly.
- Copy `dplus` to an existing directory in your path, e.g., `/usr/bin` on UNIX.

You can invoke any specific version by invoking its desired main driver. Here are three ways:

- Rename the main driver for the different version. For example, to execute an older version 3.6f, rename `dplus` in the bin directory for the older version to `dplus36f`. Then access `dplus36f` in any of the usual ways as described above.
- Modify your path to put the directory containing the desired version before the directory containing any other version. The `dplus` command will then access the desired version.
- Create an alias or batch file that includes the complete path of the desired version.

Environment variables

Several environment variables can be set to control operation of Diab Data tools. The method used to set these variables depends on the operating system as show in the following table.

Table 2-3 Setting Environment Variables

System	Command[a]
UNIX	`export` *variable=value*
DOS	`set` *variable=value*
MPW	`Set` *variable value* `Export` *variable*

a. Commands are generally case-sensitive in UNIX and MPW but not in DOS.

The next table describes all environment variables recognized by the compiler[1].

Table 2-4 Environment Variables Recognized by the Compiler

Variable	Description
`DCONFIG`	Specifies the configuration file used to define the default behavior of D-C++. Chapter 4, "Configuration File," documents the configuration file. If neither `DCONFIG` nor the -WC option is used (see "D-C++ startup" on page 35), D-C++ will use: UNIX: *version_path*`/conf/dtools.conf` DOS: `%`*version_path*`%\CONF\DTOOLS.CON` MPW: `"{`*version_path*`}:conf:dtools.conf"`

`DTARGET`
`DOBJECT`
`DFP`
	These three environment variables specify the target processor, object and mnemonic type, and floating point method respectively. These variables are normally set by using the `dctrl` program. See "Using D-C++ for different targets" next.

1. The environment variables `DIABLIB` and `DVERSION` are still recognized but are obsolete. They may be removed in future versions and should no longer be used. The variable `DENVIRON` is also recognized but is normally set automatically to `cross`. See the next section.

Table 2-4 Environment Variables Recognized by the Compiler *(continued)*

Variable	Description
DFLAGS	Specifies extra options for D-C++ and is a convenient way to specify -XO, -O or other flags without changing several makefiles. The options in DFLAGS are evaluated before the options given on the command line. See "Standard configuration files" on page 37, especially Figure 4-2 Standard dtools Configuration File - Simplified Structure on page 38 for details.

These variables may be set for one invocation of the compiler by using the -WD option or -WC for DCONFIG (see page 18), and the -t option for DTARGET, DOBJECT, and DFP (see the next section).

Using D-C++ for different targets

A complete *target configuration* specifies the target processor, the type of floating point support (hardware or software), and the object module format (e.g., ELF or COFF).

The default target configuration is set and may be changed any time by using the `dctrl` program with the -t option:

```
dctrl -t
```

This interactive program displays the full list of target configurations supported by your installation, each designated by a short code, and allows you to select one. This program is invoked during installation to select the initial default target configuration.

The target configuration is recorded in the configuration file `default.conf` (`default.con` for DOS, see "Standard configuration files" on page 37). Running `dctrl` either creates or modifies `default.conf` as required.

The target configuration is recorded in `default.conf` using three variables: DTARGET for the processor, DOBJECT for the object format, and DFP for the floating point support (a fourth variable, DENVIRON is always set to "`cross`" in the cross development environment).

The valid settings for these three variables are given in "Selecting a target" in Chapter 2, "Target Dependent Command Line Options," of the *Compiler Target User's Manual.*

When the installed version of D-C++ is able to generate code for more than one target processor, by default the `dplus` command will produce code for the default processor. Use the -VV option to display the default target:

```
dplus -VV
```

There are five ways to change the target processor. The first two are preferred.

1. Invoke dctrl with the -t option to change the default target "permanently".
2. Use the -t option on the `dplus` command line to change the target for a single invocation of `dplus`. The -t otion takes one of the codes displayed by dctrl. See the -t option on page 16.

1. Edit the `default.conf` configuration file to change the default settings for any of `DTARGET`, `DOBJECT`, or `DFP` by hand.
2. Set `DTARGET`, `DFP`, or `DOBJECT` environment variables.
3. Use the -WD*environment_variable* command line option (see page 18).

   Example:

   ```
   dplus -WDDTARGET=newtarget -c file.cpp
   ```

➤ For additional explanation, and order of precedence when more than on of these methods is used, See Chapter 4, "Configuration File," beginning on page 35.

3 Invoking the Compiler

This section describes how to activate D-C++ and discusses all command line options.

The `dplus` command

The command to execute D-C++ is as follows:

```
dplus [options] [input-files]
```

where:

`dplus`	Invokes the main driver program for the compiler suite. See "Accessing current and other versions of D-C++" on page 7 for details.
options	Command line options which change the behavior of the tools. See the following subsection for details. Options and file names may occur in any order.
input-files	A list of file names separated by whitespace. The suffix of each file name indicates to D-C++ which actions to take as described next.

The form -@*name* can also be used for either *options* or *input-files*. If found, the name must be either that of an environment variable or file (a path is allowed), the contents of which replace -@*name*. See "How command lines, environment variables, and configuration files relate" on page 35 for details.

Example: process a single file, stopping after compilation, with standard optimization:

```
dplus -O -c file.cpp
```

The following file extensions are recognized by default.

Table 3-1 Input File Types

Extension Suffix[a]		File Type
`.c`	`.C` (DOS only)	C source file
`.cpp` `.cc`	`.cxx` `.C` (non-DOS systems)	C++ source file
`.i`		Preprocessed C or C++ source file.
`.S` (non-DOS systems)		Assembly source file that is to be preprocessed first.
`.s`		Assembly source file
`.o`		Object code file

a. On DOS systems, the corresponding upper case extensions are also allowed.

➤ These default extensions can be changed with the -Wc.*ext* option. For example the option `-W:s:.asm` specifies that the file extension `.asm` is recognized as an assembly source file. See option -Wc.*ext* on page 19 for details.

The driver program runs up to four subprograms for each file given on the command line in *input-files*. The file extension suffix for each file determines the starting subprogram applied to the file; for example, by default, a file with extension `.s` is assembled and linked but not preprocessed or compiled. Also, certain options stop processing early if present. The subprograms and stopping options are as follows.

cpp	The preprocessor. Takes a C++ program as its input and processes all # directives. This program is included in main compiler program. The -P option halts D-C++ after this phase, producing a file with the `.i` suffix. (The file is not produced if the -P option is not present.)
dtoa	The C++ to assembly compiler. Consists of several internal stages such as the parser, optimizer, and code generator and generates assembly source from the preprocessed C++ source. The -S option halts D-C++ after this phase, producing a file with the `.s` suffix.
das	The assembler. Generates linkable object code from the assembly source. The -c option halts D-C++ after this phase, producing a file with the `.o` suffix.
dld	The linker. Generates an executable file from one or several object files and object libraries. The default output name is `a.out` if the -o *outname* option is not given.

The following figure shows the subprogram flow graphically.

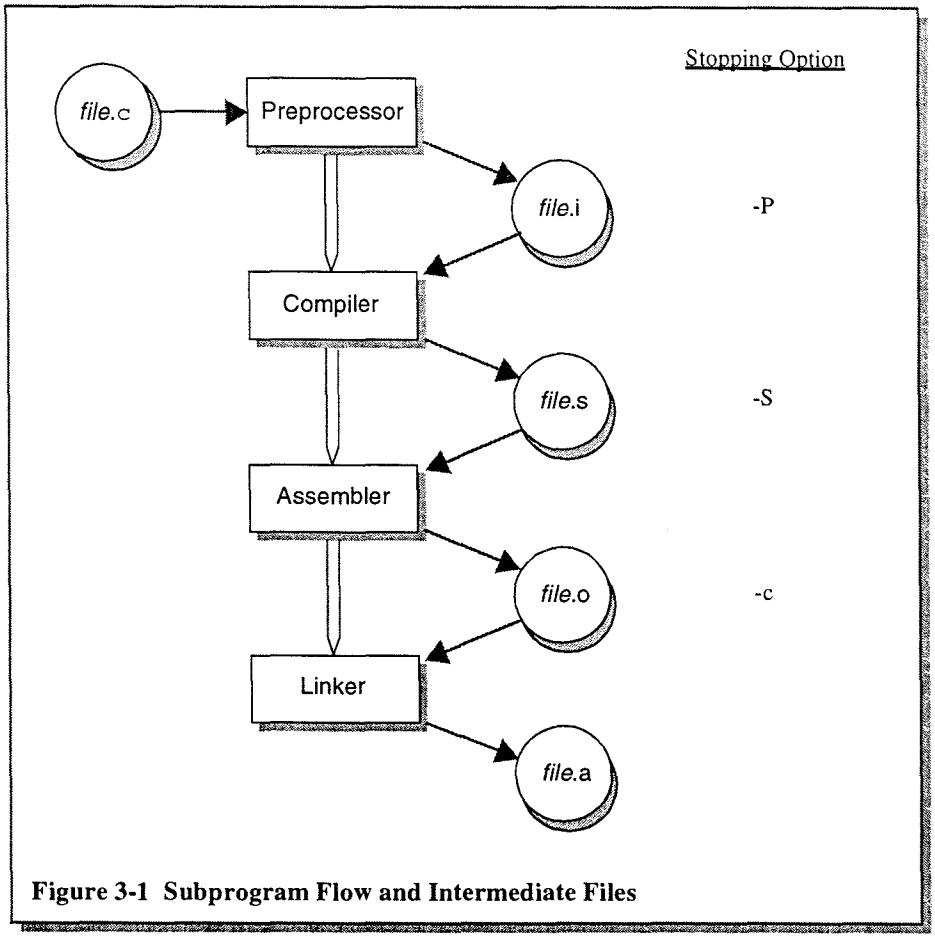


Figure 3-1 Subprogram Flow and Intermediate Files

Command line options

The next table shows all general command line options. Additional target-dependent options are in the *Compiler Target User's Manual* and may also be in the `relnote.txt`.

➤ Command line options are case-sensitive, for example, -c and -C are two unrelated options. For easier reading, command line options may be shown with embedded spaces in the table. In writing options on the command line, space is allowed only follow the option letter, not elsewhere. For example, "`-D DEBUG=2`" is valid; "`-D DEBUG = 2`" is not.

Arguments not listed in the table are passed to the linker (only).

Table 3-2 Command Line Options

Option	Action
-A-	Cause the preprocessor to ignore all predefined macros and assertions.
-A *pred(ident)*	Cause the assertion *pred(ident)* to be defined. See "assert and unassert" on page 52.
-C	Cause the C++ preprocessor to pass along all comments. Only useful in conjunction with -E or -P. (Note: the C preprocessor may be used with any language supported by Diab Data.)
-c	Stop after the assembly step and produce an object file with default file extension `.o` (see the -o option below).
-D *name* [=*definition*]	Define the preprocessor macro *name* as if by the **#define** directive. If no *definition* is given, the value 1 is used.
-E	Run only the C++ preprocessor on the named files and send the output to the standard output. All preprocessor directives are removed except for line number directives used by the compiler to generate line number information. The source files do not require any particular suffix
-g	Generate symbolic debugger information. If the -g[*n*] option is set more than once, use the last option set. -g is the same as -g2.
-g0	Turn off generation of symbolic debugger information. This is the default.
-g1	Generate symbolic debugger information, but leave out line number information. Does not affect optimization; that is, optimized as indicated by other switches and defaults.
-g2	Generate symbolic debugger information. Do most optimizations except the following, since most object formats have no way to describe them: Function Inlining Structure Member Optimization Complex Branch Optimization Loop Count-down Optimization Static Function Optimization
-g3	Generate symbolic debugger information and do all optimizations except some rescheduling. Fully optimized code can be difficult to debug. For example, there is no way to break on in-lined functions (except at the assembly level). Hence, when debugging is required, -g2 is usually a better choice.
-H	Print the path names of all include files to the standard error output.

Table 3-2 Command Line Options *(continued)*

Option	Action
-i *file1=file2*	Substitute *file2* for *file1* in an **#include** directive.
-i *file1=*	Ignore any **#include** directive for *file1*.
-i *=file2*	Include *file2* before processing any other source file.
-I *path*	Add *path* to the list of directories to be searched for include files. More than one -I option can be given on the command line. For an **#include** "*filename*" directive, search for the file in the following locations in order: 1. The directory containing the current source file. If an **#include** directive includes a path, that path becomes the current directory for the duration of that **#include**. 2. The directories given by the -I *path* option, in the order encountered. 3. The directory *version_path*`/include` (*version_path*`\include` for DOS) or as given by the -YI option. For an **#include** <*filename*> directive, search only locations 2. and 3.
-L *dir*	Add *dir* to the list of directories searched by the `dld` linker for libraries. More than one -L option can be given on the command line. The search is performed in the following order: 1. The directories given by -L *dir* in the order encountered. 2. The directory *version_path*`/`*target* (*version_path*`\`*target* for DOS) or as given by -YL, -YU, or -YP options. For a discussion of *target* subdirectories, see Table 2-2 on page 7.
-l *name*	Cause the `dld` linker to search for library `lib`*name*`.a`. See -L *dir* for search order.
-M *target-spec*	Specify the pathname of the *target-spec* file to the compiler (see `target.cd` in Table 2-2, "version_path Subdirectories and Important Files," on page 6). This file contains the target description and is read by the compiler at start-up. If the -M option is set more than once, D-C++ uses the final setting. (This options is primarily for use by Diab Data.)
-O	Optimize code. Either this or -XO must be specified to enable optimization and to invoke the reorder program. See the -XO option on page 28 for the difference between these options.

Table 3-2 Command Line Options *(continued)*

Option	Action
-o *filename*	Use *filename* when naming the file output by a subprogram instead of the default name. This option works with the -P, -S and -c options. When compiling *filename*`.cpp` the following file names are used by default if the -o option is not given: -P: *filename*`.i` -S: *filename*`.s` -c: *filename*`.o` none of -P, -S, or -c `a.out`
-P	Stop after the preprocessor step and produce a source file with default file extension `.i` (see the -o option above). Unlike with the -E option, the output will not contain any preprocessing directives. The source files do not require any particular suffix.
-S	Stop after the compilation step and produce an assembly source code file with default file extension `.s` (see the -o option above).
-t*tof*	Select the target processor with *t* (a several letter code), the object format with *o* (a one letter code), and the floating point method with *f* ("H" for hardware, "S" for software). To determine the proper *tof*, execute `dctrl -t` to display all valid combinations.
-U *name*	Undefine the preprocessor macro *name* as if by the **#undef** directive.
-V	Display the current version number of D-C++.
-VV	Display the current version number of D-C++ and the version number of all subprograms.
-v	Run the main drive program in verbose mode, printing a message as each subprogram is started.

Table 3-2 Command Line Options *(continued)*

Option	Action
-W *c,arg1*[,*arg2*...]	Pass the arguments to the subprogram designated by *c*, which is one of the following:
`:cpp:` or `p`	• The C++ preprocessor. The preprocessor is incorporated in the compiler, so this becomes a synonym for `0`.
`0`	• The or C++ compiler.
`:c:`	• The C compiler.
`:c++:`	• The C++ compiler.
`:as:` or `a`	• The assembler.
`:ld:` or `l`	• The linker.
`L`	• The object converter. Usually not implemented. If given, it will execute after the linker.
`1`	• The reorder program.
`2 - 6`	• Other filter programs. Usually not implemented. -W1 and -W2 are only executed if -O or -XO is given. They process the output from the compiler. -W3 and -W4 are always executed if given and process the output from the compiler. -W5 and -W6 process the input to the assembler.
Example:	`-W:as:,-l` or `-Wa,-l` Pass the option "-l" (lower case letter "l") to the assembler to get an assembler listing file.

Table 3-2 Command Line Options *(continued)*

Option	Action
-W *c name*	Use the program or file *name* instead of the program or file indicated by *c*. Some cases take the form -W *c name=value* *c* is one of the following:
`:cpp:` or `p`	• The C++ preprocessor. The preprocessor is incorporated in the compiler, so this becomes a synonym for `0`.
`0`	• The or C++ compiler.
`:c:`	• The C compiler.
`:c++:`	• The C++ compiler.
`:as:` or `a`	• The assembler.
`:ld:` or `l`	• The linker.
`L`	• The object converter. Will execute after the linker.
`s`	• The start-up module (`crt0.o`). Additional object files, to be loaded along with the startup file and before any other files, can be given separated by commas.
`c`	• The C library. The default is -lc. More than one can be specified, separated by commas.
`d`	• The C++ library. The default is -ld. More than one can be specified, separated by commas.
`m`	• The linker command language file (LECL). The default is the built-in LECL file.
`C`	• The configuration file to be used.
`D`	• Set a variable equal to a value for use during configuration file processing as follows: -WD*variable=value* More than one -WD option can be used to set several variables. The effect is as if an assignment statement for each such -WD variable had been added to the beginning of the main configuration file.
`1`	• The reorder program.
`2 - 6`	• Other filter programs. -W1 and -W2 execut if -O or -XO is given and process the output from the compiler. -W3 and -W4 also process the output from the compiler. -W5 and -W6 process the input to the assembler.
Example	`-W:ld:/usr/lib/dcc/3.6e/bin/dld` Use an old version of the linker.

Table 3-2 Command Line Options *(continued)*

Option	Action
-W *c .ext*	Associate a source file extension with a tool; that is, indicate to the main driver program `dplus` which tool should be invoked for an input file with a particular extension. *.ext* specifies the extension, and *c* specifies a tools as follows:
`0` or `:c:`	• The C compiler.
`:c++:`	• The C++ compiler.
`:as:` or `a`	• The assembler.
`:pas:` or `A`	• Preprocessor and assembler: both the preprocessor and assembler will be applied to the source. Allows use of C++ directives with assembly language.
	Example: `-W:as:.asm` specifies that `file.asm` is an assembly source file.
-w	Suppress all warnings.
-X *option*[=*m*] -X*n*[=*m*]	Set options providing detailed control of the compiler. There are two ways to set -X options, either with a name, -X *option*, or by a number, -X*n*. Options can be set to a decimal, octal (leading `0`) or hexadecimal (`0x`) value *m*. If an -X option is set more than once, D-C++ uses the final setting. To turn off an option, set it to zero: -X *name*=0 or -X*n*=0. To find all -X options, see: • Table 3-3, "Standard -X Options," on page 21. • Chapter 2, "Target Dependent Command Line Options," in the *Compiler Target User's Manual* • the `relnote.txt`.
-Y *c*, *dir*	Specifies a new directory *dir* for *c*. *c* can be one of the following
`I`	• Use *dir* as the default include directory.
`L`	• Use *dir* as the first default directory used by the linker for -l libraries.
`U`	• Use *dir* as the second default directory used by the linker for -l libraries.
`P`	• Use the colon-separated list *dir* as the default directories used by the linker for -l libraries.
-#	Prints the subprograms with arguments as they are executed.
-##	Prints the subprograms with arguments without actually executing them.

Table 3-2 Command Line Options *(continued)*

Option	Action
-###	Prints the subprograms with arguments inside quotes without executing them.
-@*name*	Reads command line options from either a file or an environment variable. When -@*name* is encountered on the command line, D-C++ first looks for an environment variable with the given *name* and substitutes its value. If an environment variable is not found then D-CC tries to open a file with given *name* and substitutes the contents of the file. If neither an environment variable or a file can be found, an error message is issued and D-C++ terminates.
-@@*name*	Same as -@*name* but also prints all command line options on standard output.
-@E=*filename*	Redirects any output to standard error to file *filename*.
-@O=*filename*	Redirects any output to standard output to file *filename*.

Compiler -X options

Over 100 compiler -X options provide fine control over many aspects of the compilation process when behavior other than the default is needed.

There are two ways to set these options, either by name, -X*name*, or by number, -X*n*. Options can be set to a value, *m*, given in decimal, octal (leading `0`) or hexadecimal (`0x`) by using an equal sign, -X*name*=*m* `or` -X*n*=*m*, or can be set in some cases to an unquoted string, e.g., -Xfeedback=*filename*.

To turn off an option, set it to zero: -X*name*=0 or -X*n*=0.

➤ If an option is not provided, it defaults to a value of `0` unless specified otherwise in the description below or in the `relnote.txt`.

If an option which takes a value is provided without one, then the value 1 is used unless otherwise stated.

Therefore, the following three forms are all equivalent:

-Xtest-at-top -X6 -X6=1

However, if neither option -Xtest-at-top nor -X6 had been given, the value of option -X6 would default to 0, which is equivalent to -Xtest-at-bottom.

The following table shows all standard -X options in both forms (name and number). For the most part, these options are independent of a particular target. For additional -X

options, see Chapter 2, "Target Dependent Command Line Options," in the *Compiler Target User's Manual*, and also the `relnote.txt`.

Table 3-3 Standard -X Options

X Option	Description
-Xaddr-data=*n* -X100=*n*	Specify how non-constant static and global data should be addressed. See **#pragma section** page 49 for more information.
-Xaddr-sdata=*n* -X101=*n*	Specify how non-constant static and global data that are less than or equal to -Xsmall-data should be addressed. See **#pragma section** on page page 49 for more information.
-Xaddr-const=*n* -X102=*n*	Specify how constant static and global data should be addressed. See **#pragma section** page 49 for more information.
-Xaddr-sconst=*n* -X103=*n*	Specify how constant static and global data that are less than or equal to -Xsmall-const should be addressed. See **#pragma section** for more information.
-Xaddr-string=*n* -X104=*n*	Specify how strings should be addressed. See **#pragma section** for more information.
-Xaddr-code=*n* -X105=*n*	Specify how code should be addressed. See **#pragma section** for more information.
-Xaddr-user=*n* -X106=*n*	Specify how user defined sections should be addressed. See **#pragma section** for more information.
-Xansi -Xa -X7=1	Follow the ANSI C standard with some additions. This is the default in C mode. See Table A-2, "Features of Compatibility Modes for C Programs," on page 83 for details. This option is ignored when compiling C++ programs.
-Xargs-not-aliased -X65	Assume that pointer arguments to a function are not aliased with each other, nor with any global data. This enables greater optimization.
-Xascii-target -X60=1	Generate code for a target using the ASCII character set. All strings and character constants are converted to ASCII. The default is to use the same character system as the host machine. See also option -Xebcdic-target.
-Xauto-vtbl -X208=1	Specify that the virtual function table for a class is generated in only one file. This option is the default. C++ only.
-Xblock-count -X24	Insert code in the compiled program to keep track of the number of times each basic block (the code between labels and branches) is executed. See option -Xfeedback on page 23 and the D-BCNT Manual Page for details.

Table 3-3 Standard -X Options *(continued)*

X Option	Description

-Xbottom-up-init
-X21
	Both K&R and ANSI C specify that structure and array initializations with missing braces should be parsed top-down, however some C compilers parse these bottom-up instead:
	Example:
	`struct z { int a, b; };`
	`struct x {`
	`  struct z z1[2];`
	`  struct z z2[2];`
	`} x = { {1,2},{3,4} };`
	should be parsed according to ANSI & K&R as:
	`{ { {1,2},{0,0} } , { {3,4},{0,0} } };`
	-Xbottom-up-init causes bottom-up parsing:
	`{ {1,2},{3,4} } , { {0,0},{0,0} } };`
	This option is set when -Xpcc (-X7=3) is set.

-Xcall-MAIN -X211	Specify that the compiler should insert a call to a function **_MAIN( )** as the first operation in the **main( )** function. This is useful if no `crt0.o` (the startup module) is used. The default is to have `crt0.o` call the **__init_main( )** function that initializes global constructors, etc. This applies to C++ only.
-Xdollar-in-ident -X67	Allow dollar signs in identifiers.
-Xdouble-error -X96=1	Generate an error if any double precision operation is used. It will also force all double constants to single precision.
-Xdouble-warning -X96=2	Generate a warning if any double precision operation is used. It will also force all double constants to single precision.
-Xenum-is-best	Use the smallest integer type permitted by the range of the values for an enumeration without regard to sign. Thus, an enumeration with values from 1 through 128 will have base type **unsigned char** and will require one byte. Contrast with -Xenum-is-small.
-Xenum-is-int -X8	The **enum** type is always equal to **int**.
-Xenum-is-small -X8=0	Use the smallest **signed** integer type permitted by the range of values for an enumeration, i.e., the shortest of **char**, **short**, **int**, or **long** depending on the values of the enumeration constants. Thus, an enumeration with values from 1 through 128 will have base type **short** and require two bytes. Contrast with -Xenum-is-best.
-Xexception -X200	Enable exception handling. This is the default. C++ only. See also -Xno-exception.

Table 3-3 Standard -X Options *(continued)*

X Option	Description
-Xextend-args -X77	Extend all arguments to the precision given by whichever of -Xuse-double, -Xuse-long-double, or -Xuse-float is in force (all are settings of -X3), even if prototypes are used. (If none of the -X3 options are given, the default is -Xuse-double as that is equivalent to -X3=0).
-Xfeedback -Xfeedback=*filename* -X-1 -X-1=*filename* (The numeric form of this option number is -1.)	The compiler will use profiling information generated by the -Xblock-count option to create faster code. *filename* is the name of the profiling file. The default is `dbcnt.out`. To use this option: • Compile a program with -Xblock-count. • Run the program, which now creates `dbcnt.out` with profiling information. (See Chapter 8, "Use in an Embedded Environment," in the *Compiler Target User's Manual* for file I/O in an embedded environment.) • Recompile, now with the -XO -Xfeedback options to produce high-level speed optimized code.
-Xforce-declarations -X9	Generate warnings if a function is used without a previous declaration. This option is ignored when compiling C++ programs.
-Xforce-prototypes -X9=2	Generate warnings if a function is used without a previous prototype declaration. This option is ignored when compiling C++ programs.
-Xhi-mark=*n* -X68=*n* -Xlo-mark=*n* -X69=*n*	Change the parameters used to control optimization effort when using profile data. When using -Xfeedback, the compiler divides the basic blocks into three categories: code executed "frequently", "sometimes", and "seldom". It spends more effort optimizing code executed frequently, while it never does inlining, etc., on code executed seldom. The higher the thresholds, the more often code must be executed to get into the "frequent" category. The defaults are -Xhi-mark=1000 -Xlo-mark=10 and are used as follows: each execution of a basic block recorded in the profile counts as one "tick". The lo-mark and hi-mark values are normalized on a basis of 1,000,000 ticks. With the default values, code executed fewer than 10 times is marked "seldom", that executed from 10 to 1,000 times marked "sometimes", and that executed 1,000 or more times marked "frequent". Example: `-Xhi-mark=100 -Xlo-mark=10` means that more code will go into the "frequent" category, and the compiler will do more inlining etc., trading speed for space.

Table 3-3 Standard -X Options *(continued)*

X Option	Description
-Ximport -X75	Treat all **#include** directives as if they are **#import** directives. This means that any include file is only included once.
-Xinline=*m* -X19=*m*	Inline functions with fewer than *m* nodes. See -Xunroll-size on page 30 for a definition of node count. Since D-C++ collects functions until -Xparse-size KBytes of memory is used, the inlined function does not need to be defined before the function using it.
-Xinit-locals=*n* -X87=*n*	Initialize all local variables to zero or the value specified with -Xinit-value at every function entry. *n* is a bit mask specifying what kind of variables should be initialized: 0x1 - integers 0x2 - pointers 0x4 - floats 0x8 - aggregates If *n* is not given, all local variables will be initialized. This option is useful in finding "memory dependent" bugs.
-Xinit-value=*n* -X90=*n*	Define the initial value used by the -Xinit-locals option. This option can be useful to identify uninitialized variables, since it initializes variables to some invalid or recognizable value that might produce a memory access error.
-Xjmpbuf-size=*n* -X201=*n*	Set the size in bytes of the buffer that D-C++ allocates for **setjmp** and **longjmp** when using exception handling. The default size depends on the target. C++ only.
-Xk-and-r -Xt -X7=0	Follow the "C standard" as defined by the K&R C reference manual, but with all the new ANSI C features added. Where K&R and ANSI differ -Xk-and-r follows K&R. See Table A-2, "Features of Compatibility Modes for C Programs," on page 83 for details. This option is ignored when compiling C++ programs. See also -Xansi, -Xpcc, and -Xstrict-ansi.
-Xkeywords=*x* -X78=*x*	Recognize new keywords according to *x*, a bit mask specifying which keywords to add: 0x01 - **extended** 0x02 - **pascal** 0x04 - **inline** 0x08 - **packed** 0x10 - **interrupt** See Chapter 5, "Additions to ANSI C and C++," beginning on page 45 for more information on these keywords.

Table 3-3 Standard -X Options *(continued)*

X Option	Description
-Xkill-opt=*x* -X27=*x*	Turn off individual optimization. The number *x* is a decimal, octal (0nn) or hexadecimal (0xnn) number with one bit for each optimization. The -O or -XO flag must have been given. Reasons to turn off optimizations include: • Saving code space by turning off optimizations that increase space. • Trying to locate program or compiler bugs seen only with the optimizer turned on. • Testing the effect of different optimizations. Multiple optimizations can be turned off by OR-ing their values:, e.g., -Xkill-opt=0x100004. The various optimizations have the following numbers. See Chapter 7, "Optimizations," in the *Compiler Target User's Manual*, for more details: 0x00000002 - Tail recursion 0x00000004 - Inlining 0x00000008 - Argument address optimization 0x00000010 - Structure members to registers 0x00000020 - Local strength reduction 0x00000040 - Question-expression pop 0x00000080 - Assignment pop 0x00000100 - Simple branch optimization 0x00000200 - Space optimization 0x00000400 - Split optimization 0x00000800 - Constant and variable propagation 0x00001000 - Complex branch optimization 0x00002000 - Loop strength reduction 0x00004000 - Loop count-down Optimization 0x00008000 - loop unrolling 0x00010000 - Global common subexpression elimination 0x00020000 - Undefined variable propagation 0x00040000 - Unused assignment deletion 0x00080000 - Minor transformations 0x00100000 - Delayed register saving 0x00200000 - Register coloring 0x00400000 - Interprocedural optimizations 0x00800000 - Remove entry and exit code 0x01000000 - Use scratch registers for variables 0x02000000 - Extend optimization 0x04000000 - Loop statics optimization 0x08000000 - Loop invariant code motion 0x20000000 - Static function optimization

Table 3-3 Standard -X Options *(continued)*

X Option	Description
-Xkill-reorder=*x* -X28=*x*	Turn off individual optimizations in the `reorder` program. The list of optimizations is target-dependent. See Chapter 2, "Target Dependent Command Line Options," in the *Compiler Target User's Manual* for details. Multiple optimizations can be turned off by OR-ing their values.
-Xlint[=*x*] -X84[=*x*]	Generate warnings when suspicious and non-portable code is encountered. Individual warnings can be turned off by OR-ing the following values: 0x0000002 - Variable used before set 0x0000004 - Label not used 0x0000008 - Condition always true/false 0x0000010 - Variable/function not used 0x0000020 - Missing return expression 0x0000040 - Variable set but not used 0x0000080 - Statement not reached Note that -Xlint=1 turns all lint features on while -Xlint=0 turns them off
-Xlocals-on-stack -X5	By default, D-C++ attempts to allocate all local variables to registers. If -Xlocals-on-stack is given, only variables declared with the **register** keyword are assigned to registers.
-Xmin-align=*n* -X93=*n*	Set the minimum alignment that the target processor needs to access data in memory. If the target can access any unaligned data, *n* is set to 1. The default value of *n* is dependent on the processor.
-Xmemory-is-volatile -X4	Do not perform optimizations that can cause device drivers, etc., to fail. By default, D-C++ keeps data in registers as long as possible whenever it is safe. Difficulties can arise if a memory location changes because it is mapped to an external hardware device and D-C++, unaware of this change, continues to use the old value that is stored in a register. These situations can be handled with the keyword **volatile**. However, in order to allow for the compilation of older programs - D-C++ provides the -Xmemory-is-volatile option.
-Xmismatch-warning -X2	Generate a warning only (instead of a fatal error) when pointers of different types or integers are mixed in expressions. Example: `long i1, i2 = &i1;` is invalid in C, however older programs expect the compiler to handle this. This options is also set by -Xpcc (-X7=3). If the option -Xmismatch-warning=2 or -X2=2 is given, the compiler will also generate a warning instead of an error when identifiers are redeclared. This option is ignored when compiling C++ programs.

Table 3-3 Standard -X Options *(continued)*

X Option	Description
-Xno-bss -X83	Put all data in the **.data** section instead of allocating uninitialized data to the **.bss** section.
-Xno-digraphs -X202	Disables digraphs. If digraphs are enabled, the compiler recognizes the following keywords as digraphs: **bitand**, **and**, **bitor**, **or**, **xor**, **compl**, **and_eq**, **or_eq**, **xor_eq**, **not**, and **not_eq.** C++ only.
-Xno-double -X70=2	Force **double** and **long double** to be the same as the **float**. See also -Xno-long-double.
-Xno-exception -X200=0	Disable exception handling. Compiling a program with any of the keywords **try**, **catch**, or **throw** will cause a compilation error. Compiling with this option will reduce the stack space and increase the execution speed, when classes with destructors are used. C++ only. See also -Xexception.
-Xno-ident -X63	Do not pass **#ident** strings to the assembler.
-Xno-implicit-templates -X207	Instantiate templates only where the explicit instantiation syntax is used. See discussion about templates. C++ only.
-Xno-long-double -X70	Force **long double** to be the same as **double** on machines where they differ. See also -Xno-double.
-Xno-old-style -X203	Disable the use of old-style C function declarations. Most users probably want this option set, because it reduces the number of legal constructs available to the parser and therefore error messages may be more informative. C++ only.
-Xno-optimized-debug -X89	Disable most optimizations when using the -g option for use with debuggers that cannot handle optimized code.
-Xno-postfix -X206	Specify that the expression parser should always look for an **operator++**( ) or **operator--**( ) regardless of whether the operator was used as a prefix or postfix operator. An **operator++(int)** or **operator--(int)** may still be declared and used with explicit member function calls. This feature is available to ease compilation of old C++ code. C++ only.
-Xno-recognize-lib -X66	Cause the compiler to disregard all knowledge of ANSI C library functions.
-Xno-rtti -X205=0	Disable Run-time type information. Using this option will save some space since the compiler does not need to create type tables. C++ only. See also -Xrtti.

Table 3-3 Standard -X Options *(continued)*

X Option	Description
-XO -X26	A short-cut to specify more than the standard optimizations. -XO is equivalent to setting the following options: -O -Xparse-size=2000 -Xtest-at-both -Xinline=40 -Xopt-count=2 -Xrestart
-Xopt-count=*n* -X25=*n*	Execute the complier's optimizing stage *n* times. The default is once. In most cases this is enough. In rare instances one stage of the optimizer will generate an opportunity for a previous stage. Setting -Xopt-count=2 or more will cause a somewhat longer compilation time but may produce slightly better code. This option is set to 2 by -XO.
-Xparse-size=*m* -X20=*m*	Delay code generation of functions until *m* KBytes of main memory is used. By delaying generation, D-C++ can perform inter-procedural optimizations such as inlining and register tracking. The default is 500 KBytes.
-Xpass-source -X11	Output the source as comments in the generated assembly language code. Certain interprocedural optimizations will be turned off by this option.
-Xpcc -X7=3	Follow the C standard as defined by the Unix System V.3 C compiler. See Table A-2, "Features of Compatibility Modes for C Programs," on page 83 for details. This option is ignored when compiling C++ programs.
-Xrestart -X29	Restart optimization from scratch if too many optimistic predictions were made. Compilers may have difficulty predicting the best way to perform specific optimizations, since the information needed will not be available until a later compiler stage. For example, better code may be produced by moving a loop invariant expression outside the loop if the result can be placed in a register. However, the compiler does not know if any register is available until after register allocation, which is performed much later in the compilation. D-C++ uses an "optimistic" approach which generates optimal code when registers are available but not when all registers are taken. The -Xrestart option will restart optimization and code generation if any optimistic prediction is false. This will typically slow the compilation of large functions by a factor of almost two while generating better code. This option is turned on by -XO.
-Xrtti -X205=1	Enable Run-time type information. This is the default. C++ only.C++ only. See also -Xno-rtti.

Table 3-3 Standard -X Options *(continued)*

X Option	Description
-Xsigned-bitfields -X12=0	Handle bitfields without the **signed** or **unsigned** keyword as signed integers. See also -Xunsigned-bitfields.
-Xsigned-char -X23=0	Treat variables declared **char** without either of the keywords **signed** or **unsigned** as signed characters. See also -Xunsigned-char. In C++, plain **char**, **signed char** and **unsigned char** are always treated as different types, but this option defines how arithmetic with plain **char** is done. The default setting is target dependent. See Chapter 5, "Internal Data Representation," in the *Compiler Target User's Manual.*
-Xsize-opt -X73	Optimize for size rather than speed when there is a choice. Optimizations affected include inlining, loop unrolling, and branch to small code.
-Xsmall-const=*n* -X98=*n*	Place small constant static and global variables with a size less than or equal to *n* in the **SCONST** section class. See **#pragma section** for more information.
-Xsmall-data=*n* -X97=*n*	Place small non-constant static and global variables with a size less than or equal to *n* in the **SDATA** section class. See **#pragma section** for more information.
-Xstack-probe -X10	Enable stack checking (probing) on machines which support it. See Chapter 8, "Use in an Embedded Environment," in the *Compiler Target User's Manual.*
-Xstatic-addr-error -X81=2	Generate an error if the address of a variable, function, or string is used by a static initializer. This is useful when generating position independent code (PIC).
-Xstatic-addr-warning -X81=1	Generate a warning if the address of a variable, function, or string is used by a static initializer. This is useful when generating position independent code (PIC).
-Xstatic-vtbl -X208=0	Specify that the virtual function table for a class is generated in each file that references it. C++ only.
-Xstop-on-warning -X85	Treat warnings like errors and stop compilation.
-Xstrict-ansi -Xc -X7=2	Strictly follow the ANSI C standard. See Table A-2, "Features of Compatibility Modes for C Programs," on page 83 for details. This option is ignored when compiling C++ programs.
-Xstring-align=*n* -X18=*n*	Align strings on a multiple of *n*-byte boundaries. The default value is 4.
-Xstrings-in-text -X74	Locate all strings and **const** data in the **.text** section instead of the **.data** section. Identical string constants will use the same string space. This is the default in version 3.6 and later.
-Xstruct-arg-warning=*n* -X92=*n*	Emit a warning if the size of a structure argument is larger than *n* bytes.

Table 3-3 Standard -X Options *(continued)*

X Option	Description
-Xstruct-max-align=*n* -X88=*n*	Control the maximum alignment of structure members. If the natural alignment of a member is less than *n*, the natural alignment is used. See **#pragma pack** on page 47 and the **__packed__** keyword on page 55 for details.
-Xstruct-min-align=*n* -X76=*n*	Force structure alignments to be at least *n* bytes in size. If any member has a greater alignment, the highest value is used. See **#pragma pack** on page 47 and the **__packed__** keyword on page 55 for details.
-Xsuppress-warnings -X14	Same as the `-w` option. No warnings are generated.
-Xswap-cr-nl -X13	Swap `'\n'` and `'\r'` in character and string constants. Used on systems where carriage return and line feed are reversed.
-Xtest-at-both -X6=2	Force D-C++ to always test loops both before the loop is started and at the bottom of the loop. This option produces the fastest possible code but uses somewhat more space. Even if -Xtest-at-both is not set, other optimizations may cause D-C++ to generate double tests. This option is turned on by -XO.
-Xtest-at-bottom -X6=0	Use one loop test at the bottom of a loop.
-Xtest-at-top -X6=1	Use one loop test at the top of a loop.
-Xtruncate=*m* -X22=*m*	Truncate all identifiers after *m* characters. If *m* is zero, no truncation is done. This is the default.
-Xunroll=*m* -X15=*m*	Unroll small loops *m* times. Set to 2 as a default if `-O` is given. *m* must be a power of two. See Chapter 6, "Optimization Hints," beginning on page 59.
-Xunroll-size=*m* -X16=*m*	Specify the maximum number of nodes a loop can contain to be considered for loop unrolling. Each operator and each operand counts as one node, so the expression `a=b-c;` contains 5 nodes. *m* is set to 20 as a default if `-O` is given.
-Xunsigned-bitfields -X12	Handle bitfields without the **signed** or **unsigned** keyword as unsigned integers. This is the default setting. See also -Xsigned-bitfields.
-Xunsigned-char -X23	Treat variables declared **char** without either of the keywords **signed** or **unsigned** as unsigned characters. See also -Xsigned-char.
-Xuse-double -X3=0	Use **double** as the minimum precision in expressions and for floating point arguments. Lesser precisions are used in expressions if the -Xansi option is used. If prototypes are used, D-C++ uses the declared precision for arguments, unless the -Xextend-args option is used.

Table 3-3 Standard -X Options *(continued)*

X Option	Description
-Xuse-float -X3=1	Use **float** as the minimum precision in expressions and for floating point arguments.
-Xuse-.init -X91	Create a **.init** section that includes code to call all initialization code, i.e. global constructors in C++. This is the default.
-Xuse-long-double -X3=2	Use **long double** as the minimum precision in expressions and for floating point arguments. Lesser precisions are used in expressions if the -Xansi option is used. If prototypes are used, D-C++ uses the declared precision for arguments, unless the -Xextend-args option is used.
-Xwchar=*n* -X86=*n*	Define the type that **wchar** should correspond to. For values of *n*, see the **sizeof(type,2)** operator in Additions to ANSI C.

Examples of processing two source files

The following examples show typical ways of compiling with D-C++.

The two files, `file1.cpp` and `file2.cpp`, contain the source code:

```
/* file1.cpp */

void outarg(char *);

int main(int argc, char **argv)
{
   while(--argc) outarg(*++argv);
   return 0;
}
```

```
#include <iostream.h>

void outarg(char *arg)
{
   static int count;

   cout<<"arg #"<<++count<<": "<<arg<<endl;
}
```

Compile and link

When compiling small programs such as this, D-C++ can be invoked to execute all four stages of compilation in one command. See "The dplus command" on page 11.

```
dplus file1.cpp file2.cpp
```

D-C++ preprocesses, compiles, and assembles the two files, and links them together with the C++ library to create a single executable file, by default called `a.out`. As a consequence when more than one file is compiled to completion, object files called `file1.o` and `file2.o` are created.

If the target system support command line execution, to execute this program enter `a.out` with some arguments:

```
a.out abc def ghi
```

This will print:

```
arg #1: abc
arg #2: def
arg #3: ghi
```

(See Chapter 8, "Use in an Embedded Environment," in the *Compiler Target User's Manual* for comments on executing programs in embedded environments.)

To give the generated program a different other than `a.out`, use the -o option:

```
dplus file1.cpp file2.cpp -o prog1
```

To also enable optimization, use the -O option:

```
dplus -O file1.cpp file2.cpp -o prog1
```

Generally speaking, the -O (or -XO) options should be invoked whenever compiling an important program, because your program should be running at full speed at all times, even when optimizations slightly increase compilation time.

To convert the linked output to S records:

```
ddump -Rv a.out
```

will produce file `srec.out` by default. See the *Utilities User's Manual* for additional options and details.

Separate compilation

When compiling programs consisting of many source files, it is time consuming and impractical to recompile the whole program whenever a file is changed. Separate compilation is a time-saving solution when recompiling larger programs. The -c option creates an object file which corresponds to every source file, but does not call the linker. These object files can then be linked together later into the final executable program. When a change has been made, only the altered files need to be recompiled. To create object files and then stop, use the following command:

```
dplus -O -c file1.cpp file2.cpp
```

The files `file1.o` and `file2.o` will be created.

Create the executable program with:

```
dplus file1.o file2.o -o prog2
```

If `file2.cpp` is altered, `prog2` is rebuilt with:

```
dplus -O -c file2.cpp
dplus file1.o file2.o -o prog2
```

Usually, the compilation process is automated with utilities similar to `make`, which finds the minimum command sequence to create an updated executable.

Assembly output

It is frequently desirable to look at the generated assembly code. The -S option is used for this purpose.

```
dplus -O -S file1.cpp
```

This command creates `file1.s` which consists of the compiler's assembly output. The option -Xpass-source outputs the compiled C++ source as comments in the generated file and makes it easier to see which assembly instructions correspond to each line of source:

```
dplus -O -S -Xpass-source file1.cpp
```

➤ The -Xpass-source option disables some interprocedural optimizations which would otherwise generate functions in a different order than that given in the source file.

4 Configuration File

One or more *configuration files* are normally read by the main driver program `dplus` when it starts. This section describes the D-C++ configuration file and language.

The purpose of a configuration file is to provide values for options to be used by D-CC in processing all source files. Options may be given literally, and may also be constructed from constant strings and variables.

How command lines, environment variables, and configuration files relate

In theory, D-C++ could be executed with no options on the command line, no configuration file, and no environment variables set. In that case, all options would have their default values as described in Table 3-2 on page 14 andTable 3-3 on page 21.

In practice, D-C++ is often executed with a few options on the command line, perhaps a few options set with environment variables, a number of site-dependent defaults set in configuration files, and the remaining options having their default values.

Variables and precedence

Variables may be set in three places:

- in the operating system environment (see "Environment variables" on page 8);
- on the command line using the -WD option for any variable, the -WC option for variable `DCONFIG`, and the -t option to implicitly set variables `DTARGET`, `DOBJECT`, and `DFP`.
- in configuration files using assignment statements.

These are in order of precedence from lowest to highest: a variable defined on the command line overrides an environment variable of the same name, and a variable set in a configuration overrides both a command line or an environment variable of the same name. (Thus, in a configuration file, it is usual to test whether a variable has a value before assigning it a default value – see examples below.)

D-C++ startup

Here is how D-C++ processes the command line and configuration files at startup.

➤ Order is important. If a variable is given a value or an option occurs more than once, the last instance is taken unless noted otherwise.

1. D-C++ scans the command line for an -@ option followed by the name of either an environment variable or a file, and replaces the designated item with its contents.
2. D-C++ scans the command line for each -WD *variable=value* option. As noted above, If a variable matches an existing environment, the new value effectively replaces the existing value for the duration of the command (the actual operating system environment is not changed). If the same variable name appears more than once (perhaps because of an -@ option expansion), the final value will be used.

   The option -WC *config-file-name* is equivalent to -WDDCONFIG=*config-file-name*. Thus, if both -WC and -WDDCONFIG options are present, the *config-file-name* will

be taken from the final instance, and if either is present, they will cover up any DCONFIG environment variable.

3. D-C++ finds the main configuration file by checking first for a value of variable DCONFIG, and then if not set, going to the standard location as given in Table 4-1, "Main Configuration File: Standard Name and Location," on page 37. D-C++ parses each statement in the configuration file as described in the following subsections.

4. After parsing the configuration file (or files if **include** statements are encountered), D-C++ processes each of the input files on the command line using the options set by command line and configuration file processing.

Figure 4-1 below provides a simplified example of how the above works.

The remainder of this chapter provides additional details and examples and explains each of the statements allowed in a configuration file.

Figure 4-1 Example of Command Line and Configuration File Processing

Situation

An engineer's works on Project 1 which normally uses 68000 targets and standard optimization (the -O option). Today, the engineer has a 68040 prototype, and wants to use extended optimization (-XO).

Environment Variables (set using operating system commands not shown)

```
DCONFIG:  project1.con
DFLAGS:   -O
```

As described in Table 2-4, "Environment Variables Recognized by the Compiler," on page 8, DFLAGS is a convenient way to give widely used options.

Command Line

```
dcc -WDDTARGET=MC68040 -XO test1.c
```

The command line is used to select the special processor and extended optimization.

Excerpts from Configuration File project1.con

```
if (!$DTARGET) DTARGET=MC68000
...
$DFLAGS
$*
```

If the target had not been set on the command line, it would default to the 68000.

$DFLAGS evaluates to -O. $* is a special variable evaluating to all of the command line arguments (the -WD option on the command has already been processed and is ignored the second time). The -XO option from the command line does not conflict with the -O option from the DFLAGS environment variable – because -XO is a superset of -O, it effectively takes precedence.

Standard configuration files

Diab Data recommends the use of three configuration files in a hierarchy. Standard versions of two of these, `dtools.conf` and `default.conf` are shipped with D-C++. (DOS users, please substitute extension `.CON` throughout this section.)

D-C++ identifies the main configuration file using the DCONFIG variable as described in steps 2 and 3 in "D-C++ startup" on page 35. If DCONFIG is not set, then D-C++ looks for the file `dtools.conf`. Its standard location is the `conf` subdirectory of the directory holding the selected version of the tools as shown in the following table (see also Table 2-1, "Example Default Installation Path Names," on page 5).

Table 4-1 Main Configuration File: Standard Name and Location

System	Path and Name
UNIX	`/usr/lib/diab/`*version*`/conf/dtools.conf`
MS-DOS	`C:\DIAB\`*version*`\CONF\DTOOLS.CON`
MPW	`{MPW}diab:`*version*`:conf:dtools.conf`

The standard location of the main configuration file can be changed by setting the `DCONFIG` environment variable, by using the -WC option, or by using the -WDDCONFIG option.

The standard `dtools` file is structured broadly as shown in Figure 4-2. A study of `dtools` will help show how the compiler combines the various environment variables and command line options. `dtools` also serves as an extended example of how to write the configuration language.

As noted at the beginning of `dtools`, you should avoid altering `dtools`, and should instead set defaults and specific options by using `dctrl` and/or the -t option on the command line to set `DTARGET`, `DOBJECT`, and `DFP` in `default.conf` (see "Using D-C++ for different targets" on page 9), or otherwise modifying `default.conf`, and/or by providing your own `user.conf`.

Figure 4-2 Standard `dtools` Configuration File - Simplified Structure

- Variables used to customize selection and operation of the tools.
- `include default.conf` — Read the second of the two configuration files included with the D-C++. It is intended for default values of options related to selection of the target such as `DTARGET`. Use `dctrl` and/or the -t command line option (both preferred, see "Using D-C++ for different targets" on page 9), or change this file to specify the defaults for
  - the target processor
  - the mnemonic type
  - the software or hardware floating point mode
- `include user.conf` — This file is not always provided with the compiler, but if it exists, it is intended for the following types of options:
  - default include files and libraries to use
  - default C compatibility mode (ANSI, PCC, etc.)
  - default preprocessor macros to be predefined
  - default optimizations to be executed
  - default -X options
- At this point, both DTARGET and DOBJECT will have been set. Next come extensive switch and other statements to set options and flags, especially with respect to different targets, and to select the tools if not customized above.
- dtools ends with the following variables in order:

`$UFLAGS1`	Standard options that should "always" be used unless overridden by `$UFLAGS2` (see page 39).
`$DFLAGS`	As described in Table 2-4 on page 8, `DFLAGS` is a convenient way to give widely used options.
`$*`	All arguments from the command line (except -WD and -WC options are not re-processed).
`$UFLAGS2`	Overrides for `$UFLAGS1` (see page 39).

As shown if Figure 4-2, the standard `dtools` configuration file begins (nearly) by including `default.conf`, and then including `user.conf`. These files must be located in the same directory as `dtools.conf` (no path is allowed on **include** statements in configuration files). If you want a private copy of these files, copy all the configuration files to a local directory and change the location of `dtools.conf` as described at the beginning of this section.

No error is reported if an **include** statements names a non-existent file; therefore, both files are optional.

UFLAGS1, UFLAGS2, DFLAGS, and command line options

Configuration file processing gives you several ways to provide options. The standard configuration files shipped with D-C++ are intended to be used as follows:

- `UFLAGS1` and `UFLAGS2` are intended for options that should "always" be used. It is intended that `UFLAGS1` and `UFLAGS2` be set in a local configuration file, `user.conf`, that you supply. Since you will not want to change this frequently, options set there will be "permanent" unless overridden.

  As shown in Figure 4-2, `UFLAGS1` is expanded before user supplied options and files, and `UFLAGS2` after.

  Example: to make sure that the lint facility is always on and that the compiler checks for prototypes, create a `user.conf` with the following lines:

  ```
  # File: user.conf
  # Always perform lint + check for prototypes.
  UFLAGS1=-Xlint -Xforce-prototypes
  ```

  And if there is a site-wide `user.conf`, the tools administrator can make sure that any user using it will not require too much memory by adding the following to `user.conf`:

  ```
  # Limit memory for optimization.
  UFLAGS2=-Xparse-size=1000
  ```

- `DFLAGS` is intended to be an environment variable for options that change more frequently than those in the configuration files, but not with every compile. For example, it may be conveniently used to select levels for optimization and debugging information.

- The command line is for options for a specific compilation. It overrides any options set with `UFLAGS1` and `DFLAGS`, but not `UFLAGS2` since `UFLAGS2` occurs after `$*` in `dtools.conf`.

The configuration language

As noted above, the ultimate purpose and effect of configuration file processing is to provide values for options. The simplest type of configuration file is an ordinary text file containing multiple lines where each line sets a single option.

Beyond this, a straight-forward *configuration language* allows great control over configuration file processing, so that different options and their values may be set depending on options present on the command line, on environment variables, and on variables defined by the user within a configuration file or a file included by a configuration file.

The remainder of this section describes the configuration language and ends with an extended example.

Statements, options, and comments

A configuration file consists of a sequence of *statements* and *options* separated by whitespace. A # token at any point on a line not in a quoted string introduces a comment; the rest of the line is ignored. Thus, a line may contain multiple statements and options ending in a comment.

A *statement* is either an assignment statement or starts with one of the keywords **error**, **exit**, **include**, **if** (and **else**), **print**, or **switch** (and **case**, **break**, and **endsw**).

In general, it is preferable to write one statement or option per line. This makes a configuration file easier to understand and modify. An exceptions to this rule is made for lines containing an **if** or **else** statement, each of which governs the remaining statements and options on a line as described below.

Whitespace, consisting of spaces or tabs, may be used freely between statements and/or options for readablility. Blank lines are ignored.

D-CC does not allow a line to be continued to a second line, but there is no practical limit on the length of a line except that which may be imposed by an operating system or text editor.

Any other text which is not a statement or comment per the above is taken as options to D-CC. In general, options are in one of four forms, each introduced by a single character option letter *x*:

-x
-x name
-x value
-x name=value

Either the name or the value may a quoted or unquoted string of characters as otherwise allowed by a particular option, and either may include variables introduced by a '$' character (see"Variables" below).

Examples:

```
-O
-XO                          "O" is a name
-o test.asm                  "test.asm" is a value
-Xkill-opt=0x100004
-I$HOME/include              $HOME is a variable
```

Comments

A # token at any point on a line not in a quoted string introduces a comment; the rest of the line is ignored.

Examples:

```
.......... # This is a comment through the end of the line.
not_a_comment = "# This is an assignment, not a comment"
```

String constants

A string constant is any sequence of characters ending in whitespace (spaces and tabs) or at line-end. To include whitespace in a string constant, enclose the entire constant in double quotes. There is no practical length limit except that imposed by the maximum length of a line.

Examples:

```
Simple_string_constant
"string constant with embedded spaces"
```

Variables

All variables are of type string. Variable names are any sequence of letters, digits and underscores beginning with a letter or underscore (letters are 'A' - 'Z' and 'a' - 'z', digits are '0' - '9'). There is no practical length limit to a variable name except that imposed by the maximum length of a line.

Variables are case sensitive.

To access the value of a variable, precede it with a '$' character. See "Variables and precedence" on page 35 for a discussion of *environment* versus *internal* variables and their precedence.

Variables are not declared. A variable which has not been set evaluates to a zero-length string equivalent to `""`.

The special variable **$*** evaluates to all arguments given on the command line (however -WD and -WD arguments have already been processed and are effectively ignored). See examples below.

The special variable $-*x*, where *x* is a one or more characters, evaluates to any user specified option starting with *x*, if given previously (on the command line or in the configuration file). Otherwise it evaluates to the zero-length string. If more than one option begins with *x*, only the first is used.

For example, if the command line includes option `-Dtest=level9`, then `$-Dtest` evaluates to `-Dtest-level9`.

The special variable **$$** is replaced by a dollar sign '$'.

The special variable **$/** is replaced by the directory separation character on the host system: '/' in UNIX, '\' in MS-DOS, and ':' in MPW. (On any specific system, you can just use the appropriate character. Diab Data uses $/ for portability.)

Examples: assume that the environment variable DVERSION is set to "3.7a", and that the following command is given:

```
dcc -Dlevel99 -g2 -O -WDDFP=soft file.c
```

Table 4-2 Variable Evaluation in Configuration Files

Variable	Evaluates To	Comment (see assumptions above)
`$DVERSION`	`"3.7a"`	An environment variable.
`$DFP`	`"soft"`	Because the action of -WD is as if it set an environment variable (see option -W *c name* on page 18).
`$-WDFP`	`"-WDDFP=soft"`	In the form $-*x*, *x* is the entire WD option.
`$-Dlevel`	`"-Dlevel99"`	In the form $-*x*, *x* need match only the beginning of an option.
`$*`	`"-Dlevel99 ... file.c"`	Evaluates to the entire command minus the initial `dcc`.

Assignment statement

The assignment statement assigns a string to a variable. Its form is:

variable = string-constant

Examples:

```
XLIB=$HOME/lib          # variable XLIB is set
YFLAGS="$XFLAGS -X12"   # must use "" for spaces in a string
if (...) PF=-p GF=-g    # two on one line (see if below)
```

Error statement

The error statement causes D-C++ to terminate with an error. See the **switch** statement for an example.

Exit statement

The exit statement causes D-C++ to stop configuration file processing. This is useful, for example, in an include file that specifies all compiler options, but does not want the compiler to continue the parsing in `default.conf` and `dtools.conf`.

If statement

The **if** statement provides for conditional branching in a configuration file. There are two forms:

if (*expression*) *statements* and/or *options*

and

if (*expression*) *statements* and/or *options*
else *statements* and/or *options*

If *expression* is true, the rest of the same line is interpreted and, if the next line begins with **else**, the remainder of that line is ignored. If *expression* is false, the remainder of the line is skipped, and, if the next line begins with **else**, the remainder of that line is interpreted. Blank lines are not allowed between **if** and **else** lines.

expression is one of:

Note that because any statement can follow **else**, one may write a sequence of the form

if
else if
else if
...
else

Examples:

```
if (!$LIB) LIB=/usr/lib      # if LIB is not defined, set it

if ($OPT == yes) -O          # option -O if OPT is "yes"
else -Xkill-opt=0xffffffff   # else option -Xkill-opt=...
```

string	true if *string* is non-zero length
!*string*	true if *string* is zero length.
string1 == *string2*	true if *string1* is equal to *string2*.
string1 != *string2*	true if *string1* is not equal to *string2*.

Include statement

The **include** permits nesting of configuration files. Its form is:

include *filename*

The contents of file *filename* are parsed as if inserted in the place of the **include** statement. The file must be located in the same directory as the main configuration file as no path is allowed in **include** statements (see "Standard configuration files" on page 37). If the given file does not exist, the statement is ignored.

Example:

```
include user.con
```

Print statement

The print statement outputs a string to the terminal. Its form is:

print *string*

Example:

```
if (!$DTARGET) print "Error: DTARGET not set"
```

Switch statement

The switch provides for multi-way branching based on patterns. It has the form:

switch(*string*)
case $pattern_1$**:**
    ...
    break
case $pattern_n$**:**
    ...
endsw

where each *pattern* is any string, which can contain the special tokens '?' (matching any one character), '*' (matching any string of characters, including the empty string) and '[' (matching any of the characters listed up to the next ']'). When a **switch** statement is encountered, the **case** statements are searched in order to find a pattern that matches the *string*. If such a pattern is found, interpretation continues at that point. If no match is found, interpretation continues after the **endsw** keyword. If more than one *pattern* matches the *string*, the first will be used.

If a **break** statement is found within the case being interpreted, interpretation continues after **endsw**. If no **break** is present at the end of a case, interpretation falls through to the next case.

Example:

```
switch($DTARGET)
case MC68*:         # any DTARGET beginning with MC68
   ...
   break
case MC88*:         # any DTARGET beginning with MC88
   ...
   break
case *:             # any other DTARGET
   print "Error: DTARGET not set"
   error
endsw
```

5 Additions to ANSI C and C++

This section describes additions to the ANSI C and C++ standards implemented in D-C++. Target-dependent additions are described in the *Compiler Target User's Manual*

Predefined macros

A set of predefined preprocessor macros are defined by D-C++. The macros not starting with two underscores (__) will not be defined if the -Xstrict-ansi option is given:

Table 5-1 Predefined Macros

__DATE__	The current date in "M*mm dd yyyy*" format. It cannot be undefined.
__DCC__	The decimal constant 1.
__DCPLUSPLUS__	The decimal constant 1 in D-C++. Only defined when compiling in C++ mode.
__cplusplus	The constant 1 when compiling C++ code otherwise undefined.
__STDC__	The constant 0 if -Xansi and the constant 1 if -Xstrict-ansi is given in C mode. It cannot be undefined if -Xstrict-ansi is set. It is never defined in C++ mode.
__STRICT_ANSI__	The constant 1 if -Xstrict-ansi.
__FILE__	The current file name. It cannot be undefined.
__LINE__	The current source line. It cannot be undefined.
__TIME__	The current time in "*hh:mm:ss*" format. It cannot be undefined.
__LDBL__	The constant 1 if the type `long double` is different from `double`.

Other target dependent predefined macros are described in the *Compiler Target User's Manual.*

Pragmas

This section describes the pragmas supported by D-C++. See also the *Compiler Target User's Manual* for target-dependent pragmas. Pragmas that are not recognized are ignored without warning.

➤ **Important:** Please see Table 1-1, "Document Conventions," on page 4 for the meaning of the [ ], { }, ..., and other forms employed below.

In C++ programs, a function named in a pragma effects all functions with the same name, independently of the types and number of parameters, that is, independently of overloading.

inline

#pragma inline *func ,...*

The **inline** pragma causes the given function to be inlined whenever possible. It must be specified before the definition of the function.

In C++ programs, the **inline** function specifier is normally used instead. This specifier will however also make the function local to the file, without external linkage. The **#pragma inline** statement on the other hand provides a hint to inline the code directly to the code optimizer, without any effect on the linkage scope.

Example:

```
#pragma inline swap

void swap(int *a, int *b) {
   int tmp;
   tmp = *a; *a = *b; *b = tmp;
}
```

interrupt

#pragma interrupt *function ,...*

Designates *function* as an interrupt function. Code is generated to save all scratch registers and use a different return instruction. If a processor has special floating point registers and these are not to be saved, use the software floating point version of the compiler (see "Using D-C++ for different targets" on page 9).

no_alias

#pragma no_alias { *var1* | **var2* } ,...

Promises that the variable *var1* is not accessed in any other manner (through pointers etc.) than through the variable name. Promises that the data at **var2* is only accessed through the pointer *var2*. Helps the compiler generate better code.

Example:

```
add(double *d, double *s1, double *s2, int n)
#pragma no_alias *d, *s1, *s2
{
   int i;

   for(i = 0; i < n; i++) {
      /* "s1 + s2" will move outside the loop */
      d[i] = *s1 + *s2;
   }
}
```

Without the **pragma**, either `s1` or `s2` might point into `d` and the assignment might then set `s1` or `s2`.

no_side_effects

#pragma no_side_effects *descriptor ,...*

where each *descriptor* has one of the following forms and meanings:

function	Promises that *function* does not modify any global variables (it may use global variables).
function ({ *global* | *n* } ,...)	Promises that *function* does not modify any global variables except 1) those named, and/or 2) the data addressed by its *n*th parameter. At least one global or parameter number must be given, and there may be more than one of either kind in any order.

Example:

```
#pragma no_side_effects strcmp(1), sin(errno), \
          my_func(1, 2, my_global)
```

pack

#pragma pack [(*max*, *min* [, *byte-swap*])]

The **pack** directive specifies that all subsequent structures are to use the alignments given by *max* and *min* where:

max — Specifies the maximum alignment of any member in a structure. If the natural alignment of a member is less than or equal to *max*, the natural alignment is used. If the natural alignment of a member is greater than *max*, *max* will be used.

Thus, if *max* is 8, a 4-byte integer will be aligned on a 4-byte boundary. While if *max* is 2, a 4-byte integer will be aligned on a 2-byte boundary.

min — Specifies the minimum alignment of the entire structure itself, even if all members have an alignment that is less than *min*.

byte-swap — If 0 or absent, bytes are taken as is. If 1, bytes are swapped when the data is transferred between byte-swapped members and registers or non-byte-swapped memory. This enables access to little-endian data on a big-endian machine and vice-versa.

The following restrictions apply to byte-swapped data:

- Only structure members of an integer type (short, long, or long long, either signed or unsigned), can be byte-swapped.
- It is not possible to take the address of a byte-swapped member.

If neither *max* nor *min* are given, they are both set to 1. If either *max* or *min* is zero, the corresponding default alignment is used. If *max* is non-zero and *min* is not given it will default to 1.

An alternative method of specifying structure padding is by using the **__packed__** keyword (see page 55).

➤ A target may be limited in the alignments which it supports. The compiler will insert extra padding to assure that no alignment exception occurs when accessing members of a structure.

Examples:

```
#pragma pack(1)          same as #pragma pack(1,1), no padding

struct S1 {
  char c1;               1 byte at offset 0
  long i1;               4 bytes at offset 1
  char d1;               1 byte at offset 5
};                       total size 6, alignment 1

#pragma pack(8)          use natural alignments up to 8

struct S2 {
  char c2;               1 byte at offset 0, 3 bytes padding
  long i2;               4 bytes at offset 4
  char d2;               1 byte at offset 8
                         3 bytes padding
};                       total size 12, alignment 4

#pragma pack(2,2)        typical 68k packing

struct S3 {
  char c3;               1 byte at offset 0, 1 byte padding, 1 byte padding
  long i3;               4 bytes at offset 2
  char d3;               1 byte at offset 6, byte padding
};                       total size 8, alignment 2

struct S4 {
  char c4;               1 byte at offset 0, 1 byte padding
};                       total size 2, alignment 2 since min is 2

#pragma pack(8)          natural packing

struct S {
  char e1;               1 byte at offset 0
  struct S1 s1;          6 bytes at offset 1, 1 byte padding
  struct S2 s2;          12 bytes at offset 8
  char e2;               1 byte at offset 20, 1 byte padding
  struct S3 s3;          8 bytes, at offset 22, 2 bytes padding alignment 2
};                       total size 32, alignment 4

#pragma pack(0)          use default padding
```

pure_function

#pragma pure_function *function* ,...

Promises that each *function* does not modify or use any global or static data. Helps the compiler generate better code.

Example:

```
#pragma pure_function sum
int sum(int a, int b) {
   return a+b;
}
```

no_return

#pragma no_return *function ,...*

Promises that each *function* never returns. Helps the compiler generate better code.

Example:

```
#pragma no_return exit, abort, longjmp
```

section

#pragma section *class_name* [*istring* [*ustring*]] [*addr_mode*] [*acc_mode*] [**address=***x*]

The **#pragma section** directive controls the sections into which variables and code are placed. It also controls how the variables should be accessed. See the *Compiler Target User's Manual* for implementation details.

use_section

#pragma use_section *class_name variable ,...*

Selects the section into which a variable or function is placed. See **#pragma section** for more information.

asm and __asm

The **asm** and **__asm** keywords provide a way to include assembly code within a C++ program. Both keywords have exactly the same functionality, but **asm** is not defined if the -Xstrict-ansi option is given. In the text below, whenever **asm** is mentioned, **__asm** can be used instead.

There are two ways of using the **asm** keyword. The first is a simple way to pass a string to the assembler, an **asm** string. The second is an advanced method to define an **asm** macro that inlines different assembly code sections, depending on the types of arguments given.

asm strings

An **asm** string can be specified wherever a statement or an external declaration is allowed. It must have exactly one argument, which should be a string constant to be passed to the assembly output. Some optimizations will be turned off when an **asm** string statement is encountered.

See Chapter 8, "Use in an Embedded Environment," in the *Compiler Target User's Manual* for examples.

asm macros

The asm strings mentioned above can be useful when inlining simple assembly code fragments, but are difficult to use with C++ variables inside the assembly code. **asm** macros provide a flexible way to interface to assembly code in C++ programs.

An **asm** macro definition looks like a function definition in which the body of the function is replaced with an assembly code sequences. There are three cases.

Empty asm macro definition

asm [*return-type*] *macro-name* ([*parameter-list*])
{
} (must start in column 1)

The *return-type* and *parameter-list* are as for standard C functions. The *parameter-list* may be declared in old C-style using just names follows by separate type declarations, or

in prototype-style with both a type and name for each parameter. Only prototype-style is shown below.

The compiler discards any invocation of an empty macro. This may be useful for macros normally used for debugging purposes.

Single-body asm macro

An **asm** macro with a single asm-body is similar to a standard C function.

```
asm [return-type] macro-name ( [ parameter-list ] )
{
%storage-mode-line                    (must start in column 1)
    asm-body
}                                     (must start in column 1)
```

The *return-type* and *parameter-list* are as for standard C functions (see "Empty asm macro definition" on page 49).

The *storage-mode-line* describes the method used for passing each parameter to the macro. Every parameter name in the *parameter-list* must occur exactly once in the *storage-mode-line*. The form of the *storage-mode-line* is:

%storage-mode parameter, ...; storage-mode parameter, ... ; ...

storage-mode must be one of the keywords given in Table 5-2 on page 50. Additional rules follow the table.

The *asm-body* is a sequence of assembly language instruction lines. See the *Compiler Target User's Manual* for examples.

Table 5-2 Storage Modes for Parameters to Assembler Macros

storage_mode[a]	Description
reg	The parameter is in a non-scratch register.
con	The parameter is a constant.
mem	The parameter is any allowed addressing mode, including **reg** and **con**.
ureg	The same as **reg**.
treg	Never matched. Included for compatibility.
lab *name*	A new label is generated. The identifier following **lab** is not a parameter (a **lab** identifier is not allowed as a parameter), it is a label used in the assembly code body. For each use of the macro, D-C++ will generate a unique label to substitute for the uses of the *name* in the macro.
error	A compile time error is generated. (Most useful for Case 3.) The **error** storage mode does not take a parameter name.

a. Storage modes are target-dependent. See the *Compiler Target User's Manual.*

Rules:

- Both the '**%**' character that begins a line of *storage-mode-lines*, and the final closing brace '**}**' of the **asm** macro must be in column 1 of their respective lines.

Whitespace is not allowed because an assembler syntax supported by D-C++ may use '%' and/or '}'for other purposes.

- Every parameter must be declared in exactly one *storage-mode-line*.
- If an **asm** macro has a type, the value to be returned must be put by the assembly code in an appropriate register, depending on the calling conventions.
- The compiler treats an **asm** macro as an ordinary function with unknown properties:
    - All temporary registers can be used by the function. The compiler ensures that parameters never use any temporary registers to avoid collisions.
    - Any global or static variable can be modified.
    - **#pragma** directives can be used to tell the compiler if the function has any side effects, etc.
    - Parameters should not be modified because the compiler has no way to detect this and some optimizations will fail if a parameter is modified.

Multiple-body asm macro

An asm macro with multiple *%storage-mode-line / asm-body* pairs overloads the macro definition in a manner similar to that of an overloaded C++ function (this is valid whether in a C or C++ module).

```
asm [return_type] macro_name ( [ parameter_name ,... ] )
{
%storage-mode-line                    (must start in column 1)
    asm-body
        .
        .
        .
%storage-mode-line                    (must start in column 1)
    asm-body
}                                     (must start in column 1)
```

The compiler chooses one of the bodies based on the types of arguments provided when invoking the **asm** macro. For each invocation of the macro, the compiler searches all *storage-mode-lines* in order. It selects the first body for which there is an exact match between the storage of the actual arguments passed to the macro in that invocation, and the description given by the *storage-mode-line* for that body.

If no matching *storage-mode-line* can be found, the compiler reports an error.

➤ On some RISC processors (e.g., PowerPC and MC88000), all parameters are put in registers. If the parameter is not already in a register, it will be placed in the same temporary register it would normally use for a function call. Note that this implies a limit on the maximum number of parameters.

See Chapter 8, "Use in an Embedded Environment," in the *Compiler Target User's Manual* for examples.

assert and unassert

The **#assert** and **#unassert** preprocessor directives provide a way to define preprocessor variables without conflicting with names in the program name space. These variables can be used to direct conditional compilation.

An assert can be made in two different ways. Both methods assign a named value to a named preprocessor variable.

1. As a preprocessor directive (whitespace is allowed only where shown):

   #assert #*ident1*(*ident2*)

   Example:

   ```
   #assert #system(unix)
   ```

2. 2. As a command line option:

   -A*ident1* (*ident2*)

   Example:

   UNIX: `dplus "-Asystem(unix)" test.c`

   DOS or MPW: `dplus -Asystem(unix) test.c`

These forms associate *ident2* with *ident1* as an assertion. Assertions can be tested in either an **#if** or an **#elif** preprocessor directive with the syntax:

#if #*ident1*(*ident2*)

Example:

```
#if #system(unix)
```

This becomes true if the corresponding assertion is true.

An assertion can be removed with the **#unassert** directive:

#unassert #*ident1*(*ident2*)

Example:

```
#unassert #system(unix)
```

Direct functions

A direct function is a less graceful technique for inlining machine code instead of doing a function call. It is supported by D-C++ for compatibility reasons, but should be avoided because enhanced **asm macros** provide a much more flexible method to do the same thing. See "asm macros" on page 49.

In a direct function definition, the body of the function is replaced by a list of integers which represent the machine code. When calling a direct function, the actual branch to the subroutine is replaced by these machine instructions. Otherwise normal calling conventions are followed.

Dynamic memory allocation with alloca

The **alloca(size)** function is provided to dynamically allocate temporary stack space inside a function.

Example:

```
char *p, *alloca();

p = alloca(1000);
```

The pointer p will point to an allocated area of 1000 bytes on the stack. This area is only valid until the current function returns. Note that the use of **alloca( )** will typically increase the necessary entry/exit code needed in the function and will turn off some optimizations such as tail recursion.

If the option -Xinline-alloca is specified, calls to alloca( ) will be replaced with very fast inline code.

__ERROR__

The **__ERROR__( )** function is recognized by D-C++ and will produce a compile time error or warning if it is seen by the code generator. This is useful for doing assertions which the preprocessor cannot handle, e.g., to ensure that the size of two structures are the same. If the **__ERROR__( )** function is placed after an **if** statement which will not be executed unless the assertion fails, the optimizer will remove the **__ERROR__( )** function, and no error will be generated.

The syntax of the **__ERROR__( )** function:

__ERROR__(*error-string* [, *value*])

where *error-string* is the error message to be generated and the optional *value* defines whether the error should be:

0	A warning - compilation will continue.
1	An error - compilation will continue but will stop after the entire file has been processed.
2	A fatal error - compilation is aborted.

If no value is given, the default value of 1 is used.

Example:

```
extern void __ERROR__(char *, ...);

#define CASSERT(tst) \
        if (!(tst)) __ERROR__("C assertion failed: " #tst)
.
.
.
CASSERT(sizeof(struct a) == sizeof(struct b));
```

extended

If the option -Xkeywords=*x* is used with bit 0 set in *x* (e.g., -Xkeywords=0x1), D-C++ recognizes the keyword **extended** as a synonym for `long double`.

Example:

```
extended e;         /* the same as long double e; */
```

ident

The **#ident** preprocessor directive inserts a comment into the generated object file. The syntax used is

#ident *string*

Example:

```
#ident "version 1.2"
```

The text string is forwarded to the assembler in an **ident** pseudo-operator and the assembler outputs the text in the **.comment** or section (for COFF and ELF formats; a similar section is used for other formats).

#import

The **#import** preprocessor directive is equivalent to the **#include** directive, except that if a file has already been included, it is not included again. The same effect can be achieved by wrapping all include files with protective **#ifdefs**, but using **#import** is much more efficient because the compiler does not have to open the file. Using the -Ximport command line option will cause all **#include** directives to behave like **#import**.

inline and __inline__

The **__inline__** keyword provides a way to replace a function call with an inlined copy of the function body. **__inline__** works like the **inline** keyword in C++ and like the **#pragma inline** directive. The keyword **inline** can also be used in C if the option -Xkeywords=0x4 is given (see page 24).

Example:

```
__inline__ void inc(int *p) {
  *p = *p+1;
}

  inc(&x);
```

The function call will be replaced with

```
  x = x+1;
```

interrupt and __interrupt__

The `__interrupt__` keyword provides a way to define a function as an interrupt function. The difference between an interrupt function and a normal function is that all registers are saved, not just the those which are volatile, and a special return instruction is used. `__interrupt__` works like the `#pragma interrupt` directive. The keyword `interrupt` can also be used in C if the option -Xkeywords=0x10 is given (see page 24).

Example:

```
__interrupt__ void trap() {
  /* this is an interrupt function */
}
```

packed (*max*, *min*, *byte-swapped*)
__packed__(*max*, *min*, *byte-swapped*)

The `__packed__` keyword defines how a structure should be padded between members and at the end. The keyword `packed` can also be used if the option -Xkeywords=0x8 is given. See **#pragma pack** on page 47 for treatment of 0 values and defaults.

The *max* value specifies the maximum alignment of any member in the structure. If the natural alignment of a member is less than *max*, the natural alignment is used. See the *Internal Data Representation* chapter in the *Compiler Target User's Manual* for more information about alignments and padding.

The *min* value specifies the minimum alignment of the structure. If any member has a greater alignment, the highest value is used.

The *max* and *min* values can be set by using the -Xstruct-max-value and the -Xstruct-min-value options.

The *byte-swapped* option enables swapping of bytes in structure members as they are accessed. If 0 or absent, bytes are taken as is; if 1, bytes are swapped as they are transferred between byte-swapped structure members and registers or non-byte-swapped memory.

See **#pragma pack** on page 47 for more information and additional examples.

Examples:

```
__packed__ struct s1 {          No padding between members
  char c;
  int  i;                       starts at offset 1
};                              total size is 5

__packed__(2,2) struct s2 {     Maximum alignment is 2
  char c;
  int  i;                       starts at offset 2
};                              total size is 6

__packed__(4) struct s3 {       Maximum alignment is 4
  char c;
  int  i;                       starts at offset 4
```

```
};                                      total size is 8

__packed__(4,2) struct s4 {             Minimum alignment is 2
  char c;
};                                      total size is 2
```

pascal

If the option -Xkeywords=*x* is used with bit 1 set in *x* (e.g., -Xkeywords=0x2), D-C++ recognizes the keyword **pascal**. This keyword is a type modifier that affects functions in the following way:

- The argument list is reversed and the first argument is pushed first.
- On CISC Processors (e.g., MC68000), the called function clears the argument stack space instead of the caller.

sizeof

The `sizeof` keyword has been extended to incorporate the following syntax:

sizeof (*type*, *int-const*)

where *int-const* is an integer constant between 0 and 2 with the following semantics:

0 standard **sizeof**, returns size of type

1 returns alignment of type

2 returns an **int** constant depending on type as follows:

signed char	0
unsigned char	1
signed short	2
unsigned short	3
signed int	4
unsigned int	5
signed long	6
unsigned long	7
long long	8
unsigned long long	9
float	14
double	15
long double	16
void	17
any pointer	19
array	22
struct	23
union	24
function	25
class	32
reference	33
enum	34
plain **char**	44

Examples:

```
i = sizeof(long ,2)       /* typeof(long):   i = 6*/
j = sizeof(short,1)       /* alignof(short): j = 2 */
```

6 Optimization Hints

D-C++ is a globally optimizing compiler and will attempt to produce code as compact and efficient as possible. However, there is information about characteristics of the program that only the user has.

This section describes various ways the user can help the compiler generate the most optimal code.

What to do from the command line

The usual purpose of optimizations is to make a program run as fast as possible. Most optimizations also make the program smaller; however the following optimizations will increase program size, exchanging space for speed:

- *Inlining*: replaces a function call with its actual code.
- *Loop unrolling*: expands a loop with several copies of the loop body.

When a program expands it may have a negative effect on speed due to increased cache-miss rate and extra paging in systems with virtual memory.

Because the compiler does not have enough information to balance these concerns, two options are provided to let the user control the above mentioned optimizations:

- -Xinline=*n*

  Controls the maximum size of functions to be considered for inlining. *n* is the number of internal nodes. See -Xinline on page 24 for more details and -Xunroll-size on page page 30 for a definition of internal nodes.

- -Xunroll-size=*n*

  Controls the maximum size of a loop body to be unrolled. See -Xunroll-size on page 30 for more details.

There is another trade-off between optimization and compilation speed. More optimization requires more compile time. The amount of main memory is also a factor. In order to execute inter-procedural optimizations (optimizations across functions) the compiler keeps internal structures of every function in main memory. This can slow compilation considerably if not enough physical memory is available and the process has to swap pages to disk. The -Xparse-size=*m* option, where *m* is memory space in KByte, is set to suggest to the compiler how much memory it should use for this optimization. (See page 28.)

With all the different optimization options it is sometimes difficult to decide which options will give the best result. The -Xblock-count and -Xfeedback options, which produce and use profiling information, provide a powerful mechanism to help with this. With profiling information available, the compiler can make most optimization decisions by itself. See the -Xfeedback option on page 23 for details.

The following guidelines summarize which optimizations to use in varying situations. The options used are found in the several tables in Chapter 3.

- If execution speed is not important, but compilation speed is crucial (for example while developing the program) do not use any optimizations at all:

```
dplus file.cpp -o file
```

- The -O option is a good compromise between compilation time and execution speed:

```
dplus -O file.cpp -o file
```

- To produce highly optimized code, without using the profiling feature, use the -XO option:

```
dplus -XO file.cpp -o file
```

- To obtain the fastest code possible, first compile the program with the -Xblock-count option, then execute the program to create a file containing the profiling information. Next recompile the program with the -XO -Xfeedback options (see Chapter 8, "Use in an Embedded Environment," in the *Compiler Target User's Manual* for file I/O in an embedded environment):

```
dplus -Xblock-count file.cpp -o file
execute file
dplus -XO -Xfeedback file.cpp -o file
```

- To produce the most compact code, use the -Xsize-opt option:

```
dplus -XO -Xsize-opt file.cpp -o file
```

- If the compiler complains about "end of memory" (usually only on systems without virtual memory) try to recompile without using -O.
- When compiling large files on a host system with large memory, increase the amount of memory D-C++ can use to retain functions. This allows D-C++ to perform more inter-procedural optimizations. Use the following option to increase the available memory to 4,000 KByte:

```
-Xparse-size=4000
```

- If speed is very important and the resulting code is small compared to the cache size of the target system, increase the values controlling inlining and loop-unrolling:

```
-XO -Xinline=80 -Xunroll-size=80
```

- When it is difficult to change scripts and makefiles to add an option, set the environment variable `DFLAGS`. Examples:

UNIX:
```
DFLAGS="-XO -Xparse-size=4000 -Xinline=50"
export DFLAGS
```

DOS:
```
set DFLAGS=-XO -Xparse-size=4000 -Xinline=50
```

MPW:
```
Set DFLAGS "-XO -Xparse-size=4000 -Xinline=50"
Export DFLAGS
```

What to do with programs

The following list describes things to consider when writing C++ programs which will help the compiler to produce optimized code.

- Use local variables. D-C++ can keep these variables in registers for longer periods than global and static variables, since it can trace all possible uses of the variable.
- Avoid taking the address of variables. When the address of a variable is taken, D-C++ usually assumes that the variable is modified whenever a function is called or a value is stored through a pointer. Use function return values instead of passing addresses.
- Let the compiler perform subscript optimizations. For example:

```
for(i = 0; i < n; i++)
   d[i] = s[i];
```

  lets the compiler, with knowledge about the target, generate the optimal code while:

```
dp = d; sp = s;
while(n--)
   *dp++ = *sp++;
```

  is effective on certain architectures, but not on others.
- Use the `#pragma inline` directive for small frequently used functions.
- Use plain `int` variables when size does not matter. Local variables of shorter types must often be sign-extended on specific architectures before compares, etc.
- Use the `unsigned` keyword for variables known to be positive.
- Avoid using the -g option. The -g option turns off many optimizations in order to generate more complete debug information.
- Use the `static` keyword on functions and external variables that are not used by any other file. The optimizer can be much more clever if it knows that no other module is using the variable/function.
- Use the `#pragma no_alias` directive to inform the compiler about aliases in time critical loops.
- Avoid **setjmp**( ) and **longjmp**( ). When D-C++ finds **setjmp**( ) in a function, a number of optimizations are turned off. For example, when the -Xpcc option is specified, no variables declared without the **register** keyword will be allocated to registers. This is done to be compatible with older compilers that always allocate variables not declared **register** to the stack, which means that if they are changed between the call to **setjmp**( ) and the call to **longjmp**( ), they will keep the changed value after the **longjmp**( ). If the variables were allocated to registers, they would have the values valid at the time of the **setjmp**( ). The following example demonstrates this difference:

```
#include <setjmp.h>
static jmp_buf label;

f1()
```

```
{
  int i = 0;

  if (setjmp(label) != 0) {
    /* returned from a longjmp() */
    if (i == 0) {
      printf("i has first value: allocated to register.\n");
    } else {
      printf("i has new value: allocated on stack\n");
    }
  return;
  }

  /* setjmp() returned 0: does not come from a longjmp */
  i = 1;
  f2();
}

f2()
{
  /* jump to the setjmp call, returning 1 */
  longjmp(label, 1);
}
```

Note that both ways are valid according to ANSI.

7 The Lint Facility

The lint facility is a powerful tool to find common programming mistakes at compile time. Lint has the following features:

- It is activated through command line option -Xlint.
- -Xlint does all checking while compiling. Since it does not interfere with optimizations, it can always be enabled.
- -Xlint gives warnings when a suspicious construct is encountered. To stop the compilation (so that warnings do not scroll off the screen) use the -Xstop-on-warning option that will treat all warnings like errors.
- Each individual check that -Xlint performs can be turned off by using a bit-mask. See the -Xlint option on page 26 for details.
- For C programs, -Xlint can be complemented with the -Xforce-prototypes option to warn of a function used before its prototype.
- The comments in the following C program demonstrate probable defects that will be detected by using -Xlint and -Xforce-prototypes.

```
void f1(int);
void f2();

static int f4(int i) /* function never used */
{
  if (i == 0)
    return; /* missing return expression */
  return i+4;
}

static int f5(int i); /* function not defined */

static int i1; /* variable never used */

int m(char j, int z1 /* parameter never used */ )
{
  int i;
  unsigned u = 1; /* variable set but not used */
  int z2; /* variable never used */
  if (j) {
    u = 4294967295;
    i = 0;
  } else {
    u = 4294967296; /* constant out of range */
  }
  f1(i); /* variable might be used before set */
  switch(i) {
    j = 2; /* statement not reached */
    break;
  case 0:
    f2(i); /* function has no prototype */
```

```
    f3(i); /* function not declared */
    f5(i);
    break;
  case 4*0x7fffffff:
    /* overflow in constant expr */
    j = 1; /* variable set but not used */
    return 1;
  case 1:
  deflaut: /* label not used (misspelled!) */
    j = 1000; /* constant out of range */
    return j;
    j = 5; /* statement not reached */
    j++;
  }
  if (j < 0 && j > 10) {
    /* condition always true/false  */
    f1(j);
  }
  if (j > 0 || j < 10) {
    /* condition always true/false  */
    f1(j);
  }
  if (u == -10) f1(j); /* constant out of range */
} /* missing return expression */\
```

8 Converting Existing Code to D-CC

To use D-C++ for code developed on a different system and/or compiler is usually straight forward, especially given the large number of compatibility options supported by D-C++. This chapter gives some pointers on how to get around the most common differences between systems and compilers.

Compilation problems

The following list include hints on what to do when a program fails to compile and you want to avoid changing the source.

Look for missing standard include files

Different systems have different standard include files and the declarations within the include files may be different. Use the -i option to change the name of a missing include file. See the User's Manual for details.

Look for code using lose typing control

Some older code is written for compilers that does not check the types of identifiers thoroughly. Use the -Xmismatch-warning=2 option if you get error messages like "illegal types: ...".

Look for code written for PCC

Code written for older UNIX compilers, for example, PCC (Portable C Compiler) may not be compatible with C standard. Use the -Xpcc option to enable some older language constructs. See "Compatibility modes: ANSI, PCC, and K&R C" on page 83 for more information.

Execution problems

The following list include hints on what to do when a program fails to execute properly.

Compile with -Xlint

The -Xlint option enables compile-time checking that will detect many non-portable and suspicious programming mistakes. See Chapter 7, "The Lint Facility," beginning on page 63.

Recompile without -O

If a program executes correctly when compiling without optimizations it does not necessarily mean something is wrong with the optimizer. Possible causes include:

- Use of memory references mapped to external hardware. Add the `volatile` keyword or compile using the -Xmemory-is-volatile option.
- Use of uninitialized variables are exposed by the optimizer.
- Use of expressions with undefined order of evaluation.

Uninitialized local variables will behave differently on dissimilar systems, depending how memory is initialized by the system. D-C++ generates a warning in many instances, but in certain cases it is impossible to detect these discrepancies at compile time.

Look for code allocating dynamic memory in invalid ways

The following invalid uses of `operator new( )` or `malloc( )` may go undetected on some systems:

- Assuming the allocated area is initialized with zeroes.
- Writing past the end of the allocated area.
- Freeing the same allocated area more than once.

Look for expressions with undefined order of execution

The evaluation order in expressions like `x + inc(&x)` is not well defined. Compilers may choose to call `inc(&x)` before or after evaluating the first `x`.

Look for NULL pointer dereferences

On some machines the expression `if (*p)` will work even if `p` is the zero pointer. Replace these expressions with a statement like `if (p != NULL && *p)`.

Look for code which makes assumptions about implementation specific issues

Some programs make assumptions about the following implementation specific details:

- Alignment. Look for code like:

  ```
  char *cp; double d; *(double *)cp = d;
  ```

- Size of data types.
- Byte ordering. See **#pragma pack** on page 55 on methods for accessing byte-swapped data.
- Floating point format.
- Sign of plain `chars`. The type `char` is by default treated as signed in D-C++. Use the option -Xunsigned-char to be compatible with systems that treat plain `chars` as unsigned.
- Sign of plain `ints` in bitfields. `ints` in bitfields are unsigned by default in D-C++. Use the option -Xsigned-bitfields to be compatible with systems that treat plain `ints` in bit-fields as signed.

Functions with variable number of arguments

Two include files, `stdarg.h` (and `varargs.h`), are provided to implement functions with a variable number of arguments. The following example shows the recommended way to use `stdarg.h`:

```
#include <stdarg.h>
/* prints pairs of ints and doubles */

void fn(int no_of_pairs, ...)
{
  va_list ap;

  int     i;
  int     arg_i;
  double  arg_f;
```

```
  va_start(ap, no_of_pairs);

  for(i = 0; i < no_of_pairs; i++) {
    arg_i = va_arg(ap, int);

    arg_f = va_arg(ap, double);
    printf("pair #%d: int: %d, double: %f\n",i,arg_i,arg_f);
  }

  va_end(ap);
}


fn(3, 12, 45.5, 14, 47.8, 17, -2);
```

The following methods used to implement variable arguments are non-portable and will only work on specific architectures:

```
/*  Inadequate way to implement variable arguments #1:
    by specifying "enough" int parameters.
    Problems: arguments of other types may be passed in
    different registers and/or be aligned differently
*/

fn1(i1,i2,i3,i4,i5,i6,i7,i8,i9,i10)
{
  printf("warning: ");
  printf(i1,i2,i3,i4,i5,i6,i7,i8,i9,i10);
}

/* Better implementation with stdarg.h and vprintf() */


fn1(char *format, ...)
{
  va_list ap;

  va_start(ap, format);
  printf("warning: ");
  vprintf(format, ap);

  va_end(ap);
}

/*  Inadequate way to implement variable arguments #2:
    by taking the address of the first argument and hoping
    that the rest will follow consecutively.
    Problems: arguments may be passed in registers and not
    on the stack, and/or aligned differently
*/

fn2(i1) {
  int  *p = &i1;
```

```
  int  i  = 0;

  while(*p1 != 0) printf("arg #%d: %d\n",i++,*p++);
}

/* Better implementation with stdarg.h */

fn2(int i1, ...)
{
  va_list  ap;

  int      i   = 0;
  int      arg = i1;

  va_start(ap, i1)

  arg = va_arg (ap, int);
  while (arg != 0) {
    printf("arg #%d: %d\n",i++,arg);
    arg = va_arg(ap, int);
  }

  va_end(ap);
}
```

➤ **Optimization note:** On architectures passing arguments in registers, usage of the **stdarg** or **varargs** macros will create extra entry code in a function.

9 C++ Features and Compatibility

This section describes C++ as implemented by D-C++. There may be some differences from the evolving C++ standard.

C++ features

C++ is an extension to the C language. A short list of the main improvements is:

- Stronger type checking.
- User defined types called **class**es.
- Public, protected and private class members.
- Constructors and destructors for classes.
- Static class members, common to all object instances.
- Virtual class member functions.
- Multiple and virtual class inheritance.
- Abstract classes.
- Function and operator overloading.
- Operators **new** and **delete** for memory allocation with construction / destruction.
- Default argument values in function prototypes.
- Static object initializer expressions can include previously defined non-constant objects.
- Declarations of variables can be anywhere within a block of statements.
- Templates to create sets of functions and classes.
- Exception handling to take care of errors or other conditions detected by the program.
- A new input/output library (**iostream**) with type sensitive operators.

Header files

Generally C++ uses the same header files as C, but the declarations need to be adjusted to work in both C and C++ environments. See the next subsection.

The following header files are specific to C++:

Table 9-1 Additional Header Files for C++

Header File	Description
`new.h`	declares the functions **new_handler( )** and **set_new_handler( )** in case the user wants to overload the built-in operator **new**.
`except.h`	declares the functions **terminate( )**, **set_terminate( )**, **unexpected( )** and **set_unexpected( )** in case the user wants to overload any of these exception handling functions. The **unexpected( )** function is automatically called if an exception is thrown for which no **catch** block is defined. The **terminate( )** function is automatically called by the built-in **unexpected( )** function and terminates the program. See "Exceptions" on page 75 below for more details.

In addition, the C++ standard imposes additional requirements on other standard C headers files. See documents referred to in "Additional documentation" on page 3 for details.

Migration from C to C++

When converting a set of C functions to C++ or when a set of C functions are to be called from a C++ program, a few minor differences between C and C++ must be considered and the header files must be written in C++ style.

The standard predefined macro **___cplusplus** can be used with **#ifdef** directives in the program and the header files if the code will be used in both C and a C++ programs.

To call a C function from a C++ program, the function prototype must be declared with **extern "C"** to avoid name mangling and the function arguments must be declared in C++ compatible format.

A few general differences between C and C++ are listed below, but refer to *The Annotated C++ Reference Manual* or the ANSI C++ draft for details.

- A function declared `func( )` has no argument in C++, but has any number of arguments in C. Use the **void** keyword for compatibility, e.g. `func(void)`, to indicate a function with no arguments.
- A character constant in C++ has the size of a **char** but in C has the size of an **int**.
- In C, an **enum** has always the size of an **int**, but can have another size in C++.
- The name scope of a **struct** or **typedef** differs slightly between C and C++.
- The additional keywords in C++ could make it necessary to modify a C program if these keywords are user-defined identifiers, e.g. **asm**, **catch**, **class**, **delete**, **friend**, **inline**, **new**, **operator**, **private**, **protected**, **public**, **template**, **throw**, **try**, **this**, **virtual**.
- In C, a global **const** by default has external linkage. In C++ it does not, thus requiring explicit use of **static** or **extern** to define which is intended.

The D-CLASS C++ Library

The **Diab Data C++ library**, called D-CLASS, is described in the *Class Library Reference Manual.*

It contains the **iostream** input/output class library and libraries for a **complex** class.

Special C++ features

In this subsection a few special features of C++ are treated. They are mainly newer C++ features for which there otherwise could be some uncertainty about their implementation.

Construction / destruction of C++ static objects

Before the first statement of the *main( )* function in a C++ program can be executed, all global and static variables must be constructed. Also, before the program terminates, all global and static objects must be destructed.

These special constuctor and destructor operations are carried out by code in the .init and .fini sections as described under "Global initialization" in Chapter 8, "Use in an Embedded Environment," in the *Compiler Target User's Manual.*

Alternative method using -Xcall-Main option

With an older method, the compiler adds a call to the C++ library function **_MAIN( )** before any other statements in **main( )**. This function:

- Calls each function in the null-terminated **array-of-pointers-to-functions _CTORS**.
- Registers, using **atexit( )**, each function in the null-terminated **array-of-pointers-to-functions _DTORS** to be executed at program termination.

The two arrays are usually created by the linker, from the compiler generated symbols with the prefixes **_STI__** and **_STD__** respectively.

This method is obolete and should not be used with new code. For existing code, the -Xcall-MAIN option can be used to cause this older behavior.

Templates

Function templates and class templates are implemented per the draft standard. See it for details, which are not given here. A short introduction follows to provide background for a description of how D-C++ instantiates templates.

Function templates

A function template is a means for defining a pattern for a set of overloaded functions, from which the compiler generates (instantiates) actual overloaded functions, based on the template argument types.

The template argument types can be of any type, not only user defined classes, even though the **class** keyword is used in the definition. However, restrictions may be imposed on the template argument types, depending on how they are used in the code in the template definition. Explicit template function definitions can be provided to override the automatic instantiation for a given type.

Example:

This is an example of a function template definition. `T` is a template argument type. All template arguments must be used in the function parameter declarations and they are used in the function definition. The types will be replaced by the actual types when the compiler instantiates a specific function, i.e. when it finds a call to a function with argu-

ments corresponding to the template function declaration. An exact match of the types is required.

```
#include <iostream.h>

template<class T> // Declare template function
T max(T a, T b) { return a > b ? a : b; }

main()
{
  double d1 = 123.456, d2 = 654.321;

  cout << max(d1, d2)   << endl; // max(double, double)
  cout << max(4711, 17) << endl; // max(int, int)
}
```

Class templates

A class template is a means for defining a pattern for a set of similar user defined classes, from which the compiler generates the actual class definitions based on the template argument types.

The template argument types can be any type, not only user defined classes, even though the **class** keyword is used in the definition. However, restrictions may be imposed on the template argument types, depending on how they are used in the code in the template definition. Explicit template class definitions can be provided to override the automatic class instantiation for a given type.

Example:

A class template is defined, depending on template arguments types, e.g. in this case the classes `T1` and `T2`. The template arguments are used in the class definition and will be replaced by the actual types when the compiler instantiates a specific class.

```
template<class T1, class T2> class table
{
   // definition using T1 and T2 as types
}
```

Instantiation of a template class is performed by the compiler (as for any user defined class) when a class object is declared, explicitly or implicitly. An instantiation defines a new user defined class with a class name, containing the template arguments.

The following is a more complete example with two templates:

```
#include <iostream.h>

template <class T> // Declare a template struct
struct listitem {
  T elem;
  listitem<T> *next;
  listitem(T e) : elem(e), next(NULL) {}
};

template <class T> // Declare a template class
class list {
```

```
  listitem<T> *l;
  listitem<T> **lp;
public:
  void add(const T &);
  void print(void);
  list() : l(NULL) , lp(&l) {}
};

template <class T>
void list<T>::add(const T &arg)
{
  *lp = new listitem<T>(arg);
  lp = &((*lp)->next);
}

template <class T>
void list<T>::print(void)
{
  for (listitem<T> *p = l; p != NULL ; p = p->next) {
    cout << p->elem << endl;
  }
}

main()
{
  list<int> il;
  list<char> cl;
  list<double> dl;

  il.add(3); il.add(5); il.add(7);
  cl.add('a'); cl.add('b'); cl.add('c');
  dl.add(12.34); dl.add(23.45); dl.add(34.56);

  cl.print();
  il.print();
  dl.print();
}
```

Template instantiation

C++ templates are the first C++ language feature to require more intelligence from the environment than one usually finds with C development tools. The compiler and linker have to make sure that each template instance occurs exactly once in the executable if it is needed, and not at all otherwise. There are two basic approaches to this problem, which referred to here as the Borland model and the Cfront model.

Borland model

The Borland C++ linker includes the code equivalent of common blocks; template instances are emitted in each translation unit that uses them, and they are collapsed together at run time. The advantage of this model is that the linker need only consider the object files themselves; external complexity is eliminated. The disadvantage is that compilation time is increased because the template code is compiled repeatedly.

Cfront model

The AT&T C++ translator, Cfront, implements a template repository, an automatically maintained place where template instances are stored. As individual object files are built,

notes are placed in the repository to record the location of templates and potential type arguments so that the subsequent instantiation step knows where to find them. At link time, any needed instances are generated and linked in. The advantages of this model are faster compilation speed and the ability to use the system linker; to implement the Borland model a compiler vendor also needs to replace the linker. The disadvantages are increased complexity, and thus potential for error. Theoretically, this scheme should be just as transparent to the programmer at the Borland model, but in practice it has been very difficult to build multiple programs in one directory and one program in multiple directories using Cfront. Code written for this model tends to separate definitions of non-inline member templates into a separate file, which is found by the link preprocessor when a template needs to be instantiated.

Template support in D-C++

Currently, D-C++ implements neither automatic model. There are two options for dealing with template instantiations:

- Do nothing. Code written for the Borland model will work fine, but each translation unit will contain instances of each of the templates it uses. In a large program, this can lead to an unacceptable amount of code duplication.
- Compile with -Xno-implicit-templates and explicitly instantiate all the template instances you use. To explicitly instantiate a template class for a function use the following syntax:

```
template class A<int>;    // Instantiate A<int> and all
                          // member functions.
template int f1(int);     // Instantiate int f1(int).
```

This strategy will work with code written for either model. If you are using code written for the Cfront model, the file containing a class template and the file containing its member templates should be implemented in the same translation unit. This is usually the best alternative; it may require more knowledge of exactly which templates you are using, but it keeps the code size down and requires no modification of existing code. You can create one big file to do all the instantiations, or you can create small files like the following example to *explicitly instantiate* template instances:

```
// Instantiate class and member functions for
// foo<int> and foo<char *>.
#include "foo.h"          // Include interface for foo.
#include "foo.C"          // Include implementation if the
                          // code uses the Cfront model.
template class foo<int>;
template class foo<char *>;
```

Create one such file for each instance you need, and create a template instantiation library from those. If you are using Cfront-model code, you can need not use -Xno-implicit-templates when compiling files that do not `#include` the member tem-

plate definitions. For ease of use, the compiler can be told which templates have to be instantiated:

Table 9-2 Template Instantiation (-Xno-implicit-templates)

-Xno-implicit-templates -X207=1	Implicit instantiations from declarations of or references to template-type objects are **not** done, but explicit instantiations as illustrated just above are done.
-Xno-implicit-templates=2 -X207=2	Same as the above except the compiler reports which instantiations are done as exemplified by the following: `"example.C", line 14: Instantiation: template A<int>;`
-Xno-implicit-templates=3	Also reports instantiations of member functions. These should not be instantiated explicitly; they are instantiated when the class is instantiated.

This scheme will provide the user with the option of filtering the diagnostics beginning with "Instantiation:" to extract the instantiation string and collect them into a file that can be compiled.

Exceptions

C++ exception handling provides a mechanism for handling software generated error and/or other exceptional events. It is implemented according to the draft standard. Refer to it for details not given here. A short introduction follows.

The generation of exception handling code can be disabled using the -Xexception=0 D-C++ option. With this option, the compiler will also flag the keywords **try**, **catch** and **throw** as errors.

Exception handling allows a program which detects an exceptional event to transfer control to an unspecified handler, defined in any calling function on any previous level, while automatically unwinding the stack and calling the destructors for all automatic objects. Other resources can also be restored automatically if the user ensures that the resources are allocated in a suitable automatic object constructor and deallocated in the corresponding destructor. At the transfer of control, information about the cause of the exception is available as an exception object. If no exception handler corresponding to the exception object type or a base class of the type is found, the program is automatically terminated.

Three new keywords: **throw**, **try** and **catch** are used.

Table 9-3 Keywords Related to Exception Handling in C++

throw *eobj* **throw** *etype*( ) **throw**	Generates an exception with an exception object *eobj* or an exception with a temporary exception object of the type *etype*. Within an exception handler **throw** can be used without a parameter to rethrow the active exception object. The **throw** keyword is also used to add an optional exception specification in a function declaration. This specification contains a list of exception types the function is allowed to throw.

Table 9-3 Keywords Related to Exception Handling in C++

try { }	Defines a block in the program, from which an exception can be thrown by a **throw** statement and handled by the exception handlers defined for this block. The handlers are defined by **catch** blocks following immediately after the **try** block. The **throw** statement can be within the **try** block itself or in any function called directly or indirectly from this block.
catch(*etype*) { } **catch**(*etype e*) { } **catch**(...) { }	Defines an exception handler as a block of code with a parameter indicating which exception object type this handler accepts. Any number of **catch** blocks can follow immediately after a **try** block. A handler can be set up to handle any exception type by specifying the ellipsis (...) parameter. Also objects which are derived from the specified exception object are handled.

The default action if no corresponding handler is found is to terminate the program. This action can be redefined by the user. A user defined function **unexpected**( ) can be activated by the function **set_unexpected**( ), and a user defined function **terminate**( ) can be activated by the function **set_terminate**( ). These return pointers to the previously defined action functions which makes it possible to build a stack of action functions. To call these functions, the header file `except.h` must be included.

Table 9-4 User Definable Functions for Unexpected Exceptions in C++

unexpected( )	Called if a function with an exception specification list throws an exception which is not specified in the list. It can also be called from a handler after doing some user defined cleanup. The default action of **unexpected**( ) is to call **terminate**( ).
terminate( )	Terminate the program as a last resort if no handler is found for the exception, if the stack if corrupted or when a destructor during the exception generated stack unwinding calls an exception. It is also called by the default **unexpected**( ) function.

Example:

The following example demonstrates a program structure, using exceptions.

```
#include <iostream.h>

class except { // Declare an exception class
public:
  char * a;
  except(char *b="") : a(b) {}
};
const int bad=1, wrong=2, verywrong=3;

void testf(int something) throw (int, except)
{
  if (something == verywrong) throw 2;
```

```
  if (something == wrong) throw except("wrong");
// .....
}

void testing(int test)
{
  try {
    if (test == bad) throw except("bad");
    testf(test);
  }

  catch(except &e) {// Catch exceptions of type except
    cout << "Exception: " << e.a << ", Go on!" << endl;
  }

  catch(...) {// Catch any exception type
    cout << "Unknown exception type. Stop!" << endl;
    throw;// Rethrow to terminate
  }

  cout << "Continuing after try block" << endl;
}

// -------------------
main()
{
  testing(0); testing(1); testing(2); testing(3); return(0);
}
```

Array new and delete

The two memory allocation/deallocation operators `operator` **new[ ]( )** and **operator delete[ ]( )** are implemented as defined in the ANSI C++ draft standard.

Type identification

The **typeid** expression returns an expression of type **typeinfo&**. The **type_info** class definition can be found in the header file `typeinfo.h`.

Dynamic casts in C++

Dynamic casts are made with **dynamic_cast**(*expression*) as described in the C++ ANSI draft.

C++ name mangling

The compiler encodes every function name in a C++ program with information about the types of its arguments to 1) enable overloading, and 2) improve the ability of the linker to detect errors. This encoding process is called *name mangling*.

When linking C++ and C code together, functions called as C functions must be declared with the following linkage specification to tell the C++ compiler that those function names are not mangled:

```
extern "C"
```

The scheme used for encoding function names follows the suggestions in *The Annotated C++ Reference Manual* by Bjarne Stroustrup, with two underline characters between the

name and the encoded information. The user should avoid using double underline within user function names.

A function name is encoded (mangled) with the types of all arguments. A member function has also the class name encoded in the name. The names of classes (user defined names) are encoded as the length of the name followed by the name itself. Nested class names will contain the name of all classes in the hierarchy, using the **Q** modifier (see below). Most other type indicators are single characters.

Functions differing only in return type are considered the same and can not be overloaded.

For a constructor, destructor or operator, a special string with two double underlines is prepended to the class name (used as function name). A table with operator names follows at the end of this chapter. These special strings are:

__ct__	Constructor
__dt__	Destructor
__xx__	Operator with abbreviation *xx* (e.g. **__pl__**)

A global function has a double underline appended to the name, followed by the global function indicator **F** and the types of the arguments.

A class member function has the class name inserted before the **F** indicator.

Argument types are encoded as follows. Intrinsic types are encoded with one character, while user defined types are encoded as the length of the type name in decimal plus the type name. **???F: are m, n, n1, and n2 single-digit unless?, or is it NmBn, or ???**

Table 9-5 Type Encodings for Name Mangling in C++

Code[a]	Type
A *n*_	Array (followed by the simple type name), where *n* is the array size
B	Used to separate two other encoding elements that would otherwise appear to be one.
d	**double**
c	**char**
e	Ellipses parameter (...)
F	Global function
f	**float**
i	**int**
l	**long**
M	Member pointer (followed by a simple type name)
N *m n*	Repeating *m* arguments with the same type as argument number *n*

Table 9-5 Type Encodings for Name Mangling in C++ *(continued)*

Code[a]	Type
n name	User defined type, with *n* giving the length of *name* and *name* giving the type name.
P	Pointer (followed by the simple type name)
Q *m n1 name1 n2 name2...*	Nested class name (*m* user defined type names after Qm) (See examples below.)
R	Reference (followed by the simple type name)
r	**long double**
s	**short**
T *n*	Same type as argument number *n*
v	**void**

a. Embedded spaces in some codes are for readability only and do not appear in the actual encodings.

Modifiers are inserted before the type indicator, and are encoded as follows. If more than one modifier is used, they are inserted in alphabetical order.

Table 9-6 Modifiers for Type Encodings

C	**const** type
S	**signed** type
U	**unsigned** type
V	**volatile** type

The following table shows examples of names mangled using the rules above.

Table 9-7 Examples of C++ Name Mangling

Function Header	Mangled Name
`myfunc()`	`_myfunc__Fv`
`myclass::init()`	`_init__7myclassFv`
`myfunc(int, char)`	`_myfunc__Fic`
`myfunc(complex, int&, complex)`	`_myfunc__F7complexRiT1`
`typedef int myarr[6][2];` `void myfunc(myarr)`	`_myfunc__FA6_A2_i`
`myfunc(char&, signed int*)`	`_myfunc__FRcPSi`

Names used for operator encoding

Operators are treated as functions with special encoding used as names during the name mangling.

__xx__ Operator with abbreviation *xx* (e.g. **__pl__** for the + operator)

The *xx* codes used are:

```
aa        operator &&  ( )
aad       operator &=  ( )
ad        operator &   ( )
adv       operator /=  ( )
aer       operator ^=  ( )
als       operator <<= ( )
amd       operator %=  ( )
ami       operator -=  ( )
amu       operator *=  ( )
aor       operator |=  ( )
apl       operator +=  ( )
ars       operator >>= ( )
as        operator =   ( )
cm        operator ,   ( )
co        operator ~   ( )
dl        operator delete( )
dv        operator /   ( )
eq        operator ==  ( )
er        operator ^   ( )
ge        operator >=  ( )
gt        operator >   ( )
le        operator <=  ( )
ls        operator <<  ( )
lt        operator <   ( )
md        operator %   ( )
mi        operator -   ( )
ml        operator *   ( )
mm        operator --  ( )
ne        operator !=  ( )
nt        operator !   ( )
nw        operator new( )
oo        operator ||  ( )
or        operator |   ( )
pl        operator +   ( )
pp        operator ++  ( )
rm        operator ->* ( )
rf        operator ->  ( )
rs        operator >>  ( )
```

Unary operators have no argument and binary operators have one argument in their mangled names. For example:

```
class aclass {
public:
  operator -();              // __mi__6aclassFv
  operator -(aclass&);       // __mi__6aclassFR6aclass
}
```

Avoid setjmp and longjmp

It is difficult to safely use **setjmp( )** and **longjmp( )** in C++ code because jumps out of a block may miss calls to destructors and jumps into a block may miss calls to constructors.

Note that in addition to visible user-defined objects, the compiler may have created temporary objects not visible in the source for use in optimized code.

Consider instead C++ exception handling in situations which might have used **setjmp** and **longjmp**. As usual, it will still be necessary account for allocations and deallocations not performed through contructors and destructors of automatic objects.

A Compatibility modes: ANSI, PCC, and K&R C

➤ This section relates to C, not C++. It is included for C++ users who may be working with mixed C/C++ programs.

Many existing C programs are coded in accordance with slightly varying C standards. To ease porting of these programs, D-C++ can run in four different modes as selected by an option from the following table:

Table A-1 Compatibility Mode Options for C Programs

Mode	Option	Meaning
ANSI	-Xansi	D-C++ conforms to ANSI X3.159-1989 with some additions as shown in the table below.
Strict ANSI	-Xstrict-ansi	D-C++ conforms strictly to the ANSI X3.159-1989 standard.
K & R	-Xk-and-r	D-C++ conforms to the pre-ANSI "standard" defined in *The C Programming Language* by Kernighan and Ritchie, with most ANSI extensions activated.
PCC	-Xpcc	D-C++ emulates the behavior of System V.3 UNIX compilers.

The following table describes the differences among these modes. If not otherwise noted, "y" means "yes" and "n" means "no".

Table A-2 Features of Compatibility Modes for C Programs

Functionality	K&R	ANSI	Strict	PCC
long float is same as `double`	y	n	n	y
The **asm** keyword is defined	y	y	n	y
The **volatile**, **const**, and **signed** keywords are defined	y	y	y	n
The type of a hexadecimal constant >= 0x80000000 is **unsigned int** (u) or **int** (i)	i	u	u	i
In ANSI it is legal to initialize automatic arrays, structures, and unions. D-C++ always accepts this and is either silent (s) or gives a warning (w)	s	s	s	w

Table A-2 Features of Compatibility Modes for C Programs *(continued)*

Functionality	K&R	ANSI	Strict	PCC
When two integers are mixed in an expression they cause conversions and the result type is either "unsigned wins" (u) or "smallest possible wins" (s) Example: `((unsigned char)1 > -1)` which is 0 if (u) and 1 if (s).	u	s	s	u
When prototypes are used and the arguments do not match, generate an error (e) or warning (w)	w	e	e	w
Float expressions are computed in **float** (f) or **double** (d)	f	f	f	d
When an array is declared without a dimension in an invalid context, give an error (e) or warning (w)	e	e	e	w
When an array is declared with a zero dimension, give a warning	n	n	y	n
Incompletely braced structure and array initializers can either be parsed top-down (t) or bottom-up (b). Same as the -Xbottom-up-init option	t	t	t	b
When pointers and integers are mismatched, give an error (e) or a warning (w). Same as the -Xmismatch-warning	e	e	e	w
Trigraphs, e.g. '??' sequences, are recognized	y	y	y	n
Illegal structure references give either an error (e) or a warning (w). If more than one defined structure contains a member, an error is always given: `int *p; p->a = 1;`	e	e	e	w
Comments are replaced by nothing (n) or a space (s)	n	s	s	n
Macro arguments are replaced in strings and character constants. Example: `#define x(a) if (a) printf("a\n");`	y	n	n	y
A missing parameter name after a # in a macro declaration generates an error	n	n	y	n
Characters after an **#endif** directive will generate a warning	n	n	y	n
Preprocessor errors are either errors (e) or warnings (w)	e	e	e	w
__STDC__ macro is predefined to (0), (1) or is not predefined (n)	n	0	1	n
__STDC__ macro can be undefined with `#undef`	y	y	n	y
__STRICT_ANSI__ macro is predefined	n	n	y	n

Table A-2 Features of Compatibility Modes for C Programs *(continued)*

Functionality	K&R	ANSI	Strict	PCC
Spaces are legal before cpp #-directives	n	y	y	n
Parameters redeclared in the outer most level of a function will give an error (e) or warning (w)	w	e	e	w
If the function **setjmp( )** is used in a function, variables without the **register** attribute will be forced to the stack (s) or can be allocated to registers (r)	r	r	r	s
C++ comments "//" are recognized in C files	y	y	n	y
Predefined macros without leading underscores, e.g., **unix,** are available	y	y	n	y
The following construct is legal: `f(i) typedef int i4; i4 i; {}`	n	n	n	y

B Compiler Limits

- Compiler limits usually relate to the size of internal data structures. Most internal data structures in D-C++ are dynamically allocated and limited only by the total available memory. There are, however, some internal stacks that can overflow. The size of these stacks are set higher than that required by the minimum translation limits defined in any standards(e.g., as defined in section 2.2.4.1 in ANSI X3.159-1989 of the C standard.
- Limitations related to memory size depend on how much memory D-C++ will try to claim. These include:
    - The size of the largest function in the file. The size is measured in number of expression nodes, where each operand and operator generates one node. After code generation, the memory used by a function can be reused.
    - Optimization level. Some optimizations use a large amount of memory. Reaching analysis uses memory proportional to the number of basic blocks multiplied by the number of variables used in the function. If there is not enough memory to perform this analysis, a warning will be output, and code generation will continue without some optimizations.
    - Large initialized arrays.
    - The number of KBytes D-C++ is allowed to use to delay code generation in order to perform inter-procedural optimizations. The default value with -O is 500, and can be changed with the -Xparse-size option (see page 28).
- Include files may be nested to a depth of 20.

C Implementation-Defined Behavior

The ANSI C standard X3.159-1989 leaves certain aspects of a C implementation to the implementor. This appendix describes how Diab Data has implemented these details in D-C++. Note that there are differences between C and C++. This section addresses C only.

➤ This chapter contains material applicable to execution environments supporting file I/O and other operating system functions. Much of it therefore depends on the operating system present, if any, and may not be relevant in an embedded environment.

Translation

Four types of diagnostics are generated:

`Info`	Compilation continues.
`Warning`	Compilation continues.
`Error`	Compilation continues until end of file, then aborts.
`Fatal`	Compilation aborts immediately.

Warning diagnostics have the following format on UNIX and DOS systems, where *message* is one of the error messages listed in a separate appendix.

```
"filename", line #: warning: message
```

Error and fatal diagnostics have the following format:

```
"filename", line #: message
```

Example:

```
"file.c", line 2: identifier i not declared
```

Warning diagnostics have the following format on MPW systems, where 'message' is one of the error messages listed in a separate appendix.

```
### warning: message
 File "filename" ; line #
```

Error and fatal diagnostics have the following format:

```
### message
 File "filename" ; line #
```

Example:

```
### identifier a not declared
 File "file.c" ; line 2
```

Identifiers

There are no limitations on the number of significant characters in an identifier. The case of identifiers is preserved.

Characters

ASCII is the character set for both source and for generated code (constants, library routines).

There are no shift states for multi-byte characters.

A character consists of eight bits.

All members in the source character set are mapped to the same character in the execution set.

There may be up to four characters in a character constant. The internal representation of a character constant with more than one character is constructed as follows: as each character is read, the current value of the constant is multiplied by 256 and the value of the character is added to the result. Example:

```
'abc' == (('a'*256)+'b')*256+'c'
```

Wide characters are implemented as long integers. See also the -Xwchar=n option in Table 3-3 on page 21.

Unless specified by the use of the -Xsigned_char or -Xunsigned_char options (see Table 3-3 on page 21), the treatment of plain **char** as a **signed char** or an **unsigned char** is as defined in the *Compiler Target User's Manual.*

Integers

Integers are represented in two's-complement binary form. The properties of the different integer types are defined in the *Compiler Target User's Manual.*

The result of bitwise operations on signed integers is the same as if the operands were treated as unsigned.

The sign of the remainder on integer division is the same as that of the divisor on all processors supported by Diab Data.

The result of a right shift of a negative signed integer is arithmetic, and the sign bit is propagated to the right on all processors supported by Diab Data.

Floating point

The floating point types use the IEEE 754-1985 floating point format on all processors support by Diab Data. The properties of the different floating point types are defined in the *Compiler Target User's Manual.*

The default rounding mode is "round to nearest".

Arrays and pointers

The type required to hold the maximum size of an array is that of the type of the **sizeof** operator (**size_t**), an **unsigned int**.

Pointers are implemented as 32 bit entities. A cast of a pointer to an **int** or **long**, and vice versa, is a bitwise copy and will preserve the value.

The type required to hold the difference between two pointers, **ptrdiff_t**, is **int**

Registers

All local variables of any basic type, declared with or without the **register** storage class can be placed in registers. **struct** and **union** members can also be put in registers.

Variables explicitly marked as having the **auto** storage class are allocated on the stack.

Structures, unions, enumerations, and bit-fields

If a member of a **union** is accessed using a member of a different type, the value will be the bitwise copy of original value, treated as the new type.

See the chapter "Internal Data Representation" in the *Compiler Target User's Manual* for more information about the implementation of structures and unions, bit-fields, and enumerations.

Qualifiers

Volatile objects are treated as ordinary objects, with the exception that all read / write / read-modify-write accesses are performed prior to the next sequence-point as defined by ANSI.

Declarators

There is no limit to how many pointer, array, and function declarators are able to modify a type.

Statements

There is no limit to the number of **case** labels in a **switch** statement.

Preprocessing directives

Single-character constants in **#if** directives have the same value as the same character constant in the execution character set. These characters can be negative.

Include files are searched for in the order specified below. If the syntax **#include** "*file*" is used, searching starts with item 1. If the syntax **#include** <*file*> is used, searching starts with item 2.

1. The directory where the current preprocessed file resides.
2. The directories given with the -I option, in left to right order.
3. The directory *version_path*`/include` (see Table 2-1, "Example Default Installation Path Names," on page 5 for the definition of *version_path*).

The name of the included file is passed to the operating system (after truncation if necessary to conform to operating system limits).

The **#pragma** directives supported are described in "Pragmas" on page 45.

Environment

The function called at startup is called **main**( ). It can be defined in three different ways:

- With no arguments:

```
int main(void) { ... }
```

- With two arguments, where the first argument (`argc`) has a value equal to the number of program parameters plus one. Program parameters are taken from the command line and are passed untransformed to **main**( ) in the second argument `argv[]`. `argv[0]` is the program name. `argv[argc]` contains the null pointer.

```
int main(int argc, char **argv) { ... }
```

- With three arguments, where `argc` and `argv` are as defined above. The argument `env` is a pointer to a null terminated array of pointers to environment variables. These environment variables can be accessed with the **getenv( )** function.

```
int main(int argc, char **argv, char **env) { ... }
```

Library functions

The **NULL** macro is defined as 0.

The **assert** function, when the expression is false, will write the following message on standard error output and call the **abort** function:

`Assertion failed:` *expression*, `file` *filename*, `line` *line-number*

The **ctype** functions test for the following characters:

Table C-1 ctype Functions

Function	Decimal ASCII Value and Character
isalnum	65-90 ('A'-'Z'), 97-112 ('a'-'z'), 48-57 ('0'-'9')
isalpha	65-90 ('A'-'Z'), 97-112 ('a'-'z')
iscntr	1 0-31
isdigit	48-57 ('0'-'9')
isgraph	33-126
islower	97-112 ('a'-'z')
isprint	32-126
ispunct	33-47, 58-64, 91-97, 123-126
isspace	9-13 (TAB,NL,VT,FF,CR), 32 (' ')
isupper	65-90 ('A'-'Z')
isxdigit	48-57 ('0'-'9'), 65-70 ('A'-'F'), 97-102 ('a'-'f')

The mathematics functions do not set **errno** to ERANGE on underflow errors.

The first argument is returned and **errno** is set if the function **fmod** has a second argument of zero.

Information about available signals can be found in the target operating system documentation.

The last line of a text stream does not need to contain a new-line character.

All space characters written to a text stream appear when read in.

No null characters are appended to text streams.

A stream opened with append ('**a**') mode is positioned at the end of the file unless the update flag ('**+**') is specified, in which case it is positioned at the beginning of the file.

A write on a text stream does not truncate the file beyond that point.

D-C++ supports the following three buffering schemes; unbuffered streams, fully buffered streams, and line buffered streams.

Zero-length files exist.

The rules for composing valid file names can be found in the documentation of the target operating system.

The same file can be opened multiple times.

If the **remove** function is applied on an opened file, it will be deleted after it is closed.

If the new filename already exists in a call to **rename**, that file is removed.

The **%p** conversion in **fprintf** behaves like the **%X** conversion.

The **%p** conversion in **fscanf** behaves like the **%x** conversion.

The character '**-**' in the scanlist for **%[** conversion in the **fscanf** function denotes a range of characters.

On failure, the functions **fgetpos** and **ftell** set errno to the following values:

EBADF if file is not an open file descriptor.
ESPIPE if file is a pipe or FIFO.

The following messages are generated by the **perror** and **strerror** functions.

Error 0
Not owner
No such file or directory
No such process
Interrupted system call
I/O error
No such device or address
Arg list too long
Exec format error
Bad file number
No child processes
No more processes
Not enough space
Permission denied
Bad address
Block device required
Device busy
File exists
Cross-device link
No such device
Not a directory
Is a directory
Invalid argument
File table overflow
Too many open files

Not a typewriter
Text file busy
File too large
No space left on device
Illegal seek
Read-only file system
Too many links
Broken pipe
Argument out of domain
Result too large
No message of desired type / Operation would block
Identifier removed / Operation now in progress
Channel number out of range / Operation already in progress
Level 2 not synchronized / Socket operation on non socket
Level 3 halted / Destination address required
Level 3 reset / Message too long
Link number out of range / Protocol wrong type for socket
Protocol driver not attached / Protocol not available
No CSI structure available / Protocol not supported
Level 2 halted / Socket type not supported
Deadlock condition if locked / Operation not supported on socket
Protocol family not supported
Address family not supported by protocol family
Address already in use
Can't assign requested address
Network is down
Network is unreachable
Network dropped connection on reset
Software caused connection abort
Connection reset by peer
No buffer space available
Socket is already connected
Socket is not connected
Can't send after socket shutdown
Too many references: can't splice
Connection timed out
Connection refused
Too many levels of symbolic links
File name too long
Host is down
No route to host

The memory allocation functions **calloc**, **malloc**, and **realloc** return **NULL** if the size requested is zero. The function **abort** flushes and closes any open file.

Any status returned by the function **exit** other than **EXIT_SUCCESS** indicates a failure.

The set of environment variables defined is dependent upon which variables the system and the user have provided. See "Environment variables" on page 8. These variables can also be defined with the **setenv** function.

The **system** function executes the supplied string as if it were given from the command line.

The local time zone and the Daylight Saving Time are defined by the target operating system.

The function **clock** returns the amount of CPU time used since the first call to the function clock if supported.

D Error Messages

There are four kinds of error messages generated by D-C++:

Table D-1 Error Message Severity Codes

I	Informational: a message will be printed, compilation will continue, and an output file will be produced. Usually this is a continuation of an earlier error message with more detailed information.
W	Warning: a message will be printed, compilation will continue, and an output file will be produced.
E	Error: a m-Xpcc, message will be printed, compilation will continue, but no output will be generated.
F	Fatal: a message will be printed and compilation aborted.

C++ error messages are marked in the margin like this paragraph. Also, some messages have slightly different meanings between C and C++. Where necessary, both are given and the C versions also marked in the margin. Finally, some messages are warnings in C but errors in C++ and vice versa. These are not separated; instead, the appropriate language is marked with each severity. ???R: need to mark error messages for C.???

➤ An error message can seem to be generated for code which is apparently correct. Such a message is often the result of earlier errors which are not closely related to the message in question. If a message persists after all other errors have been cleared, please report the circumstances to Customer Support at Diab Data.

C++ **'0' expected in pure specifier**

A value other than 0 was found in a pure specifier. (E)

```
class foo {
    virtual bar() = 5; // Should have been 0
}
```

C++ **a local class can't have static data members**

Only non-static members can be used in a local class. (E)

address taken in initializer (PIC)

A static initializer containing the address of a variable or string has been found when generating position independent code. Such address values cannot be position independent. (W) or (E) depending on whether -Xstatic-addr-warning or -Xstatic-addr-error is used.

C++ **all dimensions but the first must be positive constant integral expressions**

The first dimension of an array may be empty in some contexts. In a multi-dimensional array, no other dimensions may be empty (and none may be negative). (E)

```
int array[-4];
```

C++ **all dimensions must be specified for non-static arrays**

For an array in a class all dimensions must be specified, even if the array is not static. (E)

C++ **already const**

An variable was declared **const** more than once. (W)

```
int * const const foo;
```

C++ **already volatile**

A variable was declared **volatile** more than once. (W)

```
int * volatile volatile foo;
```

C++ **ambiguous conversion -- cannot cast operand**
ambiguous conversion from *type-designator* **to** *type-designator*

The compiler cannot find an unambiguous way to convert an item from one type to another. (E)

C++ **ambiguous reference to** *identifier*, **could be** *candidate1 candidate2 ...*

The identifier couldn't be resolved unambiguously. The error message is followed by a list of possible candidates. (E)

```
struct A { int a; };
struct B { int a; };
struct C : public A, public B {};

foo()
{
    C c;
    c.a = 1; // Which a, A::a or B::a?
}
```

C++ **anonymous unions can't have members functions**
anonymous unions can't have protected or private members
anonymous unions in file scope must be static

A special rules for an anonymous unions is violated. (E)

arglist in declaration

An old style function declaration is found in the wrong context. (W)

```
f1() { int f2(a,b,c); ... }
```

argument list not terminated

The end of the source file was found in a macro argument list. (W) if -Xpcc. (E) otherwise.

argument type does not match prototype

The type of an argument to a function is not compatible with its type as given in the function's prototype. (W) if -Xpcc or -Xk-and-r or -Xmismatch-warning, (E) otherwise.

```
int f(char *), i; ... i = f(&i);
```

C++ **argument type is abstract**
A function argument type has pure virtual functions. (E)

array is incomplete
An array of undefined size is used with an index. (W) if -Xpcc, (E) otherwise.

```
int a[]; a[5] = 6;
```

C++ **array is incompletely specified**
An array cannot be declared with an incomplete type. (E)

asm macro line too long
A very long line was given in an asm macro. See "asm and __asm" on page 49 for more details. (E)

bad include syntax
The **#include** directive is not followed by '<' or '"'or the file name is too long. (W) if -Xpcc, (E) otherwise.

bad octal constant
A numeric constant with a leading zero is an octal constant and can only contain digits '0' through '7'. (W)

```
i = 078;
```

bad #pragma [*name* **]**
If issued without the *name* the compiler did not recognize the pragma. If issued with a *name*, there is a problem with either the operands to the pragma or the context in which it appears. (W)

C++ **base class expected**
Base class not found after ':' or ',' in a class definition. (E)

```
class A : {}; // The base class is missing
```

bitfields must be int or unsigned
The compiler does not support bitfield types other than **int** or **unsigned int**. (E)

```
struct { float a:4; };
```

C++ **can't construct objects of abstract type**
The type in a **new** expression is of abstract class. (E)

```
struct A {
    virtual foo() = 0;
};
A *p = new A;
```

C++ **can't construct objects of array type**
Array elements in an array allocated with **new** cannot be given initial values. (E)

```
struct A {};
A *p = new A[5](1,2,3,4,5);
```

C++ **can't create void objects**
can't declare void objects
The type in a **new** expression was void.

```
void *p = new void;
```

C++ **can't delete pointer to const objects**
The argument to **delete** points to a **const** object. (E)

```
void foo (const int *p)
{
    delete p;
}
```

C++ **can't distinguish** *function_name1* **from** *function_name2*
Two overloaded functions cannot be distinguished from each other, that is, they effectively have the same number and types of arguments in the same order. (E)

```
int foo(int);
int foo(int &);
```

C++ **can't enable access to** *identifier*
can't restrict access to *identifier*
An access declaration can't restrict access to a member that is accessible in the base class, nor can an access declaration enable access to a member that is not accessible in the base class. (E)

can't find current module in profile file ...
No data about the current source file is available in the profiling file. (W)

Possible causes:

- No function in the current file was actually executed during profiling.
- The profiling file belongs to another executable program.

can't find include file *unknown*
The preprocessor cannot find a file named in an **#include** directive. (W) if -Xpcc, (E) otherwise.

can't find program *program-name*
program-name will be the name of some component of the compiler or other Diab Data tool. (F)

Possible causes:

- D-C++ is not installed properly.
- One of the D-C++ files has been deleted, hidden, or protected.
- The `dtools.conf` or other configuration file is incorrect.

can't fork
The system cannot start a new process. (F)

C++ **can't have a destructor in a nameless class/struct/union**
A nameless class cannot have a destructor since the destructor takes its name from the class. (E)

```
class {
    ~foo();
};
```

can't init arguments

It is not valid to attempt to initialize function parameters. (E)

```
f(i) int i = 5; { ... }
```

can't init typedefs

A **typedef** declaration cannot have an initializer. (E)

```
typedef unsigned int uint = 5;
```

can't initialize variable of type *type_designator*

Some types do not allow initialization. (E)

```
void a = 1;
```

C++ **can't initialize ... with a list**

An objects of a class which has constructors, bases, or non-public member cannot be initialized as an aggregate.

```
struct foo {
    private:
        int i
    public:
        int j, k;
};

foo bar = { 1, 2, 3 }; // i is private
```

C++ **can't nest function definitions**

Functions cannot be defined inside other functions.

```
void foo()
{
    void bar() { } // No nesting
}
```

can't open *filename* **for input**

can't open *filename* **for output**

The given *filename* cannot be opened. (F)

can't open .cd file!

The compiler reads a target description file, normally named *target*`.cd`, during initialization (see "Targets" on page 7 in Table 2-2). Either the required file cannot be found or appears to be incorrect. (F)

Possible causes:

- D-C++ is not installed properly.
- One of the D-C++ files has been deleted, hidden, or protected.

- The `dtools.conf` or other configuration file is incorrect.
- Incorrect use of the -M option.

can't open profiling file *filename*
The file given with the -Xfeedback=*filename* option cannot be opened. (W)

can't push *identifier*
It is invalid to use an expression of type function or **void** as an argument. (E)

```
void *pv; int (*pf)(); fn(*pv,*pf);
```

can't recognize storage mode *unknown*
The storage mode specified in an **asm** macro is unknown. See "asm and __asm" on page 49 for more details. (E)

can't recover from earlier errors
Certain earlier errors have made it impossible for the parser to continue. (F)

can't take address of object
Trying to take the address of a function, constant, or register variable that is not stored in memory.(E)

```
register int r; fn(&r);
```

C++ **can't use ... in default argument expression**
Class members can only be used in default arguments if they are static. Function arguments cannot be used in default arguments. Local variables cannot be used unless they are declared **extern**. (E)

```
int foo(int a, int b = a)
{
    ...
}
```

C++ **can't use typedef name (*typedef-name*) after class, struct or union keyword**
A typedef name cannot be used as a **class**, **struct**, or **union** name. (E)

```
struct A { };
typedef A B;
struct B foo; // B is a typedef name. Use A instead
```

case constant should be integral
The expression in a **case** statement within a **switch** is not an integer. (E)

```
#define PORT ((char *)0xffffff00)
 ...
case PORT:
```

C++ **cast to abstract type**
Objects cannot be cast to an abstract class. (E)

C++ **.cd file is of wrong type!**
.cd file is of wrong version!
.cd file *filename* **too small?!**
See "can't open .cd file!" above.

class *class-name* **has no constructor**

It is invalid to initialize an object that does not have a constructor by using the constructor initialization syntax. (E)

```
struct A {
    int b, c;
};
A a(1,2);
```

C++ **class** *class-name* **used twice as direct base class**

Cannot use the same class as a base class more than once. (E)

```
class A {};

class B : A, A {};
```

C++ **class already has operator delete with one argument**
class already has operator delete with two arguments

The **delete** operator cannot be overloaded. (E)

C++ **class name expected after ~**

Encountered '~' in a class, apparently to declare a destructor, but it was not followed by the class name. (E)

```
class foo {
    ~;
};
```

C++ **class/struct/union cannot be declared** *specifier*

A function specifier is applied to a definition of a **class**, **struct**, or **union**. (E)

```
inline class foo {     /* inline is invalid for a class */
    ...
};
```

C++ **comma at end of enumerator list ignored**

A superfluous comma at the end of a list of enumerators was ignored. (W)

```
enum foo { bar, };
```

compiler error ...

The compiler has detected an internal error. May result from other errors reported earlier. If the problem does not appear to be a consequence of some earlier error, please report it to Diab Data. (F)

C++ **compiler out of sync: mismatching parens in inline function**

The compiler is unable to parse an inline function. Check the function to see if the parentheses are nested correctly. (F)

compiler out of sync. Probably missing ';' or '}'

The compiler is unable to parse what appears to be an external declaration. Could be due to a missing or misspelled token or keyword. (E)

```
int i int j;        # missing ';' after i
duoble f;           # should be "double"
```

C++ **conflicting declaration specifiers:** *specifier1 specifier2*

Illegal mixing of **auto**, **static**, **register**, **extern**, **typedef** and/or **friend**. (E)

```
extern static int foo;
```

C++ **conflicting type declarations**

More than one type specified in a declaration. (E)

```
int double foo;
```

constant expression expected

The expression used in an enumerator list is not a constant. (E)

```
enum a { b = f(), c };
```

constant out of range [*operator* **]**

A constant is out of the range of the context in which it is used. If the operator is present, it shows the operator near the use of the invalid constant. (W)

```
short int i = 32768;        range of short int is -32,768 .. 32,767
if ((char) c == 257)        range of char is 0 .. 255
```

C++ **constructors and destructor may return no value**

A constructor or destructor may not return a value. (E)

C++ **constructors can't be static**
constructors can't be declared const or volatile
constructors can't be virtual
constructors can't have a return type
constructor is illformed, must have other parameters

A constructor declaration is invalid. (E)

C++ **conversion functions must be members of a class**
conversion functions may take no arguments
conversion to original class or reference to it

It is not valid to define a conversion function that is not a class member. A conversion function cannot take arguments. A conversion function cannot convert to the type of the class if it is a member of or a reference to it.(E)

C++ **destructors must have same name as the class/struct/union**
destructors may have no return type
destructors can't be declared const or volatile
destructors can't be static
destructors may take no arguments

The destructor declaration is invalid. (E)

division by 0

The compiler has detected a source expression that would result in a divide by 0 during target execution. (W)

```
int z = 0; fn(10/z);
```

don't know size of object

The **sizeof** operator is used on an incompletely specified array or undefined struc-

ture, or an array of objects of unknown size is declared. (E)

```
extern int ar[]; sz = sizeof(ar);
```

duplicate default statements
duplicate case constants

A **case** constant, or the default statement should not occur more than once in a switch statement. (E)

```
case 1: ... case 1:
```

C++ **ellipsis not allowed as argument to overloaded operator**

Cannot declare an overloaded operator with "..." as arguments. (E)

ellipsis not allowed in pascal functions

Functions declared with the **pascal** keyword are not allowed to have a variable number of arguments as indicated by an ending ellipsis (...). (E)

empty character constant

There are no characters in a character constant. If an empty string is wanted, use string quotes " ". (E)

```
int i3 = '';      /* This is two single quotes characters. */
```

end of memory

Ran out of virtual memory during compilation. D-C++ first attempts to skip some optimizations in order to use less memory, however this error can occur for really large functions on machines with limited memory. **Note:** initialized arrays require the compiler to hold all initial data and can contribute to this error. (F)

C++ **enumerators cannot have external linkage**

extern cannot be specified for **enum** declarations. (E)

```
extern enum foo { bar };
```

EOF in comment

The source file ended in a comment. (W) if -Xpcc, (E) otherwise.

C++ **EOF in inline function body**

The end of the source file was found while parsing an inline function. (F)

EOF inside #if

The source file ended before a terminating **#endif** was found to match an earlier **#if** or **#ifdef**. If not caused by a missing **#endif**, then frequently caused by an unclosed comment or unclosed string. (W) if -Xpcc, (E) otherwise.

EOF in character constant
EOF in string constant

The source file ended at an unexpected place during parsing. (F)

C++ **exception handling disabled**

Exception handling has been turned off. Use -Xexception=1 to enable it. (E)

C++ **expression expected**

Could not find an expression where it was expected. (E)

```
if () { // The condition is missing.
    ...
}
```

expression not used

The compiler has detected all or part of an expression which will never be used. (W)

```
a+b;            /* statement with no side effects */
a=(10,b+c);     /* 10 is not used */
*s++;           /* the '*' is not needed: s++; */
```

Note: the compiler will not issue this warning for an expression consisting solely of a reference to a volatile variable.

expression too complex. Try to simplify.

Can occur if an expression is too complex to compile. Should not happen on most modern processors. Can occur on a processor with few registers and no built-in stack support. (F)

C++ **extern objects can only be initialized in file scope**

Attempt to initialize an **extern** object in a function. (E)

```
foo()
{
    extern int one = 1;
}
```

C++ **first dimension must be an integral expression**

The first dimension of an array type in a **new** expression must be an integral expression. (E)

```
double d;
int *p = new int[d];
```

floating point error

Floating point exception occurred during compilation. This can occur when attempting to process a floating point constant expression in the source code. Simplify or reorganize the expression. (F)

floating point overflow

Floating point overflow occurred during constant evaluation. See the previous error. (E)

floating point value (...) out of range

A floating point constant exceeds the range of the representation format. (E)

```
double d = 1e10000;
```

found #elif without #if
found #else without #if
found #endif without #if

Found an **#elif**, **#else**, or **#endif** directive without a matching **#if** or **#ifdef**. (W) if -Xpcc, (E) otherwise.

C++ **friend declaration outside class/struct/union definition**

The keyword **friend** is used in a invalid context (E)

```
friend class foo {
    ...
};
```

C++ **friend declaration outside class declaration**

Attempt to declare a function as a **friend** outside of a class. (E)

```
friend int foo();
```

C++ **friend** *function-name* **not in** *template-name*

Could not find a friend during template instantiation. (E)

C++ **friends can't be virtual**

Because friends are not members of the class they cannot be virtual. (E)

function-declaration in wrong context

A function may not be declared inside a **struct** or **union** declaration. (E)

```
struct { int f(); };
```

C++ **function** *function-name* **already has "C" linkage**

Only one of a set of overloaded functions can have "C" linkage. (E)

```
extern "C" foo(int);
extern "C" foo(double);
```

C++ **function must return a value**

Found a **return** statement with no value in a function. (E)

```
int foo()
{
    return; // Must return a value.
}
```

function *funcion-name* **not declared**

If the -Xforce-declarations option is used, D-C++ will generate this error message if a function is used before it has been declared. (W)

C++ *name* **has no default constructor, must be initialized**
non-extern object *name* **of type** *type-name* **must be initialized**
non-extern const object *name* **must be initialized**
non-extern reference *name* **must be initialized**

References and **const** objects, which are not declared **extern**, must be initialized. So must objects of classes that have constructors but no default constructors. (E)

C++ **identifier ... declared as union. Use union specifier**
identifier ... declared as struct or class. Use struct or class specifier

There was a type mismatch between the declaration and the use of an identifier. (E)

```
union u {
    ...
};
```

```
struct u foo; // u was a union, cannot also be struct
```

C++ **identifier** *name* **is not a class**
identifier *name* **not a nested class nor a base class**

An identifier that is not a class was used before "::" or something that is not a class was used as a base class. (E)

C++ **identifier** *name* **is not a direct member**

Attempt to initialize a variable that is not a direct member of the class. (E)

```
struct B { int b; };

struct C : public B {
    int c;
    C(int i) : c(i), b(i) {} // Can't initialize b here
}
```

C++ **identifier** *name* **is not a direct nor a virtual base**

Only classes that are direct bases or virtual bases can be used in a member initializer. (E)

```
struct A { A(int); };
struct B : public A { B(int); };

struct C : public B {
    C(int i) : A(i) {} // Can't initialize A here
}
```

C++ **identifier** *identifier* **is not a member of class** *class-name*

The identifier to the right of :: is not in the class on the left side. (E)

C++ **identifier** *identifier* **is not a type**

What appeared to be a declaration began with an identifier that is not the name of a type.

identifier *identifier* **not an argument**

An identifier that is not in the parameter list was encountered in the declaration list of an old-style function. (E)

```
f(a) int b; { ... }
```

identifier *identifier* **not declared**

An identifier is used but not declared. Check the variable name for spelling errors. (E)

identifier *identifier* **not member of struct/union**

The expression on the right side of a '.' or '->' operator is not a member of the left side's **struct** or **union** type. (E)

C++ **identifier** *name* **previously declared** *linkage*

The identifier was already declared with another linkage specification. (E)

```
int foo;
extern "C" int foo;
```

C++ **identifier *name* used as template name in this scope**

The identifier cannot be used as a **class**, **struct**, or **union** tag since it is already a template name. (E)

```
template<class T>
class foo {
    ...
};

struct foo {
}
```

C++ **identifier *name* used both as member and in access declaration**

A use of the *name* would be ambiguous. (E)

```
class A {
    public:
    int foo;
};

struct B : private A {
    int foo;
    A::foo;
};
```

illegal assert name

An **#assert** name must be an identifier and must be preceded by a '#' character. (W) if -Xpcc, (E) otherwise.

illegal break

A **break** statement is only allowed inside a **for**, **while**, **do** or **switch** statement. (E)

illegal continue

A **continue** is only allowed inside a **for**, **while** or **do** statement. (E)

illegal case

A **case** label is only allowed inside a **switch** statement. (E)

illegal cast

An attempt is made to perform a cast to an invalid type, i.e., a structure or array type. (E)

```
a = (struct abc)10;
```

illegal character: 0*n* (octal)

The source file contains a character with octal code *n* that is not defined in the C language. This can occur only outside of a string constant, character constant, or comment. (E)

```
name$from$PLM = 1;
```

illegal default

A default label is only allowed inside a **switch** statement. (E)

illegal declaration

Common causes and examples: (E)

A scalar variable can only be initialized to a single value of its type.	`int i = 1, 2;`
Functions cannot return arrays or functions.	`char fn()[10];`
Variables cannot be of type **void**. (Usually caused by a missing asterisk, e.g.,`void *p;` is correct.)	`void a;`
Only one **void** is allowed as function argument.	`int fn(void,void);`
An array cannot contain functions.	

illegal declaration-attribute

A declaration contains an invalid combination of declaration specifiers. (W)

```
unsigned double foo;
```

C++ **illegal expression**

There was something wrong with the expression. Another error has probably already been reported. (E)

C++ **illegal function specifier for argument**

A parameter cannot be declared **inline** or **virtual**.

```
void foo(inline int);
```

illegal hex constant

Reported whenever an 'x' or 'X' is found in a numeric constant and is not prefixed with a single zero. (E)

```
i = 1xab;
```

illegal initializer

An initializer is not of the proper form for the object being initialized. Often caused by a type mis-match or a missing member in a structure constant. (E)

illegal lvalue

Only certain expressions can be on the left hand side of an assignment. (E)

```
a+b = 1;
```

illegal macro definition

illegal macro name

Macro names and arguments must start with a letter or underscore. (W) if -Xpcc, (E) otherwise.

illegal option -D*invalid_name*

The preprocessor was invoked with the -D option and an invalid name. Names must start with a letter or underscore. (W)

illegal output name *filename*

Specific output file names given with the -o option are invalid in order to avoid common typing mistakes. (F)

```
dplus a.c -o b.c # b.c is an illegal output file name
```

illegal redefinition of *macro_name*

__LINE__, **__FILE__**, **__DATE__**, **__TIME__**, **defined**, and **__STDC__** cannot be redefined. (W) if -Xpcc, (E) otherwise.

illegal storage class

External variables cannot be **automatic**.
Parameters cannot be **automatic**, **static**, **external**, or **typedef**. (E)

```
int fn(i)
static int i; { ... }
```

C++ **illegal storage class for class member**

A class member cannot be **auto**, **register**, or **extern**. (E)

C++ **illegal storage class for class/struct/union**

A storage class other than **extern** is specified for a definition of a **class**, **struct** ,or **union**. (E)

```
auto class foo {
    ...
};
```

illegal type(s): *type-signatures*

The operators of an expression do not have the correct or compatible types. (W) if -Xpcc or -Xk-and-r or -Xmismatch-warning, (E) otherwise.

```
int *pi, **ppi; ... if (pi == ppi) ...
illegal types: ptr-to-int '==' ptr-to-ptr-to-int
```

illegal type of switch-expr

A **switch** expression is of a non-integral type. (E)

C++ **illegal union member**

An object of a class with a constructor, a destructor, or a user defined assignment operator cannot be a member of a union. (E)

illegal width of bitfield

A bitfield width is greater than the underlying type used for the bitfield. (E)

Example for a target with 32 bit integers:

```
struct { int i:33; }
```

include nesting too deep

The preprocessor cannot nest include files deeper than 20 levels. (W) if -Xpcc, (E) otherwise.

C++ **inconsistent exception specifications**

Two function declarations specify different exceptions. (E)

```
int foo() throw (double);
int foo() throw (int);
```

C++ **inconsistent storage class specification for** *name*

The identifier was already declared, with another storage class. (E)

```
bar()
{
    int foo; // foo is auto by default
    static int foo; // now static
}
```

C++ **insufficient access rights to** *member-name* **in** *base-class-name* **base class of** *derived-class-name*

Attempt to access a member in a private or protected base class. (E)

int constant expected

The type of expression used in a **case** should be an integral constant. (E)
A bitfield width must be an integer constant. (E)

integer constant expression expected

The size of an array must be computable at compile time. (E)

```
int ar[fn()];
```

internal table-overflow

Internal stack overflowed. May rarely occur with extremely complex, deeply nested code. To work-around, simplify or modularize the code. (F)

invalid option *unknown*

D-C++ was started with an unrecognizable -W or -Y option. Note: -X options that are not recognized generate an "unknown option" message, and unrecognized but otherwise valid non -X options are passed to the linker. (F)

label *identifier* **already exists**

A label can only refer to a single place in a function. (E)

label *identifier* **not defined**

The label used in a **goto** statement is not defined. (E)

logic error in *internal-identification*

The compiler has detected an internal error. May result from other errors reported earlier. If the problem does not appear to be a consequence of some earlier error, please report it to Diab Data. (F)

macro *identifier*: **argument mismatch**

Either too few or too many arguments supplied when using a macro. (W) if -Xpcc, (E) otherwise.

```
#define M(a,b) (a+b)
i = M(1,2,3);
```

macro *identifier* **redefined**

An already defined macro is redefined: (W)

```
#define A 1
#define A 2
```

C++ **main can't be declared inline**
main can't be declared static
main can't be overloaded
main can't be called recursively
can't take address of main

Special rules for the function **main**( ) are violated. (E)

C++ **mem initializers only allowed for constructors**

Members can only be initialized with the member initializer syntax in constructors. (E)

```
class A {
    int i;
    int foo() : i(4711) {} // Not a constructor
}
```

C++ **member** *name* **already declared**

Attempt to re-declare a member. (E)

```
class A {
    int a;
    int a; // Already declared
};
```

member declaration without identifier

A **struct** or **union** declaration contains an incomplete member having a type but no identifier. (W)

```
struct foo { int; ...};
struct { struct bar { ... }; ... };
```

C++ **member function declared as friend in its own class**

Invalid declaration. (E)

```
class A {
    foo(int);
    friend A::foo(int);
}
```

C++ **member function of local class must be defined in class definition**

Because functions cannot be defined in other functions, any function in a local class must be defined in the class body. (E)

C++ **member function declared as friend in its own class**

Invalid declaration. (E)

```
class A {
    foo(int);
    friend A::foo(int);
}
```

C++ **member function of local class must be defined in class definition**

Because functions cannot be defined in other functions, any function in a local class must be defined in the class body. (E)

member is incomplete

The structure member has an incomplete type, i.e., an empty array or undefined structure. (E)

```
struct { int ar[]; };
```

C++ **member is incomplete**

Cannot declare members of incomplete type. (E)

C++ **member of abstract class**

A class member cannot be of abstract type. (E)

C++ **member operator functions can't be static**

Operator functions in a class cannot be declared static. (E)

C++ **member *name* redeclared outside class/struct/union**

Invalid declaration. (E)

```
struct A {
    int i;
};

int A::i;
```

C++ **member *name* used outside non-static member function**

Attempt to reference a class member directly in a static member function or an inlined friend function. That is invalid in a function where keyword **this** cannot be used. (E)

C++ **mismatching parenthesis, bracket or ? : around expression**

Mostly likely, a parenthesis or bracket was left of an expression, or the '**?**' and '**:**' in a conditional expression where interchanged. (E)

```
int i = (5 + 4]; // ] should have been a )
```

C++ **missing argument declaration**

Argument declaration omitted. (E)

```
class bar {
    foo(, int);
};
```

missing comma in -Y option

The -Y*c,dir* option must include a comma. (F)

C++ **missing declarator in typedef**

No declarator was given in a **typedef** statement. (E)

```
typedef class foo {
    // ...
};
```

C++ **missing expression inside parenthesis**
missing expression inside brackets

An expression was expected between the parentheses or brackets. (E)

```
int x[5];
int i = x[]; // x must be subscripted
```

C++ **missing operand**
missing operand before ...
missing operand for function style cast
missing operand for operator ...
missing operand for operator ... inside brackets
missing operand for operator ... inside parenthesis
missing operand somewhere before :
An operand was left out of an expression. (E)

```
i = 3 j; // a binary operator is needed between '3' and 'j'
```

multiple initializations
A variable was initialized more than once. (E)

```
static int a = 4;
static int a = 5;
```

C++ **must be address of object, function or static class member**
A template was instantiated with an incorrect argument. (E)

C++ **name of anonymous union member** *name* **already defined**
An identifier with the same name as an anonymous union member was already declared in the scope. (E)

```
int i;
static union {
    int i; // i already declared
}
```

negative subscript not allowed
The size if an array cannot be negative. (E)

```
int ar[-10];
```

C++ **nested class may not have same name as its surrounding class**
nested type may not have same name as its class
enumerator may not have same name as its class
anonymous union member may not have the same name as its class
static data member may not have the same name as its class
typedef may not have the same name as its class
Only constructors and destructors for a class may have the same name as the class. (E)

newline in character constant
The end of a line was found while parsing a character constant. Usually caused by a missing single quote character at the end of the constant. (E)

```
char TAB = '\t;
```

newline in string constant
The end of a line was found while parsing a string constant. Usually caused by a

missing double quote character at the end of the constant. (E)

```
char * message = "Not everything that counts can be counted.
```

C++ **no default arguments for overloaded operators**

Overloaded operators cannot have default arguments. (E)

C++ **no initializers for members**

A pure specifier was found after a non-function member.

```
class A {
    int foo = 0; // Only functions can be pure
}
```

no matching asm pattern exists

While scanning an **asm** macro, no storage-mode-line matching the given parameters was found. See "asm and __asm" on page 49 for details.

C++ **no redefinition of default arguments**

A argument can be given a default value only once in a set of overloaded functions. (E)

```
void foo(int = 17);

void foo(int = 4711);
```

C++ **no return type may be specified for conversion functions**

The return type of conversion function is implicit. (E)

```
class foo {
    double operator int(); // Cannot specify type
}
```

C++ **no type found in new expression**

Could not find a type in the **new** expression.(E)

non-unique struct/union reference

In PCC mode (-Xpcc) the compiler attempts to locate a member of another **struct** if given an invalid reference. If no unique member can be found, this error is issued. (E)

```
struct a { int i; int m; };
struct b { int m; int n; };
int i; ... i->m = 1;
```

C++ **not a class/struct/union expression before ...**

The left hand side of a '**.**' or '**.***' or '**->**' or '**->***' operator must be of type class or pointer to class. (E)

```
5->a = 128; // 5 is not a pointer to a class
```

not a struct/union reference

The left hand side of a '**->**' or '**.**' expression is not of **struct** or **union** type. If -Xpcc is specified the offset of the given member name in another **struct** or **union** is used. (W) if -Xpcc, -Xk-and-r, or -Xmismatch-warning. (E) otherwise.

not enough memory for reaching analysis.
Certain optimizations, called "reaching analysis", will be skipped if the host machine cannot provide enough memory to execute them. D-C++ continues, but produces less than optimal code. (W)

C++ **object can't have internal linkage**
A template cannot be instantiated with the address of a static object. (E)

```
template<int *ptr>
class foo {
    ...
};

static int bar;

foo<&bar> qux;
```

C++ **object of abstract class**
Attempt to declare an object of an abstract class. (E)

C++ **old style function definition**
A function was defined using the older K & R C syntax. This is invalid in C++. (W)

```
int foo(a, b)
int a, b
{
    ...
}
```

C++ **only functions can have pascal calling conventions**
only functions can be inline
only member functions can be virtual
Can't declare a variable to be inline, virtual or to have pascal calling conventions. (E)

```
int pascal i;
```

only integrals allowed as bitfields
A bitfield has a more complex type than **int** or **unsigned int**. (E)

```
struct { int *a:2; }
```

C++ **only non-static member functions can be const or volatile**
only non-static member functions can be virtual
A static member function or a non-member function has been declared const, volatile or virtual.(E)

```
class A {
    static foo() const;
};
```

C++ **only virtual functions can be pure**
Pure specifier found after non-virtual function. (E)

```
class foo {
```

```
    bar() = 0; // Must be virtual
};
```

C++ **operator ... can not be overloaded**
operator ... can only be overloaded for classes

It is not allowed to overload any of the operators . or .* or ? : . The operators , and = and the unary & can only be overloaded for classes. (E)

C++ **operator ? without matching :**

Operator ? must be followed by a : . (E)

```
int i = 4 ? 5; // Missing : part
```

C++ **operator :: followed by operator ::**

Multiple :: operators are not allowed. (E)

C++ **operator->( ) not defined for ...**
operator->( ) must return class, reference to class or pointer to class
operator->( ) has bad return type

If a class object is used on the left hand side of the -> operator, the operator->( ) must be defined for that class. The return type for operator->( ) must be a class with operator->( ) defined or a pointer to that class. (E)

C++ **operator = must be a non-static member function**
operator ( ) must be a non-static member function
operator [ ] must be a non-static member function
operator -> must be a non-static member function

These operators can only be defined for classes. (E)

C++ **operator function must take other argument than basic types**

A non-member operator function must take at least one argument, which is of a class or enum type or a reference to a class or enum type. (E)

C++ **operator new must have a first argument of type size_t**
operator new must return void *
operator new cannot be virtual
operator delete takes one or two arguments
operator delete must have a first argument of type void *
operator delete's second argument must be of type size_t
operator delete must return void
operator delete cannot be virtual

One of the many special rules for the operator **new** or operator **delete** has been violated. (E)

parameter decl. not compatible with prototype

There is a mismatch between a prototype and the corresponding function declaration in either number of parameters or parameter types. (E)

```
int fn(int, int);
int fn(int a, float b) { ... }
```

possibly '=' instead of '==' ?

Encountered a conditional where the left hand side is assigned a constant value: (W)

```
if (i = 0) ... /* should possibly be (i == 0) */
```

C++ **previously declared as ...**

A function has be redeclared to something else. (E)

```
int i(int);
double i(int);
```

profile file is of wrong version (*filename*)
profile information out of date

The file given with the -Xfeedback option is out of date or has an old format. Re-compile with the -Xblock-count option and create a new profiling file. (E)

profile file *filename* is corrupted

The file given with the -Xfeedback option is corrupted. Re-compile with the -Xblock-count option and create a new profiling file. (E)

program *tool-name* terminated

The given executable has detected an internal error. May result from other errors reported earlier. If the problem does not appear to be a consequence of some earlier error, please report it to Diab Data. (F)

redeclaration of *identifier*

It is invalid to redeclare a variable on the same block level. (E)

```
int a; double a;
```

redeclaration of function

A function was already declared. May be caused by mis-typing the names of similar functions. (E)

redeclaration of member *identifier*

A member occurs more than once in a **struct** or **union**. (E)

```
struct { int m1; int m1; };
```

redeclaration of parameter *identifier*

One of a function's parameters is shadowed by a declaration within the function. (W) if -Xpcc or -Xk-and-r. (E) otherwise.

```
f1(int a) { int a; ... }
```

redeclaration of struct/union/enum ...

A **struct**, **union**, or **enum** tag name was used more than once: (E)

```
struct t1 { ... }; struct t1 { ... };
```

redeclaration of symbol ...

A symbol in an enumerated type clashes with an earlier declaration. (E)

C++ **redefinition of function**

The function is already defined. (E)

```
int foo() {}
int foo() {}
```

C++ **redundant semicolon ignored**

Found an extra semicolon among the members of a function. (W)

```
class A {
    int a;
    ;
};
```

C++ **return type not compatible with ...**

A virtual function has a return type that is incompatible with the return type of the virtual function in the base class. (W)

returning from function with address of local variable

A **return** statement should not return the address of a local variable. That stack area will not be valid after the return. (W)

```
int i;
return &i;
```

C++ **second argument to postfix operator '++' or '--' must be of type int**

The argument is of the wrong type. (E)

```
struct A {
    operator++(double); // Arg type must be int
};
```

C++ **shadowing virtual function in base class ...**

A function with the same name as a virtual function in the base class is declared with another set of arguments. The function will not be virtual unless declared so and it will hide the virtual function in the base class. (W)

```
struct A {
    virtual foo(int);
};

struct B : public A {
    foo(unsigned int); // Not virtual! It hides
    A::foo(int)
};
```

C++ **static function declared in a function**

There is no use declaring a static function inside another function. (E)

```
void foo()
{
    static void bar();

    bar(); // Call to bar, but where can it be defined?
}
```

static/external initializers must be constant expressions

Static initializations can only contain constant expressions. (E)

```
static int i = j+3;
```

C++ **static member ... cannot be initialized**

A static class member cannot be initialized in a member initializer. (E)

```
class A {
    static int si;
    A(int ii) : si(ii) {}
};
```

C++ **static only allowed inside { } in a linkage specification**

Attempt to declare a static object in a one-line linkage specification. (E)

```
extern "C" static int i; // static + extern at same time?
```

C++ **string literal expected in asm definition**

String missing in asm statement.

```
asm();    // the parentheses should contain an instruction
```

string too long

A string initializer is larger than the array it is initializing. (E)

```
char str[3] = "abcd";
```

structure is incomplete for variable *identifier*

A variable with a structure type that is not fully defined is used in a place where its size must be known. (W) if -Xpcc, (E) otherwise.

C++ **subsequent argument without default argument**

Only the trailing parameters may have default arguments. (E)

```
void foo(int = 4711, double);
```

C++ **syntax error - catch handler expected after try**
syntax error - catch without matching try
syntax error - class key seen after type. Missing ;?
syntax error - class name expected after ::
syntax error - colon expected after access specifier
syntax error - constant expression expected after keyword case
syntax error - declarator expected after ...
syntax error - declarator expected after type
syntax error - declarator or semicolon expected after class definition
syntax error - else without matching if
syntax error - identifier expected after ...
syntax error - identifier expected after class in template parameter
syntax error - initializer expected after =
syntax error - keyword operator must be followed by an operator or a type specifier
syntax error - parameter name expected
syntax error - template parameter list cannot be empty
syntax error - type tag expected after keyword enum
syntax error - unexpected end of file
syntax error after ..., expecting ...

The parser has found an unexpected token: (E)

syntax error in #if

The expression in an **#if** directive is incorrect. (W) if -Xpcc, (E) otherwise.

```
#if a *
```

syntax error near *token*

The parser has found an unexpected token. (E)

```
if (a == 1 { /* missing ')' */
```

C++ **template argument is not an exact match**

The actual template argument types must match the declaration exactly. (E)

```
template<int size>
class foo {
    // ...
};

foo<12.0> qux;
```

C++ **template function ... already defined**

A template function is redeclared.

```
template<class T> T add(T a, T b) { return a + b; }
template<class T> T add(T a, T b) { return a + b; }
```

C++ **this cannot be used outside a class member function body**

The keyword this was used outside the body of a (non-static) class member function. (E)

test version of dcc: File is too big!
test version of dcc: Can't continue!

These errors are generated when certain limits in an evaluation copy of D-C++ are exceeded. (F)

too complex #if expression

The expression in an **#if** directive overflowed an internal stack. This is unlikely to happen in the absence of other errors. (W) if -Xpcc, (E) otherwise.

too long constant

A numeric constant is longer than 256 characters. (E)

C++ **too many arguments for function style cast to basic type**
too many arguments for function style cast to union

Function style casts to a basic type or a union type can only take a single argument. (E)

```
int i = int(3.4, 5.6);
```

too many characters in character constant

A character constant has more than four characters. The limit is four on 32 bit machines. (E)

```
int i1 = 'abcd';  /* ok */
int i2 = 'abcde'; /* not ok */
```

too many declaration levels
An internal stack overflowed. This is unlikely to happen in the absence of other errors. (F)

too many enhanced asm parameters
There can be a maximum of 20 parameters and labels used in an **asm** macro. See "asm and __asm" on page 49 for details.

too many errors
D-C++ has found so many errors that it does not seem worthwhile to continue. (F)

too many initializers
The number of initializers supplied exceeds the number of members in a structure or array. (E)

```
int ar[3] = { 1,2,3,4 };
```

C++ **too many operands inside parenthesis**
too many operands inside brackets
An operator between the operands is missing. (E)

```
int i = (4 5);
```

C++ **too many parameters for operator ...**
too few parameters for operator ...
Overloaded operator declared with too many or too few parameters. (E)

trying to assign 'ptr to const' to 'ptr'
A pointer to a **const** cannot be assigned to an ordinary pointer. (E)

```
const int *pc; int pi; ... pi = pc;
```

trying to modify 'const' expression
A variable or reference declared with the **const** keyword can only be initialized, not assigned to. (E)

```
const a = 5; a = 6;
```

C++ **type ... is incomplete**
Attempt to access a member in an incomplete type. (E)

```
struct A *ap; // A is not yet defined

ap->foo = 5; // Who knows what A looks like, or foo()?
```

C++ **typedef specifier may not be used in a function definition**
Bad use of the typedef specifier. (E)

```
typedef int foo()
{
}
```

C++ **typedefs cannot have external linkage**
Linkage specification ignored for typedef, cannot have "C" or "C++" linkage. (W)

```
extern "C" typedef int foo;
```

C++ **type definition in bad context**

A type was defined where it wasn't allowed. (E)

C++ **type definition in condition**

Types cannot be defined in conditions. (E)

```
if (struct foo { int i; } bar) {
    // ...
}
```

C++ **type defined in return type (forgotten ';'?)**

It is not allowed to define a type in the function return type. (E)

```
struct foo {} bar()
{
}
```

C++ **type definition not allowed in argument list**

Types cannot be defined in argument lists. (E)

```
int foo( struct bar { int a; } barptr);
```

C++ **type expected in arg-declaration-clause**

An argument type is missing in a function declaration. (E)

```
class bar {
    foo(imt);
};
```

C++ **type expected in cast**

type expected

Found something else where a type was expected. (E)

C++ **type expected in template argument**

Found something that wasn't a type where a type was expected. (E)

```
template<class T>
class foo {
    // ...
};

foo<4711> bar;
```

C++ **type expected for ...**

No type found in declaration of a variable. (E)

C++ **type in new expression may not contain class/struct/enum declarations**
type in new expression may not contain enum declarations
type in new expression can't be const or volatile)
type in new expression can't be pascal
type in new expression can't be asm
type in new expression is abstract
type in new expression is incompletely specified

Can't declare types in a **new** expression. Nor can the types used in a **new** expression be const, volatile, pascal or asm. The type used must be completely specified and cannot have pure virtual functions. (E)

```
void *p = new enum foo { bar };
```

undefined control

Undefined or unsupported directive found after #. (W) if -Xpcc, (E) otherwise.

```
#unknown
```

C++ **unions may not have base classes**
unions can't be base classes
unions can't have virtual member functions
unions can't have static data members

Union cannot have base classes, virtual functions or static data members and they cannot be used as base classes. (E)

C++ **unknown language string in linkage specifier: ...**

Only "C" and "C++" allowed in linkage specifiers. (E)

```
extern "F77" { // Don't know anything about F77 linkage
}
```

unknown option -X*unknown*

The compiler was started with an -X option that is not recognized. (W)

C++ **value out of range, hexadecimal constant too large**
value out of range, octal constant too large
value out of range, decimal constant too large

Constant overflows its type. (E)

```
int i = 4294967299; // Constant bigger than ULONG_MAX
```

variable ... is incomplete

A variable is defined with a type that is incomplete. (E)

```
struct a;
struct a b;
```

variable *name* is used before set

During optimization, D-C++ discovers a variable that is used before it is set. (W)

```
func() { int a; if (a == 0) ... }
```

C++ **virtual specified both before and after access specifier**

Syntax error. (W)

```
class A {};
class B : virtual public virtual A {};
```

C++ **virtual specifier may only be used inside a class declaration**

Function cannot be declared virtual outside class body. (E)

```
struct A {
    foo();
};
virtual A::foo() {} // Not virtual in the class declaration
```

write error

Write to output file failed. (F)

wrong number of arguments

Number of arguments given does not match prototype or function definition. (W) if -Xpcc or -Xk-and-r or -Xmismatch-warning, (E) otherwise.

```
int fn(int, int); ... fn(1,2,3);
```

C++ **wrong number of parameters for template ...**
wrong number of parameters for function template

Too many or too few template parameters supplied. (E)

```
template<class T>
class foo {
    // ...
};

foo<int, double> bar;
```

wrong type of initializer

The type of an initializer is not compatible with the type of the variable. (W) if -Xpcc or -Xmismatch-warning. (E) otherwise.

```
char c; int *ip = &c;
```

PowerPC

Target User's Manual

DIAB DATA

Defining Compiler Performance

Copyright Notice

Disclaimer

Diab Data makes no representations or warranties with respect to the contents of this publication, and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Diab Data reserves the right to revise this publication and make changes from time to time in the content hereof without obligation on the part of Diab Data to notify any person or company of such revision or changes.

In no event shall Diab Data, or others from whom Diab Data has a licensing right, be liable for any indirect, special, incidental, or consequential damages arising out of or connected with a customers possession or use of this product, even if Diab Data or such others has advance notice of the possibility of such damages.

Trademarks

Diab Data, alone and in combination with D-AS, D-C++, D-CC, D-F77, and D-LD are trademarks of Diab Data, Inc. All other trademarks used in this document are the property of their respective owners.

Diab Data, Inc.
323 Vintage Park Drive
Foster City, CA 94404
USA

Tel **415-571-1700**
Fax **415-571-9068**

Email **support@ddi.com**

CONTENTS

List of Tables

1 Introduction

This is the *PowerPC Compiler Target User's Manual* for the **Diab Data Optimizing compilers** for the PowerPC family of RISC microprocessors, including the PowerPC 601, 602, 603, 603e, 604, 505, 821, 860 and the 403.

It is written for the professional programmer and contains detailed target specific information, such as internal data representation and calling conventions, for the Diab Data Optimizing Compilers:

- D-CC The Diab Data Optimizing C Compiler
- D-C++ The Diab Data Optimizing C++ Compiler
- D-F77 The Diab Data Optimizing FORTRAN 77 Compiler

The *User's Manual* for the respective compilers is referred to as the *Language User's Manual* in this manual.

See also the following documentation:

- *D-CC Language User's Manual*
- *D-C++ Language User's Manual*
- *D-F77 Language User's Manual*
- *D-AS Assembler User's Manual*
- *D-LD Linker User's Manual*
- *Utilities User's Manual*
- *C Library Manual*
- *C++ Class Reference Manual*

The following manual from Motorola/IBM may also be of interest:

- *PowerPC Microprocessor Family: The Programming Environments*

2 Target Dependent Command Line Options

This section complements Chapter 2, "Installing the Compiler," and Chapter 3, "Invoking the Compiler," in the *Language User's Manual.* All target-dependent command line options are described here.

Selecting a target

A complete *target configuration* specifies the target processor, the type of floating point support (hardware or software), and the object module format (e.g., ELF or COFF). The preferred methods for selecting a target configuration are to use the `dctrl` command or the -t compiler command line option as described in "Using D-C++ for different targets" in Chapter 2, "Installing the Compiler," of the *Compiler Target User's Manual.*

As noted there, either of these methods implicitly sets three variables maintained in `default.conf` (`default.con` on DOS): `DTARGET` to select the processor, `DOBJECT` to select the object module format, and `DFP` to select the type of floating point support wanted.

The valid values for these three variables are as follows.

Target processors:

DTARGET	Processor
PPC601	PowerPC 601 (-WDDTARGET=PPC601)
PPC602	PowerPC 602 (-WDDTARGET=PPC602)
PPC603	PowerPC 603 (-WDDTARGET=PPC603)
PPC604	PowerPC 604 (-WDDTARGET=PPC604)
PPC403	PowerPC 403 (-WDDTARGET=PPC403) DFP always set to soft
PPC505	PowerPC 505 (-WDDTARGET=PPC505)
PPC821	PowerPC 821 (-WDDTARGET=PPC821) DFP always set to soft
PPC860	PowerPC 860 (-WDDTARGET=PPC860) DFP always set to soft

Target object format and calling conventions:

DOBJECT	Description
D	COFF using AIX conventions (-WDDOBJECT=D)
E	ELF using EABI conventions (-WDDOBJECT=E)

Floating point support:

DFP	Description
hard	Use Hardware Floating Point. This is the default on processors that has a floating point unit.
soft	Use the supplied Software Floating Point emulation functions provided with the compiler. This is the default on processors that lacks an internal floating point unit.

-X options

The following -X options can be given in an environment variable, on the command line, or in a configuration file. See Chapter 4, "Configuration File," in the *Language User's Manual* for details.

Table 2-1 -X Options for PowerPC

X Option	Description
-Xkill-reorder=x -X28=x	Disable individual optimizations in the reorder program. The following optimizations can be disabled. Multiple optimizations can be disabled by OR-ing their values: 0x01 Basic Reordering 0x04 Branch to small code 0x100 Delete NOPs
-Xconventions-eabi -X31=0	Specifies that EABI calling conventions should be used. This option is controlled by DOBJECT=E and should not be set explicitly by the user. See "Argument passing" on page 28.
-Xconventions-aix -X31=2	Specifies that AIX calling conventions should be used. This option is controlled by DOBJECT=D and should not be set explicitly by the user. See"Argument passing" on page 28.
-Xstmw-slow -Xstmw-ok -Xstmw-fast -X32=*n*	Specifies whether the **stmw** and the **lmw** instructions should be used to save/restore registers at function entry/exit. -Xstmw-slow means that they should never be used. -Xstmw-ok means that they can be used in leaf functions. -Xstmw-fast means that they should always be used instead of calling a library function to perform the operation. The -XPPC*x* option sets the optimal value for each processor.

Table 2-1 -X Options for PowerPC *(continued)*

X Option	Description
-Xinline-alloca=0 -X33=0	The non-standard libc function **alloca**(*size*), which allocates dynamic memory on the stack, needs special treatment on a processor with no frame pointer such as the PowerPC. Therefore, **alloca**( ) is recognized by the compiler and usually inlined. To avoid this feature, use the -Xinline-alloca=0 option.
-Xno-alloca -X33=1	Disable recognition and special treatment of **alloca**( ).
-Xinline-alloca -X33=2	Expand calls to **alloca**( ) in line. This is the default on PowerPC.
-Xadd-underscore -X34	Concatenate an underscore before every function identifier. Concatenation of underscore is useful when compiling libraries, to avoid using the same name space as user programs.
-Xstsw-slow -Xstsw-ok -Xstsw-fast -X35=*n*	Specify whether the **stswi** and the **lswi** string instructions should be used for structure assignments, etc. -Xstsw-slow means that they should never be used. -Xstsw-ok means that they can be used for unaligned assignments. -Xstsw-fast means that they can always be used instead of using individual **lwz** and **stw** instructions. The -XPPC*x* option sets the optimal value for each processor.
-Xcrb6-never -Xcrb6-float -Xcrb6-always -X36=*n*	Specify whether **CR** bit 6 should be set or cleared when calling a function without a prototype using EABI conventions. With a prototype, normally **CRB6** is set when calling a function with a variable number of arguments if any argument is a floating type, and cleared if not. Since it is impossible for the compiler to determine if a function without a prototype uses variable arguments, the -Xcrb6-*x* option defines whether the compiler should be pessimistic or optimistic about setting **CRB6**. -Xcrb6-never will never set nor clear **CRB6** for functions without prototypes. -Xcrb6-float will set **CRB6** for functions without prototypes if any floating point argument is used. This is the default. -Xcrb6-always will always set or clear **CRB6** for functions without prototypes.
-Xtrace-table	Generate the trace table needed to do a back-trace on the PowerPC. This option is enabled by default but can be disabled with -Xtrace-table=0 to save space.

Table 2-1 -X Options for PowerPC *(continued)*

X Option	Description
-XPOWER -XPPC601 -XPPC602 -XPPC603 -XPPC604 -XPPC403 -XPPC505 -XPPC821 -X39=*n*	Generate and optimize code for the different PowerPC and POWER architectures. This option is controlled by DTARGET and should usually not be set explicitly by the user. Note: the option -XPPC821 is used for both the PPC821 and the PPC 860.
-Xsoft-float -X56	Specifies that software floating point emulation should be used. The implementation is a very fast call-based method using the calling conventions specified in the EABI. This option is controlled by DFP, which also selects which library to use, and should not usually be set explicitly by the user.
-Xnear-code-relative -X58=1	Generate position independent code for all references to the **.text** section. These include function calls, taking the address of a function, and accessing strings and **const** variables if the -Xstrings-in-text option is used. Branches use the 26-bit PC relative offsets. All other references use a 16 bit offset from register **r2**, which means that this option cannot access code and constant data larger than 64KB.
-Xfar-code-relative -X58=2	Generate position independent code for all references to the **.text** section. These include function calls, taking the address of a function, and accessing strings and **const** variables if the -Xstrings-in-text option is used. Branches use the 26-bit PC relative offsets. All other references use a 32 bit offset from register **r2**.
-Xall-near-code-relative -X58=3	Generate position independent code for references to all variables and functions. Branches use the 26-bit PC relative offsets. All other references use a 16 bit offset from register **r2**, which means that this option cannot access code and data bigger than 64KB.
-Xall-far-code-relative -X58=4	Generate position independent code for references to all variables and functions. Branches use the 26-bit PC relative offsets. References to the Small Constant Area (SCA, called SDA2 in EABI) will use a 16-bit offset from register **r2**. All other references use a 32 bit offset from register **r2**.

Table 2-1 -X Options for PowerPC *(continued)*

X Option	Description
-Xnear-data-relative -X59=1	Generate position independent data references to variables in the **.data** and **.bss** sections. These include all variables, excluding strings and **const** variables if the -Xstrings-in-text option is used. All references use a 16 bit signed offset from register **r13**, which means that this option cannot be used if the total size of the **.data** and **.bss** sections added together is larger than 64KB.
-Xfar-data-relative -X59=2	Generates position independent data references to variables in the **.data** and **.bss** sections. These include all variables, excluding strings and **const** variables if the -Xstrings-in-text option is not disabled. Data in the Small Data Area (SDA) will use a 16-bit offset from register **r13** which means that the SDA is limited to 64KB. All other references use a 32 bit offset from register **r13**.

3 Compiler Components

This section describes the target-dependent components of the Diab Data compilers. See Chapter 2, "Installing the Compiler," of the *Language User's Manual* for an overview of all compiler components.

On UNIX systems the default location of these files is
`/usr/lib/diab/`*version*

On MS-DOS systems the default location of these files is
`C:\DIAB\`*version*

On MPW systems the default location of these files is
`{MPW}diab:`*version*

where *version* is the software version number, e.g., 3.7a.

The PPC directory

This directory contains the following files used by all PowerPC targets.

PPC.cd
The PowerPC compiler description file that is read by the compiler and interpreted during code generation. This file is not normally accessed by users.

PPC.ad
The PowerPC assembler description file that is read by the assembler and interpreted during object file generation. This file is not normally accessed by users.

The target directories

The variables DTARGET, DOBJECT, and DFP are used by the compiler and assembler to select options and files for the compilation. As noted in "Selecting a target" on page 3, these variables are maintained in `default.conf` (`default.con` on DOS) and are normally set implicitly using the `dctrl` program or the -t option on the command line.

A directory name, *target*, is created from these three variables as follows:

target=`PPC`*of*

where:

- *o* is the object format used according to the variables.
- *f* is the floating point mode set by the DFP variable, `H` for hardware floating point (DFP=hard), `S` for software floating point (DFP=soft)

Example:

`PPCEH` --- PowerPC, EABI Mnemonics and Hardware floating point.

These directories contain the following files and libraries specific to each target:

crt0.o
This file contains the startup code. It typically initialize the environment before call-

ing the **main**() function.

libc.a and libm.a

These files are the C object library and the math object library.

libd.a and libcomplex.a

`libd.a` is the C++ library containing the iostream classes and `libcomplex.a` is the complex math library. They use `libc.a` routines and functions from `libm.a`.

4 Implementation Specific Behavior

Predefined macros and assertions

A set of target specific predefined preprocessor macros are defined by the compiler. The macros not starting with two underscores (__) will not be defined if the -Xstrict-ansi option is given:

Macro	Value
ppc	The constant 1.

Pragmas

section
use_section

#pragma section *class_name* [*istring*] [*ustring*] [*addr-mode*] [*acc-mode*] [**address**=*val*]

#pragma use_section *class_name* [*variable* | *function*] ,...

#pragma section defines and controls the section into which variables and code are placed. It also controls how the variables should be accessed (16/32 bit absolute, 16/32 bit code relative, and 16/32 bit data relative). **#pragma use_section** selects the section for specific variables or functions after the section has been defined by **#pragma section**.

All variables and functions are by default categorized into one of six different *classes* depending on how they are defined and how large they are. By using the **#pragma use_section** directive, any variable and function can be individually assigned to any of the six predefined classes or to a user defined class.

Multiple **#pragma section** directives with different parameters can be given for the same section class. Variables and functions will use the directive valid at the point of definition.

class_name is the symbolic name of a section class and is one of

name	**Description**
DATA	Static and global variables: `static int ar[10];`
SDATA	DATA with size <= -Xsmall-data: `int i;`
CONST	Constant DATA using the **const** keyword: `const int arc[10];`
SCONST	CONST with size <= -Xsmall-const: `const int ic;`
STRING	String constants: `"hello\n"`
CODE	Code generated in functions and global asm statements: `int inc(int i) { return i+1; }`
user	User defined section

istring is an optional quoted string giving a name for a particular section of the given class which is to contain **initialized** data. The name is used in the assembly **.section** directive to switch to an appropriate section for initialized data. An empty string or no string at all indicates that the default value should be used.

Examples: `".text", ".data", ".init"`

ustring is an optional quoted string giving a name for a particular section of the given class which is to contain **uninitialized** data. The name is used in the assembly **.section** directive to switch to an appropriate section for uninitialized data. An empty string or no string at all indicates that the default value should be used. The string "COMM" indicates that the **.comm/.lcomm** assembler directives should be used and the variable should be allocated in the uninitialized data section **.bss**.

Examples: `".bss", ".data", "COMM"`

Default values for *istring/ustring* (na = not applicable):

section	Default values for *istring/ustring*
DATA	**.data/COMM**
SDATA	**.sdata/.sbss**
CONST	**.text/.text**
SCONST	**.sdata2/.sdata2**
STRING	**.text**/na
CODE	**.text**/na
user	**.data/COMM**

If -Xstrings-in-text=0 then **CONST** and **STRING** are put in **.data** instead of **.text** and **SCONST** is put in **.sdata** instead of **.sdata2**.

addr-mode is the addressing mode to be used when referencing the variable/function/string. It is one of the values given in the following table. Notes follow the table.

Table 4-1 *addr-mode* definitions

addr-mode	Number	Description
standard	0x01	processor specific standard method; see below
near-absolute	0x10	absolute addressing, 16 bits
far-absolute	0x11	absolute addressing, 32 bits
near-data	0x20	data relative addressing, 16 bits
far-data	0x21	data relative addressing, 32 bits
near-code	0x40	code relative addressing, 16 bits
far-code	0x41	code relative addressing, 32 bits

Notes:

- The hexadecimal number is used for command line options described below.
- The *addr-mode* **standard** means that a processor specific method is being used, usually defined by an ABI standard. Currently the **CODE** section is the only section where this has a special meaning:
  - branches can be either PC-relative or absolute
  - function pointers are absolute
- When using PowerPC COFF, only the **standard** and **far-absolute** addressing modes are implemented.
- Position Independent Code (PIC) can be achieved by using the code relative addressing modes.
- Position Independent Data (PID) can be achieved by using the data relative addressing modes.

The code-relative and data-relative addressing modes compute memory addresses by adding an offset to a base register. Absolute addressing uses an offset from address 0. With a 16 bit offset, a 64 KB area can be accessed. With a 32 bit offset, the full 4 GB memory area can be accessed. The following base registers are used for the addressing modes:

absolute	**r0**
code-relative	**r2** for data references, **PC** for branches
data-relative	**r13**

The default *addr-mode* values are:

DATA	far-absolute
SDATA	near-data
CONST	far-absolute
SCONST	near-code
STRING	far-absolute
CODE	standard
user	far-absolute

The following options change the default *addr-mode*:

- -Xaddr-data=*mode*
- -Xaddr-sdata=*mode*
- -Xaddr-const=*mode*
- -Xaddr-sconst=*mode*
- -Xaddr-string=*mode*
- -Xaddr-code=*mode*
- -Xaddr-user=*mode*

These options direct that the named section, **DATA**, **SDATA**, etc., be addressed with the given addressing mode. *mode* is a hexadecimal number as given in Table 4-1, "addr-mode definitions," on page 12.

Example:

```
-Xaddr-data=0x20
```

address variables in **DATA** with near-data relative addressing

The following table describes other command line options that will affect the default *addr-mode*:

Option	Sets	
-Xnear-code-relative	-Xaddr-const=0x40 -Xaddr-string=0x40	-Xaddr-sconst=0x40 -Xaddr-code=0x40
-Xfar-code-relative	-Xaddr-const=0x41 -Xaddr-string=0x41	-Xaddr-sconst=0x40 -Xaddr-code=0x41
-Xnear-data-relative	-Xaddr-data=0x20 -Xaddr-user=0x20	-Xaddr-sdata=0x20
-Xfar-data-relative	-Xaddr-data=0x21 -Xaddr-user=0x21	-Xaddr-sdata=0x20
-Xall-near-code-relative	-Xaddr-const=0x40 -Xaddr-string=0x40 -Xaddr-data=0x40 -Xaddr-user=0x40	-Xaddr-sconst=0x40 -Xaddr-code=0x40 -Xaddr-sdata=0x40
-Xall-far-code-relative	-Xaddr-const=0x41 -Xaddr-string=0x41 -Xaddr-data=0x41 -Xaddr-user=0x41	-Xaddr-sconst=0x41 -Xaddr-code=0x41 -Xaddr-sdata=0x41
-Xstrings-in-text=0	-Xaddr-sconst=0x20	

➤ The -Xfar-code-relative and -Xfar-data-relative options still use 16-bit offsets for data in the Small Const Area (SCA, called SDA2 in EABI) and Small Data Area (SDA) respectively.

acc-mode defines how the section can be accessed and is any combination of:

R	Read permission
W	Write permission
X	Execute permission

The default *acc-mode* values are:

DATA	**RW**
SDATA	**RW**

CONST	**R**
SCONST	**R**
STRING	**R**
CODE	**RX**
user	**RW**

If -Xstrings-in-text=0 then sections **CONST**, **SCONST**, and **STRING** have the value **RW**.

Sections with data that have no write access (**W**) can share the same memory for multiple identifiers and strings.

address=*val* provides a way to place variables and functions at a specific absolute address in memory. This is accomplished by putting absolute code and data in sections named **.abs.***XXXXXXXX*, where *XXXXXXXX* is the hexadecimal address given as *val*. The ELF linker will automatically place these sections at the given address. Note that any section name given in an absolute **#pragma section** will be ignored.

Advantages of using absolute sections:

- I/O registers, global system variables, and vector functions can be placed at the correct address from the C/C++ program without the need to write a complex Link Command script.
- Absolute variables will have all symbolic information needed by a debugger to reference them. Variables defined in the Link Command language can not be debugged at a high level.

Examples:

```
// define IOSECT:
// a user defined section containing I/O registers

#pragma section IOSECT near-absolute RW address=0xffffff00
#pragma use_section IOSECT ioreg1, ioreg2

// place ioreg1 at 0xffffff00 and ioreg2 at 0xffffff04
int ioreg1, ioreg2;

// Put an interrupt function at address 0x700
#pragma interrupt programException
#pragma section ProgSect RX address=0x700
#pragma use_section ProgSect programException

void programException() {
  // ...
}
```

Usage

There are two ways to put a variable or function in a specific section.

The examples in this section use the C language. Corresponding constructs in C++ and FORTRAN will produce the same effects.

- A variable or function can be placed in a specific section by redefining the default section into which the variable or function would normally be placed.

  Examples:

  - `ar1` is placed in the default **DATA** section (**.data**) using far-absolute addressing:

    ```
    int ar1[100] = { 0 };
    ```

  - `ar2` is placed in section `.abs` using near-absolute addressing:

    ```
    #pragma section DATA ".section .abs" near-absolute
    int ar2[100] = { 0 };
    ```

  - `ar3` is again placed in the default **DATA** section:

    ```
    #pragma section DATA
    int ar3[100] = { 0 };
    ```

- By specifying a specific section in a `#pragma use_section`.

  Example:

  ```
  #pragma section VECTOR "section .absdata" \
          "section .absdata" near-absolute RW
  #pragma use_section VECTOR ar4
  int ar4[100];
  ```

  `ar4` is placed in section `.absdata` using 16-bit absolute addressing.

Implementation

The compiler will generate the following assembly code for the different *addr-mode* settings. The corresponding code, using C as an example, is:

```
reg = var;      /* var in DATA or SDATA */
func();         /* func in CODE         */
```

Mode	Instructions

standard	`addis r3,r0,var@ha`
	`lwz r3,var@l(r3)`
	`bl func`

near-absolute	`lwz r3,var(r0)` `bla func`

far-absolute	`addis r3,r0,var@ha`
	`lwz r3,var@l(r3)`
	`addis r3,r0,func@ha`
	`addi r3,r3,func@l`
	`mtspr 9,r3`
	`bcctrl 20,0`

Mode	Instructions

near-data	`lwz r3,var@sdarx(r0)`
	`addi r3,r0,func@sdarx`
	`mtspr 9,r3`
	`bcctrl 20,0`

far-data	`addis r3,r0,var@sdarx@ha`
	`lwz r3,var@sdax@l(r3)`
	`addis r3,r0,func@sdarx@ha`
	`addi r3,r3,func@sdax@l`
	`mtspr 9,r3`
	`bcctrl 20,0`

near-code	`lwz r3,var@sdarx(r0)` `bl func`

far-code	`addis r3,r0,var@sdarx@ha`
	`lwz r3,var@sdax@l(r3)`
	`bl func`

Notes:

- The assembler uses the following special PowerPC relocation types for the above @ operators:

`@sdarx`	R_PPC_EMB_SDA21
`@sdax`	R_PPC_EMB_RELSDA
`@sdarx@ha`	R_PPC_DIAB_SDA21_HA
`@sdarx@hi`	R_PPC_DIAB_SDA21_HI
`@sdarx@l`	R_PPC_DIAB_SDA21_LO
`@sdax@ha`	R_PPC_DIAB_RELSDA_HA
`@sdax@hi`	R_PPC_DIAB_RELSDA_HI
`@sdax@l`	R_PPC_DIAB_RELSDA_LO

- The `@sdarx` relocation types for the PowerPC will actually use different base pointers depending on which section the variable is defined in: **r0** for absolute addressing, **r2** for code relative addressing, and **r13** for data relative addressing. The linker will patch the instruction to use the correct register depending on where the variable is defined. Register **r2** is initialized to **_SDA2_BASE_**. Register **r13** is initialized to **_SDA_BASE_**. Both labels are defined by the linker.

Additions to ANSI C

long long

The Diab compilers support 64 bit integers for all PowerPC processors. A variable declared **long long** or **unsigned long long** is an 8 byte integer. To specify a **long long** constant, use the **LL** or **ULL** suffixes:

```
long long mask_nibbles (long long x)
{
    return (x & 0xf0f0f0f0f0f0f0f0LL);
}
```

5 Internal Data Representation

This section describes the alignments, sizes, and ranges of the C, C++, and FORTRAN data types for the PowerPC.

Basic data types

The following table describes the basic data types available in the compiler. All sizes and alignments are given in bytes. An alignment of 2 means that data of this type must be allocated on an address divisible by 2:

Data type	Size	Align	Notes
char	1	1	range (0, 255) (note 1)
			(-128, 127) with -Xsigned-char
signed char	1	1	range (-128, 127)
unsigned char	1	1	range (0, 255)
short	2	2	range (-32768, 32767)
unsigned short	2	2	range (0, 65535)
int	4	4	range (-2147483648, 2147483647)
unsigned int	4	4	range (0, 4294967295)
long	4	4	range (-2147483648, 2147483647)
unsigned long	4	4	range (0, 4294967295)
long long	8	8	range (-2^63, 2^63-1)
unsigned long long	8	8	range (0, 2^64-1)
enum	4	4	same as **int** (note 2)
	1	1	with -Xenum-is-small and fits in **signed char** or -Xenum-is-best and fits in **unsigned char**
	2	2	with -Xenum-is-small and fits in **short** or -Xenum-is-best and fits in **unsigned short**
pointers	4	4	all pointer types; the **NULL** pointer has the value zero
float	4	4	IEEE 754-1985 single precision
double	8	8	IEEE 754-1985 double precision
long double	8	8	IEEE 754-1985 double precision
reference	4	4	C++: same as pointer (note 3)
ptr-to-member	8	4	C++: pointer to member
ptr-to-member-fn	12	4	C++: pointer to member function

Notes:

1. If the option -Xunsigned-char is specified, the plain **char** type is unsigned. If the option -Xsigned-char is specified, the plain **char** type is signed.
2. If the option -Xenum-is-small or -Xenum-is-best is specified in C, **enums** take smallest type possible, **char**, **short**, **int** or **long** (signed or unsigned as required due to range).

   In C++ programs **enums** always take the smallest type possible, **char**, **short**, **int** or **long** (signed or unsigned as required due to range). The option -Xenum-is-best is always forced.
3. A reference is implemented as a pointer to the variable to which it is initialized.

Classes, structures, and unions

The alignment of class, structure, and union aggregates is the same as that of the member with the largest alignment.

The size of a structure is the sum of the size of all its members plus any necessary padding. Padding is added so that all members are aligned to a boundary given by their alignment and to make sure that the total size of the structure is divisible by its alignment.

The size of a union is the size of its largest member plus any padding necessary to make the total size divisible by the alignment.

To minimize the necessary padding, structure members can be declared in descending order by alignment.

See the **#pragma pack** and the **packed** keyword in the *Language User's Manual* for more information.

C++ classes

C++ objects of type class, struct, or union can be divided into two groups, aggregates and non-aggregates. An aggregate is a **class, struct,** or **union** with no constructors, no private or protected members, no base classes, and no virtual functions. All other classes are non-aggregates.

The internal data representation for aggregates is exactly the same as it is for C structures and unions.

Static member functions and static class members, as well as non-virtual member functions do not affect the representation of classes. Their relation to the classes are only encoded in their names (name mangling). Pointers to static member functions and static class members are ordinary pointers. Pointers to member functions are of the type *pointer-to-member-function* as described later.

The internal data representation for non-aggregates has the following properties:

- The rules for alignment are equal to the rules of aggregates.
- The order that members appear in the object is the same as the order in the declaration.

- Non-virtual base classes are inserted before any members, in the order that they are declared.
- A pointer to the virtual function table is added after the bases and members.
- For virtual base classes a pointer to the base class is added after any non-virtual bases, members, or the virtual function table. The virtual base class pointers are added in the order that they are declared.
- The storage for the virtual bases are placed last in the object, in the order they are declared, that is, depth first, left to right.
- Virtual base classes that declare virtual functions are preceded by a "magic" integer used during construction and destruction of objects of the class.

Example:

```
struct V1 {};
struct V2 {};
struct V3 : virtual V2 {};
struct B1 : virtual V1 {};
struct B2 : virtual V3 {};
struct D : B1, private virtual V2, protected B2 {
   int d1;
private:
   int d2;
public:
   virtual ~D() {};
   int d3;
};
```

The class hierarchy for this example is:

D is derived from B1, B1 is derived from V1
D is derived from B2, B2 is derived from V3, V3 is derived from V2
D is derived from V2 (which is virtual, thus there is only one copy of V2)

The internal data representation for D will be as follows:

B1
B2
Body of D: d1 d2 d3
Virtual function table pointer
Pointer to virtual base class V1
Pointer to virtual base class V2
Pointer to virtual base class V3
V1
V2

magic for V3
V3

Please note:

- When the class D is used as a base class to another class, for example:

```
class E : D {};
```

  only the base part of D will be inserted before the body of class E. The virtual bases V1, V2, and V3 will be placed last in class E, in the fashion described above. Class E would be laid out as follows:

Base part of D
Body of E: ...
V1
V2
magic for V3
V3

- The virtual function table pointer is only added to the first base class that declares virtual functions. A derived class will use the virtual function table pointer of its base classes when possible. A virtual function table will be added to a derived class when new virtual functions are declared, and none of its non-virtual base classes has a virtual function table.
- The virtual function table is an array of pointers to functions. The virtual function table has one entry per virtual function, plus one entry for the *null* pointer.
- Virtual base class pointers are added to a derived class when none of its non-virtual base classes have a virtual base class pointer for the corresponding virtual base class.
- Each virtual base class with virtual functions are preceded by an integer called *magic*. This integer is used when virtual functions are called during construction and destruction of objects of the class.

Pointers to members

The pointer-to-member type (non-static) is represented by two objects. One for pointers to member functions, and one for all other pointers to member types. The offsets below are relative to the class instance origin.

An object for a pointer to non-virtual or virtual member functions has three parts:

voffset
index
vtbl-offset or Function Pointer

The *voffset* field is an integer that is used when the virtual function table is located in a virtual base class. In this case it contains the offset to the virtual base class pointer + 1. Otherwise it has a value of 0.

The *index* field is an integer with two meanings.

1. *index* <= 0
   The *index* field is a negative offset to the base class in which the non-virtual function is declared. The third field is used as a function pointer.
2. *index* > 0
   The *index* field is an index in the virtual function table. The third field, *vtbl-offset*, is used as an offset to the virtual function table pointer of type integer.

A *null* pointer-to-member function has zero for the second and third fields.

An object for a pointer-to-member of a non-function type has two parts:

voffset
moffset

The *voffset* field is used in the same way as for pointer-to-member functions. The *moffset* field is an integer that is the offset to the actual member + 1. A *null* pointer to member has zero for the *moffset* field.

Arrays

Arrays have the same alignment as their element type. The size of any array is equal to the size of the data type multiplied by the number of elements.

Bit fields

Bit fields can be of type **char**, **short**, **int**, **long**, or **enum**. Plain bit fields are unsigned by default. By using the -Xsigned-bitfields option or by using the **signed** keyword, bit fields become signed. The following rules apply to bit fields:

1. Allocation is from most significant bit to least.
2. A bit field never crosses its type boundary. Thus a **char** bit field is never allocated across a byte boundary and can never be wider than 8 bits.
3. Bit fields are allocated as closely as possible to the previous **struct** member without violating rule 2.
4. A zero-length bit field pads the structure to the next boundary specified by its type.

Byte ordering

All data is normally stored in big-endian order. That is, with the most significant byte of any multi-byte type at the lowest address. To access data in little-endian order, see the *byte-swapped* parameter for the **#pragma pack** and the **packed** keyword in the *Language User's Manual.*

Linkage and storage allocation

Depending on whether a definition or declaration is performed inside or outside the scope of a function, different storage classes are allowed and have slightly different meanings. Notes are at the end of the section.

Outside any function and outside any class:

Specifier	Linkage	Allocation
none	external linkage, program	static allocation (note 1)
static	file linkage	static allocation (note 1)
extern	external linkage, program	none, if the object is not initialized in the current file, otherwise same as without specifier

Inside a function, but outside any class:

Specifier	Linkage	Allocation
none	current block	in a register or on the stack (note 2)
register	current block	in a register or on the stack (note 2)
auto	current block	on the stack
static	current block	static allocation (note 1)
extern	current block	none, this is not a definition (note 3)

Outside any function, but inside a C++ class definition:

Outside the class a class member name must be qualified with the :: **operator, the** . **operator or the** -> operator to be accessed. The private, protected, and public keywords, class inheritance and friend declaration will affect the access rights.

Specifier	Linkage	Allocation
none (data)	external linkage, program	none, this is only a declaration of the member. Allocation depends on how the object is defined.
static (data)	external linkage, program	none, this is not a definition. A static member must be defined outside the class definition
none (function)	external linkage, program	(uses a **this** pointer)
static (function)	external linkage, program	(no **this** pointer)

Within a local C++ class, inside a function:

A local class can not have static data members. The class is local to the current block as described above and access to its members is through the class. All member functions will have internal linkage.

Notes:

1. Static allocations are per the following table:

Definition	Initialized	Uninitialized
Variables defined with the **const** keyword and which are smaller than the value given with the -Xsmall-const=*n* option. The default value for *n* is 8.	**.sdata2**	n/a
Other variables defined with the **const** keyword, i.e., **const** variables which are not "small".	**.text**	n/a
Variables defined without the **const** keyword and which are smaller than the value given with the -Xsmall-data=*n* option. The default value for *n* is 8.	**.sdata**	**.sbss**
Other variables, i.e., variables not using **const** and which are not "small".	**.data**	**.bss**
Functions	**.text**	n/a

2. The compiler attempts to assign as many variables as possible to registers, with variables declared with the **register** keyword having priority. Variables which have their address taken are allocated to the stack. If the -Xlocals-on-stack option is given, only **register** variables are allocated to registers.

3. Although an **extern** variable has a local scope, an error will be given if it is redefined with a different storage class in a different scope.

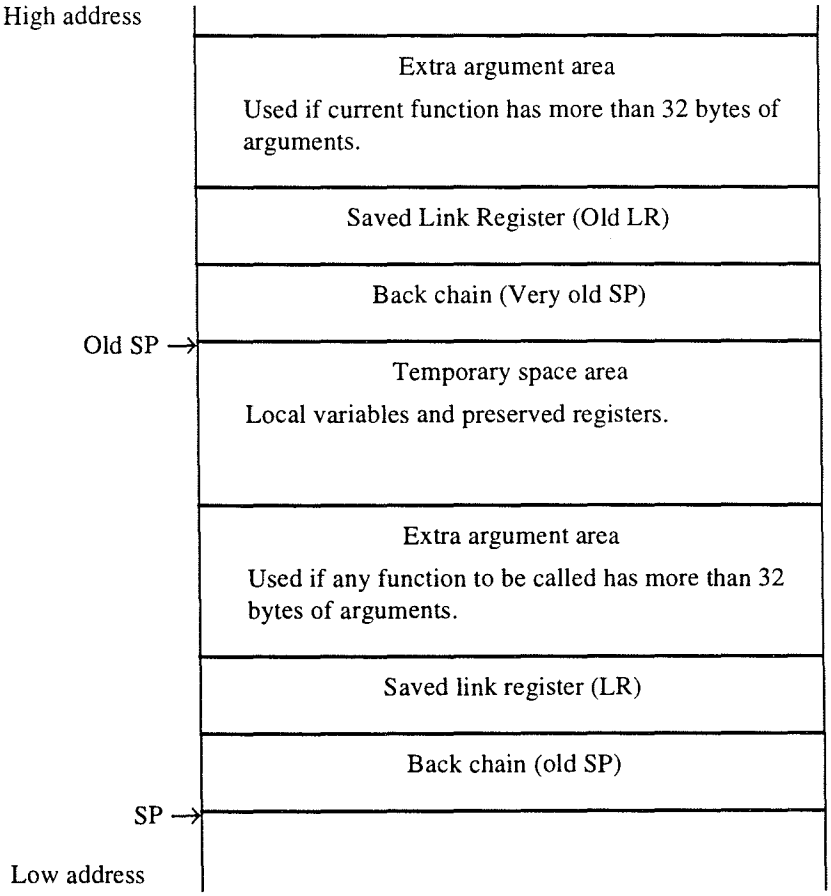
6 Calling Conventions

This section describes the interface between a function caller and the called function.

Stack layout

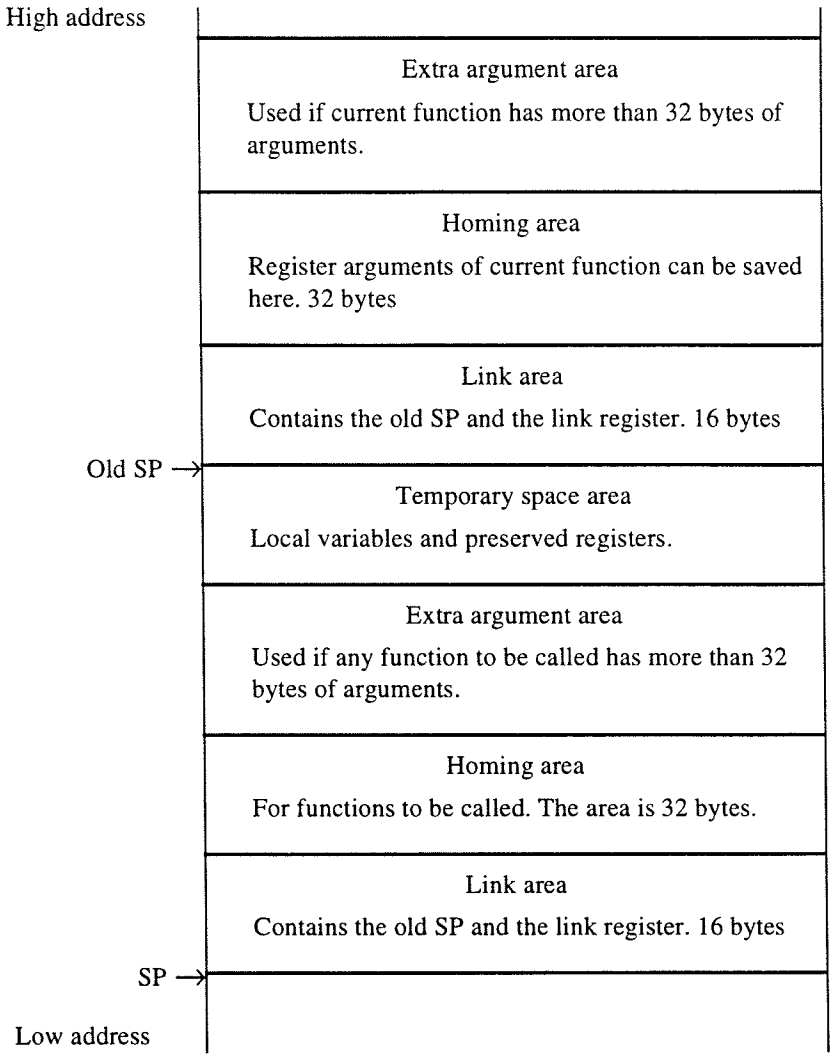
PowerPC**EABI** Stack Frame

The following figure shows the stack frame when compiling in EABI mode:
SP = Stack Pointer (**r1**)



PowerOpen Stack Frame

In COFF mode, the stack layout on the PowerPC follows the PowerOpen Standard.
SP = Stack pointer.



Note that the EABI stack frame uses considerably less stack space because it does not reserve the 32 byte homing area needed in the PowerOpen stack frame. The homing area is used to save the argument registers if the function is a variable argument (**stdarg/varargs**) function. The EABI assumes that this is not a common case and trades this space with a somewhat more complex **varargs** scheme.

Leaf routines that do not use any stack space do not create a stack area at all. The SP register (r1) always points to a back chain containing the previous stack pointer.

Argument passing

EABI Argument Passing

In EABI mode, the following algorithm is used to pass arguments:

Definitions:

- **GREG** is a general purpose register number

- **FREG** is a floating-point register number
- **OFFSET** is the offset from the stack pointer at which to place the next non-register argument
- **NEXT** is the next argument
- **GTYPE** is a type that can be passed in a single **GREG**:
  - integers of size <= 32 bits
  - pointers
  - all structures, unions, and classes since they are passed as a pointer to the object, or as a pointer to a copy of the object if the aggregate is non-constant
  - single precision floating point values when compiling with software floating-point (DFP=soft)
- **FTYPE** is a floating-point type that can be passed in an **FREG**:
  - double precision values when using hardware floating-point (DFP=hard)
  - single precision values when hardware floating-point is used. They are expanded to double precision
- **LTYPE** is a type that can be passed in two **GREG**s:
  - 64 bit integers (long long)
  - double precision values when software floating-point is used

1. Set **GREG**=3, **FREG**=1, **OFFSET**=8
2. If **NEXT** is a **GTYPE**:
   - if **GREG** > 10 goto step 5
   - put argument in **GREG**
   - **GREG** += 1
   - goto step 2
3. If **NEXT** is an **FTYPE**:
   - if **FREG** > 10 goto step 5
   - put argument in **FREG**
   - **FREG** += 1
   - goto step 2
4. If **NEXT** is an **LTYPE**:
   - if **GREG** > 9 goto step 5
   - if **GREG** is even: **GREG** += 1
   - put argument in **GREG** and **GREG**+1
   - **GREG** += 2

- goto step 2

5. Put **NEXT** on stack:
   - align **OFFSET** so it's a multiple of the alignment of **NEXT**
   - put argument on stack at **SP+OFFSET**
   - **OFFSET** += sizeof(**NEXT**)
   - goto step 2

A caller to a function using a variable argument list (`stdarg.h` or `varargs.h`) also sets condition register bit 6 to 1 if it passes arguments in one or more floating-point registers. Otherwise **CRB6** is cleared. The varargs function then checks **CRB6** to see if it needs to store the floating-point registers in memory.

Since the compiler cannot determine whether functions with no prototype use a variable argument list, there are three compiler options that define whether **CRB6** should be set or not in this case:

- -Xcrb6-never will never set nor clear **CRB6** for functions without prototypes.
- -Xcrb6-float will set **CRB6** for functions without prototypes if any floating point argument is used. This is the default.
- -Xcrb6-always will always set or clear CRB6 for functions without prototypes.

PowerOpen Argument Passing

In PowerOpen mode, all arguments to be submitted to a function have a corresponding offset from the stack pointer. All arguments are aligned on 4. Characters and shorts are extended to 32 bits. The first argument has the offset 24 and subsequent arguments are allocated offsets consecutively. Any argument in the homing area (the first 32 bytes) is passed in general register numbered 3+(offset/4). The first 13 floating point arguments are passed in **f1**:**f13**. If the callee has a prototype without ellipsis, floating point arguments are only passed in float registers, otherwise they are passed in both.

Note that if a C function uses floating point arguments, it is much more efficient, in PowerOpen mode, to use a prototype, since the arguments do not need to be copied to the general purpose registers,

C++ argument passing

In C++, the same lower level conventions are used as in C, with the following additions:

- References are passed as pointers.
- Function names are encoded (mangled) with the types of all arguments. A member function has also the class name encoded in its name. See 6.8 "C++ Name Mangling".
- An argument of **class**, **struct**, or **union** type will be passed as a pointer to the object, unless the function is declared as a C function. Example: A call to a C++ function

```
int ff(struct S s);
```

from a C function should be made like:

```
struct S xyz;
int i = ffmangledname(&xyz);
```

Note that the function name is usually mangled, and this example can't be taken literally.

Pointer to member as arguments and return types

Pointers to members are internally converted to structures. Therefore argument passing and returning of pointer to members will follow the rules of **class**, **struct**, and **union**.

Member function

Non static member functions have an extra argument for the **this** pointer. This argument is passed as a pointer to the class in which the function is declared. The argument is passed as the first argument, unless the function returns an object that needs the hidden return argument pointer, in which case the return argument pointer is the first argument and the **this** pointer is the second argument.

Constructors and destructors

Constructors and destructors are treated like any other member function, with some minor exceptions as follows.

Constructors for objects with one or more virtual base classes have one extra argument added for each virtual base class. These arguments are added just after the **this** pointer argument. The extra arguments are pointers to their respective base classes.

Calling a constructor with the virtual base class pointers equal to the *null* pointer indicates that the virtual base classes are not yet constructed. Calling a constructor with the virtual base class pointers pointing to their respective virtual bases indicates that they are already constructed.

All destructors have one extra integer argument added, after the **this** pointer. This integer is used as a bit mask to control the behavior of the destructor. The definition of each bit is as follows (bit 0 is the least significant bit of the extra integer argument):

Bit 0 — When this bit is set the destructor will call the destructor of all sub-objects except for virtual base classes. Otherwise the destructor will call the destructor for all sub-objects.

Bit 1 — When this bit is set the destructor will call the operator **delete** for the object.

All other bits are reserved and should be cleared.

Result passing

Characters and shorts are extended to 32 bits and returned in register **r3**. Integers and pointers are returned in register **r3**. Floating point values are returned in register **f1** with hardware floating point. With software floating point, **float** values are returned in **r3**, **double** values in **r3/r4**, and **long long** values in **r3/r4**. All other types are returned in the memory area pointed to by the hidden argument passed in register **r3**. See the -Xstruct-as-args option for more information.

In C++, a hidden argument is used when the return type is a **class**, **struct** or **union**. See the next subsection for details.

Class return types

A function with a return type of **class**, struct, or **union** will be called with a hidden argument of type pointer to function return type. The called function will copy the return argument to the object pointed out by the hidden argument. Example: Calling the C++ function

```
struct S gg(int);
```

from a C function should be done in the following manner:

```
struct S retval;
ggmangledname(&retval, 74);
```

Note that the function name is usually mangled, and this example can't be taken literally.

Register usage

The following table describes how registers are used by the compiler. The floating point register file (**f0** - **f31**) is utilized only when hardware floating point is used.

r0	Scratch register.
r1	Stack pointer.
r2	Pointer to the Small Constant Area (SCA, called SDA2 in EABI). Small constant values are usually put here and can be accessed with a single instruction. The linker initializes the symbol _SDA2_BASE and the startup code in `crt0.o` loads register **r2** with this value. Reserved register in PowerOpen (COFF).
r3 - **r12**	Temporary registers. Not preserved by functions. Hold variables whenever possible. **r3** - **r10** are also used for parameter and result passing.
r13	Pointer to the Small Data Area (SDA) in EABI. Small variables are usually put here and can be accessed with a single instruction. The linker initializes the symbol _SDA_BASE and the startup code in `crt0.o` loads register **r13** with this value. Reserved register in COFF mode.
r14 - **r31**	Preserved registers. Saved when used by functions. Hold variables which cannot be put in **r3** - **r12**.
f0	Scratch register.
f1 - **f13**	Temporary floating point registers. Not preserved by functions. Hold variables whenever possible. **f1** - **f8** are also used to pass parameters and results
f14 - **f31**	Preserved floating point registers. Saved when used by functions. Hold variables which cannot be put in **f1** - **f13**.

7 Optimizations

In general, optimizations have two purposes: to improve the speed and to minimize the size of the compiled program.

Most of the optimizations are activated by the -O option. Others, such as inlining, need the -XO option to be activated. See Chapter 6, "Optimization Hints," in the *Language User's Manual* to fully utilize the optimizations described below.

Target independent optimizations

The following optimizations are performed by the compiler. To skip any single optimization, use the -Xkill-opt=x switch. The value of x is shown below in parentheses.

Tail recursion (0x2)

This optimization replaces calls to the current function, if located at the end of the function, with a branch.

Example:

```
NODEP find(NODEP ptr, int value)
{
  if (ptr == NULL) return NULL;
  if (value < ptr->val) {
    ptr = find(ptr->left,value);
  } else if (value > ptr->val) {
    ptr = find(ptr->right,value);
  }
  return ptr;
}
```

will be approximately translated to:

```
NODEP find(NODEP ptr, int value)
{
top:
  if (ptr == NULL) return NULL;
  if (value < ptr->val) {
    ptr = ptr->left;
    goto top;
  } else if (value > ptr->val) {
    ptr = ptr->right;
    goto top;
  }
  return ptr;
}
```

Inlining (0x4)

Inlining optimization replaces calls to small functions with the actual code from the same functions to avoid call-overhead and generate more opportunities for further optimizations. Inlining can be triggered in three ways:

1. In C++ use the **inline** keyword when defining the function, and in C use the **__inline__** keyword.
2. Use the **#pragma inline** *function-name* directive. The **#pragma** directive can be used in C++ code to avoid the local linkage forced by the C++ **inline** keyword.
3. Use the **-Xinline** (or -XO).

Both caller and callee must be in the same file in order for inlining to work.

Example:

```
#pragma inline swap
swap(int *p1, int *p2)
{
  int tmp;
  tmp = *p1;
  *p1 = *p2;
  *p2 = tmp;
}

func() {
  ...
  swap(&i,&j);
  ...
}
```

will be translated to:

```
func() {
  ...
  {
    tmp = i;
    i = j;
    j = tmp;
  }
...
}
```

Argument address optimization (0x8)

If the address of a local variable is used only when passing it to a function which does not store that address, the variable can be allocated to a register and only temporarily placed on the stack during the call to the function.

Example:

```
func() {                    ->      func () {
  ...                                 ...
  swap(&i,&j);                        itmp = i; jtmp = j;
  ...                                 swap(&itmp, &jtmp);
}                                     i = itmp; j = jtmp;
                                      ...
                                    }
```

`i` and `j` can now be placed in registers.

Structure members to registers (0x10)

This optimization places members of local structures and unions in registers whenever it is possible.

Local strength reduction (0x20)

Expression trees are rewritten to trees that will execute faster. Constant expressions are also evaluated. This optimization is active even if -O is not specified.

Example:

```
a*4                     ->    a<<2
0==5 ? a : b            ->    b
i+i+i+i                 ->    i<<2
```

Question - expression pop (0x40)

Expressions using the **?:** operator are rewritten to if-statements to be available to flow-control optimizations:

Example:

```
d = a ? b : c           ->    if (a) d = b; else d = c
```

Assignment pop (0x80)

Expressions with embedded assignments are rewritten:

```
k(a = b)                ->    a = b; k(a)
```

Simple branch optimization (0x100)

All **for**, **do**, and **while** loops as well as **if**, **break**, **continue**, and **return** statements are rewritten internally as conditional and unconditional branches to labels. This optimization finds branches to branches, conditional branches around unconditional branches, and so forth and makes them optimal.

Space optimization (0x200)

Different paths with equal tails are rewritten:

```
if (a) {                ->    if (a) {
  b(); c();                       b();
} else {                        } else {
  d(); c();                       d();
}                               }
                                c();
```

This optimization is most effective when many **case** statements end the same way.

Split optimization (0x400)

Variables with more than one live range are rewritten to make it possible to allocate them to different registers/stack locations:

```
m(int i, int j) {       ->    m(int i$1, int j) {
  int k = f(i,j);               int k = f(i$1,j);
  i = f(k,j);                   i$2 = f(k,j);
  return i+k;                   return i$2+k;
}                               }
```

In the above example, only two registers are needed to hold the three variables after split optimization, since `i$1` and `k` can share one register and `i$2` and `j` can share the other one.

Constant and variable propagation (0x800)

Constants and variables assigned to a variable are propagated to later references of that variable. Lifetime analysis might later remove the variable:

```
a = 1; b = 2;          ->     a = 1; b = 2;
...; k(a+b);                  ...; k(1+2);
```

Complex branch optimization (0x1000)

Branches and fall-throughs to conditional branches where the outcome can be computed are rewritten. This typically occurs after a loop with multiple exits.

Example:

```
fl = 0;
while(!fl) {
  stm1();
  if (e1) break;
  if (e2) {
    fl = 1;
  }
}
if (fl) {
  stm2();
}
```

This will be rewritten, here shown with goto's, to the following code. Note that the variable `fl` is removed by the life-analysis:

```
l1:
  stm1();
  if (e1) goto l2;  /* skip stm2 since fl is 0 */
if (!e2) goto l1;   /* if !e2 fall through    */
stm2();
l2:
```

Loop strength reduction (0x2000)

Multiplications with constants in loops are rewritten to use additions. Instead of multiplying `i` with the size every time, the size is added to a pointer (`arp++` in the example below). The array reference

```
ar[i]
```

is actually treated as

```
*(ar_type *)((char *)ar + i*sizeof(ar[0]))
```

Example:

```
for(i=0;i<10;i++){     ->     arp = ar;
  sum += ar[i];                for(i=0;i<10;i++){
}                                sum += *arp; arp++;
                               }
```

Loop count-down optimization (0x4000)

Loop variable increments are reversed to decrement towards zero:

```
for(i=0;i<10;i++){      ->      for(i=10;i>0;i--){
  sum += *arp; arp++;             sum += *arp; arp++;
}                                 }
```

Loop unrolling (0x8000)

Small loops are unrolled to reduce the loop overhead and increase opportunities for rescheduling. The -Xunroll option sets the number of times the loop should be unrolled. The -Xunroll-size defines the maximum size of loops allowed to be unrolled. Example:

```
for(i=10;i>0;i--){      ->      for(i=10;i>0;i-=2){
  sum += *arp;                    sum += *arp;
  arp++;                          sum += *(arp+1);
                                  arp += 2;
}                                 }
```

Global common subexpression elimination (0x10000)

Subexpressions, once computed, are held in registers and not re-computed the next time the subexpressions occur. Memory references are also held in registers.

```
if (p->op == A)         ->      tmp = p->op;
  ...                           if (tmp == A)
else if (p->op == B)              ...
                                else if (tmp == B)
```

Undefined variable propagation (0x20000)

Expressions containing undefined variables are removed.

Unused assignment deletion (0x40000)

Assignments to variables that are not used are removed.

Minor transformations (0x80000)

Some minor transformations are performed to ease recognition in the code generator:

```
if (a) return 1;        ->      return a ? 1 : 0;
return 0;
```

Delayed register saving (0x100000)

Preserved registers that must be saved on the stack are stored when needed instead of at function start.

Register coloring (0x200000)

This optimization locates variables that can share the same register.

Interprocedural optimizations (0x400000)

Registers are allocated across functions. Inlining and argument address optimizations are also performed.

Remove entry and exit code (0x800000)

The code at the beginning and end of a function which handles the stack-frame is removed whenever possible:

```
add(int a, int b) {
  return a+b;
}
```

```
      mfspr  r0,8          ->
      stw    r0,8(r1)
      stwu   r1,-64(r1)
      add    r3,r3,r4                    add   r3,r3,r4
      lwz    r0,72(r1)                   blr
      addi   r1,r1,64
      mtspr  8,r0
      blr
```

Use scratch registers for variables (0x1000000)

When allocating registers the compiler attempts to put as many variables as possible in scratch registers (registers not preserved by the function).

On the PowerPC, the scratch count register (**CTR**) is used as a loop index for tight, fast loops. The compiler will be especially efficient in using **CTR** when the -XO option is used.

The following example will show the use of **CTR**, as well as some other PowerPC optimizations:

```
float dotproduct(float *x, float *y, int n)
{
  int m;
  float sum = 0.00;
  for(m = 0; m < n; m++)
    sum +=  x[m]*y[m];
  return(sum);
}
```

The following assembly code shows the inner loop produced by the compiler:

```
.L4:
      lfsu    f13,4(r3)
      lfsu    f12,4(r4)
      lfsu    f11,4(r3)
      fmadds  f1,f13,f12,f1
      lfsu    f10,4(r4)
      fmadds  f1,f11,f10,f1
      bc      16,0,.L4
```

Notes:

- The loop is unrolled twice and will do two multiplies and additions every iteration.

- Strength reduction replaces the array indexing with pointers (**r3** and **r4**) to the arrays. The pointers are pre-incremented with **lfsu** instructions on every memory access and have been initialized to point one element before the arrays to compensate for the pre-increment.
- The **fmadds** instruction is utilized to perform a multiply and an add in the same instruction.
- The Control Register has replaced the index variable `m` for zero cycle loop control. Note that the **CTR** is counting down towards zero (the `bc 16` instruction decrements the **CTR** and jumps if not zero) and has been initialized with *n*/2. An extra conditional iteration (not shown) has been added to cover the case when *n* contains an odd value.

Extend optimization (0x2000000)

Sometimes the compiler must generate many **extend** instructions to extend smaller integers to a larger one. The compiler attempts to avoid this by changing the type of the variable. For example:

```
int c;
char *s;
c = *s;
if (c == 2) c = 0;
```

On some targets, the "`c = *s`" statement has an extend instruction. By changing "`int c`" to "`char c`" this instruction is avoided.

Loop statics optimization (0x4000000)

Memory references that are updated inside loops are allocated to registers.

Example:

```
int ar[100], sum;

sum_ar() {
  int i;

  sum = 0;
  for(i = 0; i < 100; i++) {
    sum += ar[i];
  }
}
```

will be translated to:

```
sum_ar() {
  int i;
  register int tmp_sum

  tmp_sum = 0;
  for(i = 0; i < 100; i++) {
    tmp_sum += ar[i];
  }
  sum = tmp_sum;
}
```

Loop invariant code motion (0x8000000)

Expressions within loops that are not changed between iterations are moved outside the loop.

Example:

```
for(i = 0; i < 100; i++) {
  sum += a*b;
}
```

will be translated to:

```
tmp = a*b;
for(i = 0; i < 100; i++) {
  sum += tmp;
}
```

Static function optimization (0x20000000)

A static function that does not have its address taken can be optimized in various ways, including:

- If the function is not used, it can be removed.

PowerPCFeedback optimization (-Xfeedback)

By providing the compiler with profiling information from an actual execution of the target program, the optimizer can make more intelligent decisions in various cases, including the following:

- Register allocation can be based on the real number of times a variable is used.
- **if-else** clauses are swapped if first part is executed more often.
- Inlining and loop unrolling is not done on code seldom executed.
- More inlining and loop unrolling is done on code often executed.
- Partial inlining is done on functions beginning with `if (e) return;`
- Branch prediction is performed.

See Chapter 6, "Optimization Hints," in the *Language User's Manual* for more information on how to use the -Xfeedback option.

Target dependent optimizations

The following optimizations are specific to the PowerPC family and performed by the `reorder` program.

They can be turned off with the -Xkill-reorder=x option.

PowerPCBasic reordering (0x1)

Instructions are reorganized to avoid stalls in the processor pipeline. For example, when loading a value from memory (**lwz r3,0(r4)**) on the PowerPC 601, the processor has to stall for one cycle if the next instruction uses the destination register. The compiler rearranges the code so the processor can execute at full speed:

```
lwz   r3,0(r4)      ->          lwz   r3,0(r4)
addi  r3,r3,4                   stw   r24,40(r1)
stw   r24,40(r1)                stw   r25,44(r1)
stw   r25,44(r1)                addi  r3,r3,4
```

Branch to small code (0x4)

A branch to a few instructions followed by another branch is rewritten by inlining these instructions at the current address:

```
        b     .L2          ->          addi  r31,r31,32
        ...                            blr
.L2:                              .L2:
        addi  r31,r31,32               addi  r31,r31,32
        blr   r1                       blr
```

Delete no-ops (0x200)

Sometimes the code generator emits instructions that do not change any register. These instructions are removed:

```
        addi  r3,r0,0x10   ->          addi  r3,r0,10
        addis r3,r3,0
```

8 Use in an Embedded Environment

Introduction

There are some significant differences between developing software for an embedded system compared to a native environment, since there is sometimes no operating system support for:

- initialization of data
- initialization of `argc`, `argv`, and environment variables
- hardware exception handling (illegal memory access, divide by zero, etc.)
- file and device I/O
- memory allocation
- signal handling
- execution of instructions to enable caches
- virtual memory

Other features needed in an embedded environment include:

- complete control over allocation of code and data to specific addresses
- placement of initialized data in ROM and its movement on startup to RAM
- packed structures to map external hardware or data from other processors
- mixing of big- and little-endian data structures

Compiler options for embedded development

The following compile-time options and pragmas control code generation in various ways:

-Xsize-opt	minimize the size of the executable code.
-Xstrings-in-text	put strings and **const** data in the **.text** section together with code.
-Xstruct-max-align -Xstruct-min-align	options to pack structures in different ways.
-Xmemory-is-volatile	treat all memory references as volatile, to avoid optimizing away accesses to hardware ports. This option is not needed if the **volatile** keyword is used for variables making accesses to volatile data.
-Xstack-probe	insert code to check that the stack does not grow outside its boundaries.
-Xdollar-in-ident	allow variable names containing '$'-signs.
-Xaddr-*x*	control addressing modes for data and code

-Xsmall-data -Xsmall-const	specify what data is to go into the Small Data Area (SDA) and the Small Const Area (SCA, called SDA2 in EABI)
-X*x*-code-relative -X*x*-data-relative	enable position independent code and data (PIC and PID)
#pragma interrupt *func*	specifies that a function *func* is an exception handler.
#pragma section ...	control placement and addressing of variables and functions
#pragma pack	control packing of structures and the byte order of members

See in the *Language User's Manual* and earlier sections in this manual for more information on the command line options and for details on the **#pragma** directive.

User modifications

Since most embedded environments are unique, some things which must be modified by the user:

- start-up code must initialize the processor in various ways
- hardware exceptions must be handled
- a linker command file must specify where to allocate code and data
- any operating system call used (for example by library functions) must be redefined

Start-up code

The file `crt0.s` supplied with the compiler has a sample start-up function. Please refer to the manuals for the target processor for more information on how to initialize the processor.

The last thing `crt0.s` does is jump to the **__init_main()** function in the `init.c` file, which does the rest of the initialization in C. This includes copying initialized data from ROM to RAM, clearing uninitialized data, setting up `argc`, `argv`, and environment variables, and calling all functions that should be executed before the **main()** function, e.g., global constructors in C++. Arguments and environment variables can be specified with the setup program described later.

`crt0.s` is assembled with the command:

```
dcc -o crt0.s
```

Either replace the standard `crt0.o` file or use the following option to use a new start-up module:

```
dcc -Wspathname ... other parameters ...
```

where *pathname* is the location and name of the replacement `crt0.o` file. This option can be added to the `user.conf` configuration file to make it permanent.

Global initialization

Global initialization code is code that is to be executed before the **main**() function, e.g., global constructors in C++. Global destruction code is code that is to be executed after the **exit**() function has been called, e.g., global destructors in C++.

While compiling each module, the compiler generates special code to call all initialization and destruction functions. This special code is placed in the **.init** section for initialization code and in the **.fini** section for destruction code. That is, each module will make a contribution to the **.init** section and to the **.fini** section.

These sections, and a final **.eini** section which ends the program, are defined in the startup module (`crt0.s` in the src directory) as follows:

```
            section .init
__init:
            initialize the stack

            section .fini
            clear the stack
            return from __init to __init_main()

__fini:
            initialize the stack

            section .eini
            clear the stack
            return from __fini to exit()
```

The linker will concatenate the contributions to the **.init** and **.fini** sections from each module in the order in which the modules are presented to the linker. Thus, `crt0.o` must be the first module presented to the linker so that the result in memory will be:

```
            section .init
__init:
            initialize the stack
            contribution to .init from module 2
            contribution to .init from module 3
            etc.

            section .fini
            clear the stack
            return from __init to __init_main()

__fini:
            initialize the stack
            contribution to .fini from module 2
            contribution to .fini from module 3
            etc.

            section .eini
            clear the stack
            return from __fini to exit()
```

The **.fini** section ***must*** follow immediately after the **.init** section, because it contains the return instruction for the **__init**() function. Likewise, the **.eini** section must immediately follow the **.fini** section because it contains the return instruction for the __fini() function.

The **__init**() function is called from **__init_main**() in the file `init.c` in the src directory, and **__fini**() is called from **exit**().

An older method of doing initialization and destruction using the **__CTORS** and the **__DTORS** arrays are now obsolete and that code can be removed from the link command files.

Hardware exception handling

Please read the target processor's user's manual for a description of the exception (interrupt) vector.

The compiler provides the following support for interrupt routines:

- a **#pragma interrupt** which specifies that a function is an exception handler
- the library function **raise**( ), which can be called with an appropriate signal from the interrupt routine to raise a signal
- a **#pragma section** directive that can place exception vectors at an absolute address

Linker command file

By specifying a link editor command file, the user can:

- specify input files and options
- specify how to combine the input sections into output sections
- specify how memory is configured
- assign addresses to symbols

4. The sample command file, `sample.lnk` - supplied in the source directory, appears next.

Figure 8-1 Sample Linker command file

```
/* ------------------------------------------------------------------------
 * This is a sample Link Editor Command Language file specifying how an
 * embedded application should be linked and located. Combined with the
 * other Link Editor Command Files provided with the compiler, it
 * provides a base to build your own file for your particular system.
 *
 * For a description of the different statements in this file, please
 * refer to the D-LD Linker User's Manual.
 * ------------------------------------------------------------------------ */


/* The following block defines the different memory areas available:
 *   1MB ROM at address 0x0
 *   1MB RAM at address 0x100000
 *   1MB Gap (unused)
 *   1MB RAM at address 0x300000 used for stack
 * ------------------------------------------------------------------------ */
MEMORY
{
        rom:    org = 0x0, len = 0x100000
        ram:    org = 0x100000, len = 0x100000
        stack:  org = 0x300000, len = 0x100000
}


/* This block specifies where and how the linker should locate different
 * modules of the system.
 *
 * This example will allocate according to the following map:

   0x0:          +--------------------------------+
   "rom"         | Program code(1)                |
                 | (2)                            |
                 +--------------------------------+ <- __DATA_ROM
                 | ROM Image of initialized data  |
                 | (3)                            |
                 +--------------------------------+
                 | (Unused portion of "rom")      |
   0x100000:     +--------------------------------+ <- __DATA_RAM
   "ram"         | Memory reserved for            |
                 | initialized data               |
                 +--------------------------------+ <- __DATA_END, __BSS_START
                 | Uninitialized data             |
                 |                                |
                 +--------------------------------+ <- __BSS_END, __HEAP_START
                 | Memory reserved for the heap   |
                 | (all unused "ram")             |
                 +--------------------------------+ <- __HEAP_END (3)
                 /                                /
                 / 1MB Gap -- Not used            /
                 /                                /
   0x300000:     +--------------------------------+ <- __SP_END (3)
   "stack"       | Memory reserved for the stack  |
```

Figure 8-1 Sample Linker command file *(continued)*

```
                | (all of the "stack")          |
  0x4000000:    +-------------------------------+ <- __SP_INIT

 * In the margin are the locations of the different identifiers that are
 * used by some library routines to handle memory initialization and
 * allocation. They are defined further below.
 *
 * NOTES:
 * (1)  Constants and strings will also be in the .text segment unless
 *      the -Xstrings-in-text=0 option is used.
 *
 * (2)  If C++ code is to be linked then code calling the static
 *      contructors and destructors will be placed in the .init and
 *      the .fini sections allocated after the program code.
 *
 * (3)  If __SP_END and __HEAP_END point to the same address (i.e., the
 *      "ram" and "stack" memory areas are contiguous), then programs
 *      compiled with -Xstack-probe will allocate more stack from the
 *      top of the heap when stack overflow occurs, if possible (this is
 *      done by the __sp_grow() function in sbrk.c).
 * ------------------------------------------------------------------------ */
SECTIONS
{
        /* The first section is allocated into the "rom" area. */

        GROUP : {
                /* First take all code from all objects and libraries */

                .text (TEXT)    : {
                        *(.text) *(.rodata) *(.init) *(.fini) *(.eini)
                        . = (.+15) & ~15;
                }

                /* Next take all small CONST data */
                .sdata2 (TEXT) : {}
        } > rom

        /* The second section will allocate space for the initialized data
         * (.data/.sdata) and the uninitialized data (.bss/.sbss) in the
         * "ram" section.
         *
         * Initialized data is actually put at the end of the .text section
         * with the LOAD command. The function __init_main() moves the
         * initialized data from ROM to RAM.
         */
        GROUP : {
                /* This will reserve space for the .data in the beginning
                 * of "ram" but actually place the image at the end of
                 * .text segment
                 */
                .data (DATA) LOAD(ADDR(.sdata2)+SIZEOF(.sdata2)) : {}

                /* .sdata contains small address data */
```

Figure 8-1 Sample Linker command file *(continued)*

```
                .sdata (DATA) LOAD(ADDR(.sdata2)+SIZEOF(.sdata2)
                                                +SIZEOF(.data)) : {}

                /* This will allocate the .bss symbols */
                .sbss (BSS)     : {}
                .bss  (BSS)     : {}

                /* Any space left over will be used as a heap */
        } > ram
}

/* Definitions of identifiers used by sbrk.c, init.c and the different
 * crt0.s files. Their purpose is to control initialization and memory
 * allocation.
 *
 * __HEAP_START : Used by sbrk.c. Start of memory used by malloc() etc.
 * __HEAP_END   : Used by sbrk.c. End of heap memory
 * __SP_INIT    : Used by crt0.s. Initial address of stack pointer
 * __SP_END     : Used by sbrk.c. Only used when stack probing
 * __DATA_ROM   : Used by init.c. Address of initialized data in ROM
 * __DATA_RAM   : Used by init.c. Address of initialized data in RAM
 * __DATA_END   : Used by init.c. End of allocated initialized data
 * __BSS_START  : Used by init.c. Start of uninitialized data
 * __BSS_END    : Used by init.c. End of data to be cleared
 * ------------------------------------------------------------------------ */

__HEAP_START    = ADDR(.bss)+SIZEOF(.bss);
__SP_INIT       = ADDR(stack)+SIZEOF(stack);
__HEAP_END      = ADDR(ram)+SIZEOF(ram);
__SP_END        = ADDR(stack);
__DATA_ROM      = ADDR(.sdata2)+SIZEOF(.sdata2);
__DATA_RAM      = ADDR(.data);
__DATA_END      = ADDR(.sdata)+SIZEOF(.sdata);
__BSS_START     = ADDR(.sbss);
__BSS_END       = ADDR(.bss)+SIZEOF(.bss);

/* Some targets use an extra underscore in front of identifiers
 * ------------------------------------------------------------------------ */
___HEAP_START   = __HEAP_START;
___HEAP_END     = __HEAP_END;
___SP_INIT      = __SP_INIT;
___SP_END       = __SP_END;
___DATA_ROM     = __DATA_ROM;
___DATA_RAM     = __DATA_RAM;
___DATA_END     = __DATA_END;
___BSS_START    = __BSS_START;
___BSS_END      = __BSS_END;
```

Other link files written for some specific targets are also provided in the source directory.

See the chapter Linker Command Language in the *Linker User's Manual* for more information about the command language. The link editor command file (map file) is specified with the -Wm compiler option:

`-Wm`*path_name*

where *path_name* is the full name of the map file. To use the same map file for all compilations, specify this option in the `user.conf` configuration file.

Operating system calls

The source files available in the `src` directory implement the following POSIX/UNIX functions for an embedded environment:

access	**getpid**	**open**	**unlink**
clock	**isatty**	**read**	**write**
close	**kill**	**sbrk**	
creat	**link**	**signal**	
_exit	**lseek**	**time**	

These are the system routines which implement all ANSI library functions plus most of the other non-system functions in D-LIBC.

To use these functions, the following must be done:

- Modify the files discussed below.
- Compile the files. The script `compile` can be used to do this.
- Copy the `libc.a` archive to the directory with the object files.
- Replace the object files in the `libc.a` archive. The script `replace` can be used to do this.
- Either replace the standard `libc.a` file or use the following option to use the modified library:

  `-Wc`*pathname*

  where *pathname* is the location and name of the new `libc.a` file. This option can be added to the `user.conf` configuration file to make it permanent.

Character I/O

The predefined files `stdin`, `stdout`, and `stderr` use the **__inchar()/__outchar()** functions in `chario.c`. These functions can be modified in order to read/write to a serial interface on the user's target. The files `/dev/tty` and `/dev/lp` are also pre-defined and mapped to these character I/O functions.

File I/O

The file I/O functions are implemented as a RAM disk. Space is allocated by calls to **malloc()**. The following system calls are supported:

access( ) in `access.c`, checks if a file is accessible.

close( ) in `close.c`, closes a file.

creat( ) in `creat.c`, opens a new file by calling **open( )**.

fcntl( ) in `fcntl.c`, checks the type of a file

fstat( )	in `stat.c`, gets some information about a file
isatty( )	in `isatty.c`, checks whether a file is connected to an interactive terminal. It is used by the `stdio` functions to decide how a file should be buffered. If it is a terminal, the stream will be flushed at every end-of-line, otherwise the stream will be buffered and written in large blocks.
link( )	in `link.c`, causes two file names to point to the same file.
lseek( )	in `lseek.c`, positions the file pointer in a file
open( )	in `open.c`, opens a new or existing file
read( )	in `read.c`, reads a buffer from a file
stat( )	in `stat.c`, gets some information about a file
unlink( )	in `unlink.c`, removes a file from the file system
write( )	in `write.c`, writes a buffer to a file

Pre-defined files can be created by using the program `setup.c`. This program creates the file `memfile.c` which includes the files given to setup as arguments. Example:

```
setup -t /etc/passwd -t input.bak
```

creates the pre-defined files '/etc/passwd' and 'input.bak' with the contents of these files.

Dynamic memory allocation

The function **sbrk()** handles dynamic memory allocation and is called from the **malloc()** routines. If the macro **SBRK_SIZE** is defined, then **sbrk()** will allocate a fixed size buffer. Otherwise the identifiers **__HEAP_START** and **__HEAP_END** must be defined, typically in the link command file. See the file `sample.lnk` for an example.

Miscellaneous functions

The following functions provide miscellaneous services.

_exit()	in `_exit.c`, closes all open files and then loops for ever. Modify this code if you want to return control to a monitor or operating system.
clock()	in `clock.c`, is an ANSI C function returning the number of clock ticks elapsed since program startup. It is not used by any other library function.
getpid()	in `getpid.c`, returns a process number. Modify this if you have a multiprocessor system.
__init_main()	in `init.c`, is called from the start-up code and performs some initializations.
kill()	in `kill.c`, sends a signal to a process. Only signals to the current process are supported.
signal()	in `signal.c`, changes the way a signal is handled.
time()	in `time.c`, returns the system time. Other functions in the library expect this to be the number of seconds elapsed since 00:00 January 1st 1970.

Stack checking

PowerPCIf the -Xstack-probe option is used when compiling, the compiler inserts code to check if the stack overflows. The __sp_grow() function is defined in sbrk.c. The identifier __SP_END must be defined in the link command file for this checking to work. See the file sample.lnk for an example.

Position independent code (PIC)

By using the link editor command language, it is easy to have complete control over where in memory different sections of the program should be allocated. However in some cases there is no way of knowing where a program will reside in memory until load time. For example:

- In a multi-process environment without virtual memory, new programs are loaded wherever there is unallocated space.
- When more than one process executes the same code section, but uses different data sections. In this case only the data has to be position independent.

In general there are two ways to provide load-time allocation:

- By patching the code with the correct address while loading. The -r and -r2 options to the linker keep the relocation data in the file and can be used by the loader to change all memory references. See the *Linker User's Manual* for details about the -r options and relocation information.
- By generating position independent code (PIC) which can be executed from any address. The compiler will only use addressing modes that are relative to either the current address or a reserved register.

There are two types of PIC: data position independence, where the data can be placed anywhere in the memory, and code position independence, where the code can be placed anywhere. When generating PIC, the compiler can handle both. See the information on **#pragma section** on page 11 for more information.

PowerPCFor the PowerPC family, the following options provide position independence:

- The -Xnear-code-relative and the -Xfar-code-relative options achieve code position independence by only using pc-relative branches and by using **r2**-relative addressing modes when accessing addresses in the code section, such as function addresses and references to strings and **const** data if the -Xstrings-in-text option is used. The -Xnear-code-relative option can be used safely only if the code section is less than 64KB.
- The -Xnear-data-relative and the -Xfar-data-relative options achieve data position independence by using register **r13** as a pointer to the data section, and referencing all data through **r13**. The -Xnear-data-relative option can be used only if the data section is less than 64KB.
- The -Xall-near-code-relative and the -Xall-far-code-relative options achieve code and data position independence by only using pc-relative branches and by using **r2**-relative addressing modes when accessing all data. The -Xall-near-code-relative option can be used safely only if the size of the code and data sections together is less than or equal to 64KB total.

Example:

The following command generates totally position independent code.

```
dcc -Xfar-code-relative -Xfar-data-relative -O c.c
```

➤ Note that even when the -Xfar-code-relative and the -Xfar-data-relative options are used, data in the Small Data Area will be addressed efficiently with 16-bit offsets.

Restrictions for position independent code

It is not possible to statically initialize data with the address of a variable, function, or string when generating PIC:

```
int i = 1;          /* always ok              */
int *p = &i;        /* does not work with PIC */
char *s = "abc";    /* does not work with PIC*/
```

Initializations like this will be flagged by the compiler as a warning or an error if either of the -Xstatic-addr-warning or the -Xstatic-addr-error options are used.

Even without -Xstatic-addr-warning, the compiler will give these warnings if any of the PIC options are used.

Communicating with the hardware

The following features facilitate access to the hardware in an embedded environment.

Mixing C and assembler functions

Since the calling conventions of the compiler are well defined, it is fairly simple to call C functions from assembler and vice versa. Chapter 6, "Calling Conventions," beginning on page 27 for details.

Note that the compiler sometimes prepends and/or appends an underscore character to all identifiers. Use the -S option to examine how this works.

In C++, the extern "C" declaration can be used to avoid name mangled function names for functions to be called from assembler.

Using assembler code from within C programs

Use the **asm** keyword to intermix assembler instructions in a the compiler function. See the section titled "asm and __asm" in Chapter 5, "Additions to ANSI C and C++," in the *Language User's Manual* for details.

Accessing variables and functions at specific addresses

There are four ways to access variables and functions that are located at an absolute address:

1. At compile time by using the **#pragma section** directive to specify that a variable should be placed at an absolute address. See "Pragmas" on page 11.

   Advantages of using absolute sections:

- I/O registers, global system variables, and vector functions can be placed at the correct address from the C/C++ program without the need to write a complex Link Command script.
- Absolute variables will have all symbolic information needed by symbolic debuggers. Variables defined in the Link Command language can not be debugged at a high level.

Examples:

```
// define IOSECT:
// a user defined section containing I/O registers

#pragma section IOSECT near-absolute RW address=0xffffff00
#pragma use_section IOSECT ioreg1, ioreg2

// place ioreg1 at 0xffffff00 and ioreg2 at 0xffffff04
int ioreg1, ioreg2;

// Put an interrupt function at address 0x700
#pragma interrupt programException
#pragma section ProgSect RX address=0x700
#pragma use_section ProgSect programException

void programException() {
  // ...
}
```

2. At compile time by using a macro. For example:

```
/* variable at address 0x100 */
#define mem_port (*(volatile int *)0x100)

/* function at address 0x200 */
#define mem_func (*(int (*)())0x200)

mem_port = mem_port + mem_func();
```

3. At link time by defining the address of an identifier. For example:

   In the C file:

```
extern volatile int mem_port; /* variable */
extern int mem_func(); /* function */

mem_port = mem_port + mem_func();
```

   In the link command file add:

```
_mem_port = 0x100;/* Both with and without '_' */
mem_port = 0x100;

_mem_func = 0x200;
mem_func = 0x200;
```

Note the use of the **volatile** keyword to specify that all accesses to this memory must be executed in the correct order, without the optimizer eliminating any of the accesses.

4. At compile and link time by placing the variables and functions in a special section.

See the *Linker User's Manual* for more information about using the Link Editor Command Language.

Re-entrant library functions

In general the library functions are re-entrant, although in some cases this is impossible because the functions are by definition not re-entrant. The re-entrant functions are marked in the *C Library Manuall*.

Command line arguments and environment variables

When running a program in an embedded environment the program is usually not started from a command shell in an operating system. This implies that command line arguments (`argc`, `argv`) and environment variables do not exist. Typically, no file system exists either.

Sometimes it is very useful to be able to pre-define arguments, variables, and files:

- When porting an existing program (e.g., a test program or benchmark program) the program can be compiled and run as-is, without modifications.
- When running a program which requires large amounts of input data, it is practical to put that data in a file.

The program setup.c initializes arguments, variables, and files. Compile the file `setup.c` with a native compiler and run it with the following arguments:

```
setup [-b file] [-t file] [-a arg] [-e evar] ...
```

where the options are:

-b *file* is a binary file which will be saved as a RAM file
-t *file* is a text file which will be saved as a RAM file
-a *arg* is an argument which will be passed to **main**
-e *evar* is an environment variable which can be accessed with **getenv**()

Any combination and number of the different options are allowed.

Example:

```
setup -t /etc/passwd -b db.dat -t f1.asc
      -a -f -a db.dat -e TERM=vt100
```

This means that the ASCII files `/etc/passwd` and `f1.asc` are pre-defined and can be opened with an **fopen**() or **open**() library call.

The binary file `db.dat` is pre-defined. Note that there is no difference between ASCII and binary files on a UNIX system.

The environment variable TERM is set to 'vt100'.

The program will be started as if it was issued with the command line:

```
prog -f db.dat
```

Profiling in an embedded environment

Profiling enables the user to determine where a program spends most of its execution time. It can also be used by the compiler to do more aggressive optimizations. See the -Xblock-count option, the -Xfeedback option, and the `D-BCNT` program in the *Utilities User's Manual* for more information.

When profiling in an embedded environment one must decide where to put the profiling data. If the RAM-disk file I/O, described in the operating system calls section above, is used, the profile data must be transferred back to the host system on program termination.

The profiler code saves the information in a file called `dbcnt.out`. In the **_exit()** function (in the `_exit.c` source file) code must be inserted which reads that file and transfers it to the host. See the `_exit.c` source for an example of how to do this.

If the profiling information is transfered back to the host in ASCII format, the **ddump -B** command can be used to convert it back to a binary file.

Support for multiple object formats

The compiler internally uses ELF or COFF object formats. The `D-DUMP` program converts COFF format to Motorola S-Record and IEEE 695 formats. Please see the *Utilities User's Manual* for more information.

9 Example of Optimizations

The following C program demonstrates several of the optimizations available in the compiler and how they interact with each other. The target processor is the PowerPC 601.

The optimizations shown are:

(1) Remove entry and exit code
(2) Use scratch registers for variables
(3) Unused assignment deletion
(4) Complex branch optimization
(5) Peephole optimization
(6) Loop strength reduction
(7) Loop count-down optimization
(8) Global common subexpression elimination
(9) Inlining of functions
(10) Constant and variable propagation
(11) Basic reordering optimization

`bubble.c` implements sorts an array in ascending order.

```
swap2(int *ip) /* swap two ints */
{
  int tmp = ip[0];
  ip[0] = ip[1];
  ip[1] = tmp;
}

/* "bubble" sorts the array pointed to by "base", containing
   "count" elements, and returns the number of tests done */

int bubble(int *base, int count)
{
  int change = 1;
  int i;
  int test_count = 0;

  while (change) {
    change = 0;
    count--;
    for (i = 0; i < count; i++) {
      test_count++;
      if (base[i] > base[i+1]) {
        swap2(&base[i]);
        change = 1;
      }
    }
  }
  return test_count;
}
```

When `bubble.c` is compiled with the line

```
dcc -XO -S bubble.c
```

the file `bubble.s` is generated.

Comments have been added below to explain the optimizations performed. *The numbers in parentheses refer to the table of optimizations above.*

```
        text
        .globl bubble
bubble:                             Start of function bubble. No entry code is
                                    necessary (1) since all variables are put in
                                    scratch registers (2) and the link register LR
                                    is not used. The stack pointer is not used and
                                    does not need to be adjusted.
                                    The assignment change = 1 is eliminated
                                    (3) since it is only used in the first while
                                    test, which is known to be true and removed
                                    (4).
        addi    r6,r0,0             test_count = 0;
.L4:                                Top of while (change) loop.
        addic.  r4,r4,-1            count--; Sets condition codes (5)
        addi    r5,r0,0             change = 0;
        bc      4,1,.L16            Top test of for loop. Loop strength reduc-
                                    tion (6) has replaced all references to
                                    base[i] with a created pointer, $$2, placed
                                    in register r10. Since no more references are
                                    made to i, Loop count-down optimization
                                    (7) decrements i from count to 0.
        addi    r10,r3,0            Temporary pointer $$2 is set to base.
        addi    r9,r4,0             i is set to count.
.L8:                                Top label of for loop.
        lwz     r7,0(r10)           $$2[0] is loaded to r7. Since this value is
                                    used later on, it is remembered in $$4 (8).
        lwz     r8,4(r10)           $$2[1] is remembered in $$3 (8).
        addi    r6,r6,1             test_count++ is moved here where the
                                    CPU would otherwise be stalled while wait-
                                    ing for the cache to bring in r8 (11).
        cmp     0,0,r7,r8           if. See if a swap must take place.
        bc      4,1,.L7             If not branch to .L7. The function swap2 is
                                    inlined (9). Variable propagation (10)
                                    removes the use of variables tmp and ip.
        stw     r8,0(r10)           ip[0] = ip[1] (= $$3);
        stw     r7,4(r10)           ip[1] = tmp (= $$4);
        addi    r5,r0,1             change = 1;
.L7:
        addic.  r9,r9,-1            i is decremented (7).
        addi    r10,r10,4           $$2++ (6)
        bc      4,2,.L8             i tested with 0 (5). Bottom of for.
```

```
        cmpi    0,0,r5,0
        bc      4,2,.L4         Bottom test of while (change).
.L16:                           Now return without the need for any
                                exit code (1)
        addi    r3,r6,0         Put the return value in r3.
        blr
# Allocations for bubble        Variable allocations are always given
        .data                   in comments to ease debugging.
#       r3      base            Function parameters are kept in their
#       r4      count           original registers (2).
#       r5      change          Other variables are put in scratch
#       r6      test_count      registers to minimize entry/exit code.
#       r9      i
#       not allocated $$1
#       r10     $$2             Loop strength reduction (6) variable.
#       r8      $$3             Global common subexpression elimination
                                (8) for variable base[i+1].
#       r7      $$4             Global common subexpression elimination
                                (8) for variable base[i].
#       not allocated tmp       Variables deleted by Variable propagation
                                (10).
#       not allocated ip
```

D-AS/PowerPC

Assembler User's Manual

DIAB DATA

Defining Compiler Performance

Copyright Notice

Disclaimer

Diab Data makes no representations or warranties with respect to the contents of this publication, and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Diab Data reserves the right to revise this publication and make changes from time to time in the content hereof without obligation on the part of Diab Data to notify any person or company of such revision or changes.

In no event shall Diab Data, or others from whom Diab Data has a licensing right, be liable for any indirect, special, incidental, or consequential damages arising out of or connected with a customers possession or use of this product, even if Diab Data or such others has advance notice of the possibility of such damages.

Trademarks

Diab Data, alone and in combination with D-AS, D-C++, D-CC, D-F77, and D-LD are trademarks of Diab Data, Inc. All other trademarks used in this document are the property of their respective owners.

Diab Data, Inc.
323 Vintage Park Drive
Foster City, CA 94404
USA

Tel **415-571-1700**
Fax **415-571-9068**

Email **support@ddi.com**

CONTENTS

List of Tables

1 Introduction

This document describes the D-AS assembler for the PowerPC microprocessor and should be utilized by assembly language programmers developing assembly programs for the PowerPC.

D-AS is part of Diab Data's suite of software tools for the professional developer. These include the D-CC C and C++ compiler, the D-LD Linker, and the D-AR Archiver, as well as front-ends for additional languages and back-ends for different target processors. Please see the corresponding manuals for information on these products.

D-AS is installed with a Diab Data compiler and other tools. See the *Language User's Manual* for installation instructions.

For in-depth information on the PowerPC architecture, please refer to Motorola/IBM documentation.

Document conventions

This manual uses the following typographic conventions:

Table 1-1 Document Conventions

Example	Description
`dcc -o test.c`	This font is used for file and program names, environment variables, examples, user input, and program output.
if, **main( )**, **#pragma**, **__pack__**	Bold type is used for keywords, operators and other tokens of the language, library routines and entry points, and section names.
	Some names begin or end with underscores. These underscores and special characters such as # shown in bold are required.
variable, *filename*	Italic type is used for placeholders for information which you must supply. Italics are also used for emphasis, to introduce new terms, and for titles.
[optional text]	An item enclosed in brackets is optional.
{ item1 | item2 }	Two or more items enclosed in braces and separated by vertical bars means that you *must* choose exactly one of the items.
item ... item ,...	An item followed by "..." means that items of that form may be repeated separated by whitespace (spaces or tabs). A character preceding the "..." means that the items are separated by the character, shown here as a comma, and optional whitespace.
	The item may be a single token, an optional item enclosed in [ ] brackets (meaning that the item may appear not at all, once, or multiple times), or a set of choices enclosed in { } braces (meaning that a choice must be made from the enclosed items one or more times).

2 Invoking the Assembler

Environment Variables

For information on setting environment variables and using different versions of the assembler, see Chapter 2 of the *Language User's Manual.*

The following environment variables are used by D-AS for selecting appropriate target tables.

Variable	Description
DTARGET	Specifies the target processor. See the *Compiler Target User's Manual* for the default and what options are available.
DOBJECT	Specifies the target object and mnemonic type. See the *Compiler Target User's Manual* for the default and what options are available.
DFP	Specifies whether to use software or hardware floating point. DFP is either set to the value "hard" or "soft". This value is ignored on DTARGETs that only support software floating point.

The das command

The command to execute D-AS is as follows:

```
das [options] [input-file]
```

where:

`das`	Invokes the assembler.
options	Command line options. See the following subsection for details. Options and *input-file* may occur in any order.
input-file	The name of a file to be assembled. The default suffix is **.s**.

D-AS assembles the input file and generates a corresponding output file in Common Object File Format (COFF). By default, the output file has the name of the input file with an extension suffix of is **.o**. The -o option can be used to change the output file name.

The form -@*name* can also be used for either *options* or *input-file*. If found, the name must be either that of an environment variable or file (a path is allowed), the contents of which replace -@*name*.

For example assemble `test.s` with a symbol named DEBUG defined as 2 that may be used in conditional assembly statements:

```
das -D DEBUG=2 test.s
```

Command Line Options

The following command line options are available.

➤ Command line options are case-sensitive, for example, -c and -C are two unrelated options. For easier reading, command line options may be shown with embedded spaces in the table. In writing options on the command line, space is allowed only follow the option letter, not elsewhere. For example, "`-D DEBUG=2`" is valid; "`-D DEBUG = 2`" is not.

If the same option is given more than once, perhaps when using the -@*name* form, the last instance is used.

Option	Description
-D *name*[=*value*]	Define symbol *name* to have the given *value*. If *value* is not given, 1 is used. The -D option can be used to set symbols used with conditional assembly. See the **if** directive on page 24 for more information.
-H	Include a header in the listing.
-I *path*	Specify a directory where the assembler will look for include files. See the **include** directive on page 25 for more information.
-l	Generate a listing file to file *input-file*`.L`.
-L	Generate a listing to standard output.
-o *outfile*	Use *outfile* as the filename for the created object file. If not specified, *infile*.o will be used.
-R	Remove *infile*.s at the end of assembly.
-T *ad-file*	Specify which assembler description (.ad) file to use. This is normally set automatically by defining the DTARGET and the DOBJECT environment variables or by using the -WDDTARGET and the -WDDOBJECT command line options. It is primarily for internal use by Diab Data.
-V	Print version number.
-WDDOBJECT=*object*	Specify the object format and mnemonic type. Overrides the environment variable DOBJECT if it is also set. See the release notice for available targets.
-WDDTARGET=*target*	Specify the target processor. Overrides the environment variable DTARGET if it is also set. See the release notice for available targets.
-x	Discard all local symbols.
-X	Discard all symbols starting with **.L**. Supports compilers using this form for automatically generated symbols.

Option	Description
-X*option*	Set options providing detailed control of the assembler. No space is allowed after the X. See -X options below.
-#	Print all command line options on standard output.

-X options

The following options control the operation of the assembler.

Table 2-1 -X options

Option	Description
-Xcpu-403	Only accept assembly code for the PowerPC 403 processor.
-Xcpu-505	Only accept assembly code for the PowerPC 505 processor.
-Xcpu-601	Only accept assembly code for the PowerPC 601 processor.
-Xcpu-602	Only accept assembly code for the PowerPC 602 processor.
-Xcpu-603	Only accept assembly code for the PowerPC 603 processor.
-Xcpu-604	Only accept assembly code for the PowerPC 604 processor.
-Xcpu-821	Only accept assembly code for the PowerPC 821 processor.
-Xcpu-all	Accept assembly code for all the above PowerPC processors.
-Xdefault-align=*value*	Set the default alignment of a COFF section. The section is padded so that the size becomes a multiple of the alignment value. Other object formats (e.g., ELF) set the alignment to the maximum alignment used within the section. The default value of this option is 8.
-Xgnu-locals	Enable local GNU labels. See "GNU-style locals" on page 12 for more information. This is the default.
-Xgnu-locals-off	Disable local GNU labels. See "GNU-style locals" on page 12 for more information. The default setting is -Xgnu-locals-on.
-Xheader	Include a header in the listing. See the -l and the -L options. This option is turned off as a default. This option has the same effect as the -H option. See also -Xheader-format.
-Xheader-off	Do not includes a header in the listing file. This is the default.

Table 2-1 -X options

Option	Description
-Xheader-format=*string*	Define the format of the header in the assembly listing. The header *string* can contain the following directives in any order introduced by a '%'. Characters not preceded by '%' are printed as is, including spaces and escapes such as '\t' for tab. *%n*E Use *n* columns to display the error count. *%n*F Use *n* columns to display the file name. %N Start a new line. *%n*P Use *n* columns to display the page number *%n*S Use *n* columns to display the subtitle given with the -Xsubtitle option *%n*T Use *n* columns to display the title given with the -Xtitle option *%n*W Use *n* columns to display the warning count The default header string is "%30T File: %10F Errors %4E".
-Xlabel-colon	Require that all label definitions have a colon ':' appended. When this option is selected, some directives are allowed to start the line.
-Xlabel-colon-off	Does not require label definitions to end with a colon ':'. When this option is selected, directives are not allowed to start the line. This is the default.

Table 2-1 -X options

Option	Description
-Xline-format=*string*	Define the format of each assembly line in a listing. The *string* can contain the following directives, in any order, starting with a '%'. Characters not preceded by '%' are printed as is, including spaces and escapes such as '\t' for tab. *%n*A Use *n* columns to display current address *%n.m*C Use *n* columns to display the generated code. A space is inserted at every *m*th column. *%n*D Display a maximum of *n* generated bytes for each source line. If *n* is zero, there is no limit. More than one listing line might be used to display lines that produce many bytes. *%n*L Use n columns to display the current source line number. *%n*P Use n columns to display the current PLC (Program Location Counter) which corresponds to a section number. The assembly source statement follows the above items on the listing line. The default line format string is "%8A %2P %32D%15.2C%5L\t".
-Xlist-file	Generate a listing file to le *input-file* . L. Same as the -l option.
-Xlist-off	Generate no listing file. This is the default.
-Xlist-tty	Generate a listing file to standard output. Same as the -L option.
-Xllen=*n*	Define the line length of the listing file. The default is 132 characters.
-Xobject-format= object-string	Set the object format D-AS should produce. The following object-strings are valid: -Xcoff COFF (Common Object File Format) -Xelf ELF (Executable and Linkable Format) The object format is set automatically by the DOBJECT environment variable and the -WDDOBJECT option and should not be set explicitly.
-Xpage-skip=*n*	If *n* is zero, page breaks in the listing file will be created using formfeed (ASCII 12). Otherwise each page will be padded with *n* blank lines. The default value is zero.
-Xplen=*n*	Define the number of lines per page in the listing file. The default value of *n* is 60.
-Xspace	Using this option, D-AS allows spaces between operands in an assembly instruction. This is the default.

Table 2-1 -X options

Option	Description
-Xspace-off	Using this option, spaces are not allowed between operands in an assembly instruction.
-Xstrip-locals	Do not include local symbols in the symbol table. This is the same as the -x option.
-Xstrip-locals-off	Include local symbols will in the symbol table. This is the default.
-Xstrip-temps=*string*	Do not include local labels starting with *string* in the symbol table. If no *string* is specified, `.L` will be used. This is the same as the -X option.
-Xstrip-temps-off	Include local symbols starting with `.L` in the symbol table. This is the default.
-Xsubtitle=*string*	Define a subtitle that will be printed in the %S field of the header. See -Xheader-format for more information.
-Xtab-size=*n*	Define how many spaces there are between tab stops. The default is 8.
-Xtitle=*string*	Define a title that will be printed in the %T field of the header. See -Xheader-format for more information.

3 Syntax Rules

Format of an Assembly Language Line

An assembly language file consists of a series of statements, one per line. The maximum number of characters in an assembly line is 1024.

The format of an assembly language statement is:

[*label*:] [*opcode*] [*operand field*] [; *comment*]

Spaces and tabs may be used freely between fields and between operands (except that -Xspace-off option prohibits spaces between operands, see page 8).

The comment character, shown as a semicolon above, is different for different mnemonics. See "Comment" on page 10.

All fields are optional depending on the circumstances. In particular:

1. Blank lines are permitted.
2. A statement may contain only a *label.*
3. A statement may contain only an *opcode.*
4. A line may consist of only a *comment* beginning in any column.

An example of assembly language code follows:

```
; mv_word(dest,src,cnt)
; move cnt (r5) words from src (r4) to dest(r3)
        .text
        .globl  mv_word
mv_word:
        b       .L5
.L4:
        addic.  r5,r5,-1
        lwzx    r12,r4,r5
        stwx    r12,r3,r5
.L5:
        bne     .L4
        blr
```

Labels

A *label* is a user-defined symbol which is assigned the value of the current location counter; both of which are entered into the assembler's symbol table. The value of the label is relocatable.

A label is a symbolic means of referring to a specific location within a program. The following govern labels:

- A label is a symbol. See "Symbols" on page 10 for the rules on forming symbols.
- A label always occurs first in a statement. There may be multiple labels on one line.
- A label may be optionally terminated with a colon, unless the -Xlabel-colon option is used in which case the colon is required.

Examples:

```
start:
genesis: restart:       ; Multiple labels
7$:                     ; A local label, defined below
4:                      ; A local label, defined below
```

Opcode

The opcode of an assembly language statement identifies the statement as either a machine instruction or an assembler directive. One or more blanks (or tabs) must separate the opcode from the operand field in a statement. No blanks are necessary between a label ending with a colon and an opcode, however, they are recommended to improve readability.

A machine instruction is indicated by an instruction mnemonic.

An assembler directive (or just "directive"), performs some function during the assembly process. It does not produce any executable code, although it may assign space in a program for data.

D-AS is case insensitive regarding opcodes.

Operand field

In general, an operand field consists of 0-5 operands separated by commas.

The format of the operand field for machine instruction statements is the same for all instructions. The format of the operand field for assembler directives depends on the directive itself.

Comment

The comment delimiters in D-AS are the semicolon ';' and the pound sign '#'.

In addition, an asterisk '*' in column 1 is also treated as a comment delimiter.

The comment field consists of all characters in a source line following and including the comment character. These characters are ignored by the assembler. With the exception of the <Newline> character, which starts a new line, any character may appear in the comment field.

Symbols

A symbol consists of a number of characters, with the following restrictions:

- Valid characters include A-Z, a-z, 0-9, period '.', underscore '_', and dollar sign '$'.
- The first character must not be numeric or the dollar sign, unless the symbol is a local label.

The only limit to the length of symbols is the amount of memory available to the assembler. Upper and lower cases are distinct, "Alpha" and "alpha" are separate symbols.

A symbol is said to be ***declared*** when the assembler recognizes it as a symbol of the program. A symbol is said to be ***defined*** when a value is associated with it. A symbol may not be redefined, unless it was initially defined with the directive symbol .set expression (see page 28).

There are several ways to define a symbol:

- As the label of a statement

- In a direct assignment statement
- With the **.equ/.set** directives
- As a local common symbol via the **.lcomm** directive

The **.comm** directive will declare a symbol as a common symbol. The linker will allocate all common symbols with the same name to the same address, unless it is defined in one file.

Direct assignment statements

A direct assignment statement assigns the value of an arbitrary expression to a specified symbol. The format of a direct assignment statement is one of the following:

symbol[:] = *expression*

symbol[:] =: *expression*

The =: syntax has the side effect that symbol will be visible outside of the current file. Examples of valid direct assignments are:

```
vect_size = 4
vectora = 0xfffe
vectorb = vectora-vect_size
CRLF: =: 0x0D0A
```

Reserved symbols

The following symbols have a special meaning for the assembler:

Symbol name	Description
., $	Program location counter.

External symbols

A program may be assembled in separate modules, and linked together to form a single program. By using external symbols it is possible to define a label in one file and use it in another. The linker will relocate the reference so that the same address is used. There are two forms of external symbols:

- Ordinary external symbols declared with the **.globl/.global/XDEF** directive.
- Common symbols declared with the **.comm** directive.

For example, the following statements define the array `table` and the routine `two` to be external symbols:

```
        .globl  table, two
        .data
table:
        .space  20              # twenty bytes long
        .text
two:
```

```
addi    r3,r0,2         # return 2
blr
```

External symbols are only declared to the assembler by the **.globl/.global/XDEF** directive. They must be defined (i.e., given a value) in another statement by one of the methods mentioned above. They need not be defined in the current file; in that case they are flagged as "undefined" in the symbol table. If they are undefined, they are considered to have a value of zero in expressions.

The following statements, located in a different file, are using the above defined labels:

```
bl      two
addis   r4,r0,table@ha
stw     r3,r4,table@l
```

Note that whenever a symbol is used that is not defined in the same file, it is concidered a global undefined symbol by the assembler.

An external symbol is also declared by the directive .comm symbol, size [,alignment] (see page 21) by multiple modules. For the rest of the assembly such a symbol, called a common symbol, will be treated as though it was an undefined global symbol. D-AS does not allocate storage for common symbols; this task is left to the link editor. The link editor computes the maximum size of each common symbol with the same name, which may appear in multiple object modules, allocates storage for it in the final `.bss` section and resolves linkages.

Local symbols

Local symbols provide a convenient way of generating labels for branch instructions. Use of local symbols reduces the possibility of multiply-defined symbols in a program, and separates entry point symbols from local references, such as the top of a loop. Local symbols cannot be referenced by other object modules. D-AS implements two styles of local variables.

Generic style locals

The generic style local symbols are of the form *n*$ where *n* is any integer.

Examples of valid local symbols:

```
1$
27$
394$
```

Leading zeroes are significant, e.g., `2$` and `02$` are different symbols. A local symbol is defined and referenced only within a single local symbol block. There is no conflict between local symbols with the same name which appear in different local symbol blocks. A new local symbol block is started when either

- a non-local label is defined
- a new program section is entered

GNU-style locals

A GNU-style local symbol uses only one digit when defined. A GNU-style local symbol is referenced with a digit followed by the character 'f' or 'b'. When the digit is suffixed by an 'f', the nearest definition going forward (toward the end of the source) is refer-

enced. When suffixed with the character 'b', the nearest backward definition (toward the beginning of the file) is referenced.

Example:

```
5:
        .long   5f      ; Reference definition below.
        .long   5b      ; Reference definition above.
5:
```

By default the GNU style local symbols are recognized by D-AS. This can be disabled with the option -Xgnu-locals-off (see page 5).

Constants

Integral Constants

Internally, D-AS treats all integer constants as signed 32-bit binary two's complement quantities. A constant may be specified in the listed formats. The order of the list is significant in that it is scanned from top to bottom, and the first matching format is used.

Format	Description
'*character*'	character constant
0x*hex-digits*	hexadecimal constant
0*octal-digits*	octal constant
$*hex-digits*	hexadecimal constant
/*hex-digits*	hexadecimal constant
@*octal-digits*	octal constant
%*binary-digits*	binary constant
*hex-digits*h	hexadecimal constant
*decimal-digits*d	decimal constant
*octal-digits*o	octal constant
*octal-digits*q	octal constant
*binary-digits*b	binary constant
decimal-digits	decimal constant

Examples:

```
abc  =  12      12 decimal
bcd  =  012     12 octal (10 decimal)
cde  =  0x12    12 hex (18 decimal)
```

4 Sections and Location Counters

Program sections

Assembly language programs are usually divided into different sections in order to separate executable code from data, constant data from variable data, initialized data from uninitialized data, etc. The table below describes some predefined sections.

Name	Directive	Description
.text	".text" (page 29)	instruction space
.data	".data" (page 22)	initialized data
.bss	".bss" (page 21)	uninitialized data
.sdata	".sdata" (page 27)	short initialized data
.sdata2	".sdata2" (page 27)	constant short initialized data
.sbss	".sbss" (page 27)	short uninitialized data

By invoking any of the directives in the table above, it is possible to switch between the different sections of the assembly language program. New sections can also be defined with the .section name, [alignment], [type] (see page 27) directive.

D-AS maintains a separate location counter for each section, thus for assembly code like:

```
.text
instruction-block-1
.data
data-block-1
.text
instruction-block-2
.data
data-block-2
```

In the object file, *instruction-block-2* will immediately follow *instruction-block-1*, and *data-block-2* will immediately follow *data-block-1*.

Location counters

The assembly current location counter is represented by the character '.' or the character '$'. In the operand field of any statement or assembly directive it represents the address of the first byte of the statement.

➤ A current location counter appearing as the third operand in a .byte expression ,... (see page 21) directive still has the value of the address at which the first byte was loaded; it is not updated while evaluating the directive.

At the beginning, the assembler sets the location counter to zero. Normally, consecutive memory locations are assigned to each byte of the generated code. However, the location

where the code is stored may be changed by a direct assignment altering the location counter:

. = *expression*

expression must not contain any forward references, must not change from one pass to another, and must not have the effect of reducing the value of '.'. Note that D-AS supports absolute sections when using ELF, so setting '.' to an absolute position is equivalent to using the **.org** directive and will produce a section named **.abs.***XXXXXX*, where *XXXXXX* is the hexadecimal address of the section. The linker will then place this section at the specified address:

```
. = 0xffff0000
```

Storage area may also be reserved by advancing the '.'. For example, if the current value of '.' is 0x1000:

```
. = .+0x100
```

would reserve 100 (hex) bytes of storage. The next instruction would be stored at address 0x1100. Note that

```
.skip 0x100
```

is a more readable way of doing the same thing.

5 Expressions

Expressions are combinations of terms joined together by unary or binary operators. An expression is always evaluated to a 32-bit value. If the instruction calls for only one byte, the low-order 8 bits are used.

A *term* is a component of an expression. A *term* may be one of the following:

1. A constant.
2. A symbol.
3. An expression or *term* enclosed in parenthesis `()`. Any quantity enclosed in parenthesis is evaluated before the rest of the expression. This can be utilized to alter the normal precedence of operators, e.g., differentiating between `a*b+c` and `a*(b+c)` or to apply a unary operator to an entire expression, e.g., `-(a*b+c)`.

The *unary* operators recognized by D-AS are:

Unary Operator	Description
%lo(*expr*) *expr*@l	The least significant 16 bits of *expr* are extracted.
%hi(*expr*) *expr*@h	The most significant 16 bits of *expr* are extracted.
%hiadj(*expr*) *expr*$ha	The most significant 16 bits of *expr* are extracted, and adjusted for the sign of the least significant 16 bits of *expr*.
%sdaoff(*expr*) *expr*@sdarx	The 16 bit offset of *expr* from the SDA base register is calculated. The produced relocation will cause the linker to modify the destination register field in the instruction.
expr@sdax	The 16 bit offset of *expr* from the SDA base register is calculated.
+	unary add
-	negate
~	complement

The *binary* operators recognized by D-AS are:

Binary Operator	Description
+	add
-	subtract
*	multiply
/	divide
|	logical or
%	modulo

Binary Operator	Description
&	logical and
^	exclusive or
<<	shift left
>>	shift right
==	equal to
!=	not equal to
<=	less that or equal to
<	less that
>=	greater that or equal to
>	greater that

Expressions are evaluated with the following precedence in order from highest to lowest:

Operator	Associativity
@ *operations*	left to right
unary + - ~	right to left
* / %	left to right
binary + -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right

Any expression, when evaluated, is either *absolute or relocatable*:

1. An expression is *absolute* if its value is fixed. An expression whose terms are constants, or symbols whose values are constants via a direct assignment statement, is absolute. A relocatable expression minus a relocatable expression, where both items belong to the same program section is also absolute.

2. An expression is *relocatable* if it contains a label whose value will not be defined until link time. In this case the assembler will generate an entry in the relocation table in the object file. This entry will point to the instruction or data reference so that the linker can patch the correct value after memory allocation. The allowed relocatable expressions are defined in the COFF and the ELF sections in the D-LD User's Manual together with the relocation type used. The following demonstrates the use of relocatable expressions:

Expression	Relocation Mode
alpha	relocatable
alpha+5	relocatable
alpha-0xa	relocatable
alpha*2	rot relocatable (error)
2-alpha	rot relocatable, since the expression cannot be linked by adding alpha's offset to it
alpha-beta	absolute, since the distance between alpha and beta is constant, as long as they are defined in the same section
alpha@l	relocatable (the low 16 bits of alpha)

6 Assembler Directives

All the assembler directives (or just "directives") described here that are prefixed with a period '.' are also available with out the period. Most are shown with a '.' except for those traditionally written without it.

If the -Xlabel-colon option is given (see page 6), then directives which cannot take a label may start in column 1. A directive which can take a label, e.g. can produce data in the current section, may not start in column 1. If -Xlabel-colon-off is in force (the default), then no directive may start in column 1.

Spaces are optional between the operands of directives unless the -Xspace-off option is in force (see page 8).

D-AS also recognizes the following directives which are generated by compilers for symbolic debugging.

.def, .endef, .ln, .dim, .line, .scl, .size, .tag, .type and .val

The C compiler automatically generates these directives when started with the -g option.

The remainder of this chapter describes each assembler directive.

symbol[:] = expression

See "symbol .equ expression" on page 23

symbol[:] =: expression

Same as "*symbol = expression*", but symbol will be declared as a global symbol.

.align *expression*

Aligns the current location counter to the value given by *expression*. When the option -Xalign-value is set, *expression* is used as the alignment value, and must be a power of 2. When the option -Xalign-power2 is set, the alignment value is 2 to the power of *expression*.

The default is -Xalign-power2.

Example:

```
.align  4
```

Will align on 4 byte boundary if the -Xalign-value is used, and on 16 byte boundary if -Xalign-power2 is used.

.ascii "*character-string*"

The `.ascii` directive stores the internal representation of each character in the string starting at the current location. Characters represented in the source text with internal values less than 128 are stored with the high bit set to zero. Characters with source text values from 128 through 255, and characters represented by the "\nnn" construct are stored as is.

A <Newline> character must not appear within the character string. It can be represented by the escape sequence \n as described below. The (") is a delimiter character and must not appear in the string unless preceded by a backslash '\'.

The following escape sequences are also valid as single characters:

Constant	Value	Meaning
\b	8	<Backspace>
\t	9	<Horizontal Tab>
\n	10	<Line Feed> (<New Line>)
\v	11	<Vertical Tab>
\f	12	<Form Feed>
\r	13	return
\"	34	double quote
\\	92	backslash
nnn	*nnn*(oct)	octal value of *nnn*

Some examples follows:

Statement	Hex Code Generated
.ascii "hello there"	68 65 6C 6C 6F 20 74 68 65 72 65
.ascii "Warning-\007\007\n"	77 61 72 6E 69 6E 67 2D 07 07 0A

.asciz "*character-string*"

The `.asciz` directive is equivalent to the ascii directive with a zero (null) byte automatically appended as the final character of the string. In the C language, strings are null terminated.

.blkb *expression*

See ".skip size" on page 28.

.bss

Switches output to the **.bss** section. Note that **.bss** contains uninitialized data only, which means that the **.skip**, **.space**, and **DS.B** directives are the only useful directives inside the **.bss** section.

.bsect

See ".bss" on page 21.

.byte *expression ,...*

Reserves one byte for each expression in the operand field and initializes the value of the byte to be the low-order byte of the corresponding expression. Multiple expressions must be separated by commas.

Example:

```
.byte   17,65,0101,0x41         ; reserves 4 bytes
.byte   0                       ; reserves a single byte
                                ; with 0
```

.comm *symbol, size* [*,alignment*]

A common block with length given by expression *size* bytes is assigned to *symbol*. All common blocks with the same name in different files will refer to the same block. The linker will allocate space for it and put it in the **.bss** section. The optional *alignment* expression is used to align the common block. If not specified the common block is aligned on its natural size, up to the default alignment specified by the -Xdefault-align option. See ".align expression" on page 20 for a description of the *alignment* format. Note that some object file formats (e.g., COFF) are not capable of handling common block alignment.

Example:

```
.comm   array,100
```

dc.b *expression*

See ".byte expression ,..." on page 21.

dc.l *expression*

See ".long expression ,..." on page 26.

dc.w *expression*

See ".word expression ,..." on page 30.

ds.b *size*

See ".skip size" on page 28.

.data

Switches output to the **.data** (initialized data) section.

.dsect

See ".data" on page 22.

.eject

Forces a page break if a listing is produced by the -L or -l options.

.else

See ".elsec" on page 22.

.elsec

The .elsec directive may be used in a conditional assembly block to reverse the state of the conditional assembly, i.e., if statements were skipped prior to the **.elsec** directive, statements following the **.elsec** directive will be processed, and vice versa.

.end

This directive indicates the end of the source program. All characters after the end directive are ignored.

.endc

This directive indicates the end of a condition block. Each **.endc** directive must be paired with a **.if** directive.

.endif

See ".endc" on page 22.

.endm

This directive indicates the end of a macro body. Each **.endm** directive must be paired with a **.macro** directive. See "Macros" on page 31 for a detailed description.

.entry *symbol*,...

See ".global symbol ,..." on page 23.

symbol .equ *expression*

The statement must be labeled with a symbol and sets the symbol to be equal to *expression*.

Example:

```
nine    .equ    9
```

.even

Aligns the location counter on the default alignment value, specified by the -Xdefault-align option.

.extern *symbol*,...

Declares each symbol in the symbol list to be defined in a separate module. The linker supplies the value from the defining module during linking. It is an error to name a symbol that is given a value in the current module.

Example:

```
.extern add,sub,mul,div
```

.file "*filename*"

Specifies the name of the source file for the symbol table of the object file. The default is the name of the file. This directive is used by compilers to pass the name of the original source file to the symbol table.

Example:

```
.file   "test.c"
```

.global *symbol*,...

Declares each symbol in the symbol list to be visible outside the current module. This makes each symbol available to the linker for use in resolving **.extern** references to the symbol.

Example:

```
.global add,sub,mul,div
```

.globl *symbol ,...*

See ".global symbol ,..." on page 23

.ident "*character-string*"

Appends the character string to a special section called **.comment** in the object file. Example:

```
.ident "version 1.1"
```

.if *expression*

The **.if** construct provides for conditional assembly. If *expression* evaluates to zero, all subsequent lines until the end of the condition block or until a **.elsec** statement, are skipped. Otherwise all subsequent lines until the end of the condition block or until a **.elsec** directive will be processed. The condition block must be terminated by an **.endc** statement. **.if** constructs may be nested.

Example:

```
        .if     long_file_names
maxname .equ    1024
        .elsec
maxname .equ    14
        .endc
```

.ifeq *expression*

.ifeq is an alias for **.if** *expression* == 0. See ".if expression" on page 24 for more details.

.ifne *expression*

.ifne is an alias for **.if** *expression* **!= 0**. See ".if expression" on page 24 for more details.

.ifge *expression*

The `.ifge` is an alias for **.if** *expression* **>= 0**. See ".if expression" on page 24 for more details.

.ifgt *expression*

The `.ifgt` is an alias for **.if** *expression* **> 0**. See ".if expression" on page 24 for more details.

.ifle *expression*

The `.ifle` is an alias for **.if** *expression* **<= 0**. See ".if expression" on page 24 for more details.

.iflt *expression*

The `.iflt` is an alias for **.if** *expression* **< 0**. See ".if expression" on page 24 for more details.

.include *filename*

Inserts the contents of the named file after the **.include** directive. May be nested to any level.

Example:

```
.include "globals.h"
```

.lcnt *expression*

Defines the number of lines on each page if the -L or -l option is specified.

Example:

```
.lcnt   72
```

.lcomm symbol, size [,alignment]

Allocates a local common block of length *size* expression bytes in the **.bss** section. The optional *alignment* expression is used to align the allocated data area. If not specified the allocated data area is aligned on its natural size, up to the default alignment specified by the -Xdefault-align option. See ".align expression" on page 20 for description of the *alignment* format. Note that symbol is not made visible outside the current module.

Example:

```
.lcomm  local_array,200
```

.list

Turns on listing of lines following the **.list** directive if the option -L or -l is specified. Listing can be turned off with the **.nolist** directive.

.llen *expression*

Defines the line length if the -L or -l option is specified.

Example:

```
.llen   132
```

.long *expression ,...*

Reserves one long word (32 bits) for each expression in the operand field and initializes the value of the word to the corresponding expression.

Example:

```
.long   0xfedcba98,0123456,-75  ; reserves 12 bytes.
```

name .macro[.*parameter*] [*parameter ,...*]

Start definition of macro *name*. All lines following the **.macro** directive until the corresponding **.endm** directive are part of the macro body. See "Macros" on page 31 for a detailed description.

.name "*filename*"

See ".file "filename"" on page 23.

.ncros

See ".nolist" on page 26.

.nolist

Turns off listing of lines following the **.nolist** directive if the option -L or -l is specified. Listing can be turned on with the **.list** directive.

.org *expression*

Sets the current location counter to the value of *expression*. The value must either be an absolute value or be relocatable and greater than or equal to the current location. Using the **.org** directive with an absolute value in ELF mode will produce a section named **.abs.***XXXXXX*, where *XXXXXX* is the hexadecimal address of the section. The linker will then place this section at the specified address. Example:

```
.org    0xffff0000
```

.page

See ".eject" on page 22.

.pagelen *expression*

See ".lcnt expression" on page 25.

.plen *expression*

See ".lcnt expression" on page 25

.previous

Assembly output is directed to the program section selected prior to the last **.section**, **.text**, **.data**, etc. directive.

.psect

See ".text" on page 29.

.psize *page-length* [,*line-length*]

Defines the page size for listing if the -L or -l option is specified. The *page-length* expression specifies the number of lines per page. The *line-length* expression specifies the number of characters per lines. See also ".lcnt expression" on page 25, and ".llen expression" on page 25.

Example:

```
.psize  72,132
```

.sbss

Switches output to the **.sbss** (short uninitialized data space) section.

.sbttl "*character-string*"

See ".subtitle "character-string"" on page 29.

.sdata

Switches output to the **.sdata** (short data space) section.

.sdata2

Switches output to the **.sdata2** (constant short data space) section.

.section *name*, [*alignment*], [*type*]

The assembly output is directed into the program section with the given name . The section name may be quoted with the (") character or not quoted. The section is created if it does not exist, with the attributes specified by *type*, a single character. Valid *type* characters are:

Type Character	Description
c	section contains executable code
d	section contains data
m	code and data is mixed
r	readonly data

The *alignment* expression specifies the minimum alignment that must be used for the section. Note that some object formats (COFF) are unable to handle the alignment information. Instead, the linker command language can be used to align the section.

.set *symbol, expression*

Defines *symbol* to be equal to the value of *expression*. This is an alternative to the **.equ** directive.

Example:

```
.set    nine,9
```

symbol .set *expression*

Defines *symbol* to be equal to the value of *expression*. This form of the .set is different from the **.equ** directive in that is it possible to redefine the value of *symbol* later in the same module. *expression* may not refer to an external or undefined symbol.

Example:

```
number  .set    9
        ...
number  .set    number+1
```

.short *expression* ,...

Reserves one 16 bit word for each expression in the operand field and initializes the value of the word to the corresponding expression.

Example:

```
.short  0xba98, 012345, -75, 17 ; reserves 8 bytes.
```

.size *symbol, expression*

Sets the size information for *symbol* to *expression*. Note that only some object file formats are using the size information.

.skip *size*

The `.skip` directive reserves a block of data initialized to zero. *size* is an expression giving the length of the block in bytes.

Example:

```
name:    .skip    8
```

is the same as:

```
name:    .byte    0,0,0,0,0,0,0,0
```

.space *expression*

See ".skip size" on page 28.

.string "*character-string*"

See ".ascii "character-string"" on page 20.

.strz "*character-string*"

See ".asciz "character-string"" on page 21.

.subtitle "*character-string*"

Sets the subtitle to the character string. The subtitle may be set any number of times. The default subtitle is blank.

```
.subtitle "string search function"
```

.text

Switches output to the **.text** (instruction space) section.

.title "*character-string*"

Sets the title to character string. The title may be set any number of times. The default title is blank.

Example:

```
.title "program.s"
```

.ttl *character-string*

See ".title "character-string"" on page 29.

.type *symbol, type*

Mark *symbol* as *type*. The *type* can be one of the following:

type	Description
#object @object	*symbol* names an object.
#function @function	*symbol* names a function.

Note that only some object file formats are using the type information.

.weak *symbol ,...*

Declares each *symbol* as a weak external symbol that is visible outside the current file. Global references are resolved by the linker. Note that only some object file formats support weak external symbols.

Example:

```
.weak add,sub,mul,div
```

.width *expression*

See ".llen expression" on page 25.

.word *expression ,...*

Reserves one word (32 bit) for each expression in the operand field and initializes the value of the word to the corresponding expression.

Example:

```
.word   0xfedcba98,0123456,-75  ; reserves 12 bytes.
```

.xdef *symbol ,...*

See ".global symbol ,..." on page 23.

.xref *symbol ,...*

See ".extern symbol ,..." on page 23.

7 Macros

Assembler macros enable the programmer to encapsulate a sequence of assembly code in a *macro definition*, and then inline that code with a simple parameterized *macro invocation*.

Example:

```
ld32:   macro   reg,ident       # macro definition
        addis   reg,r0,ident@ha
        lwz     reg,ident@l(reg)
        endm

        ld32    r3,yvar         # macro invocation #1
        ld32    r4,xvar         # macro invocation #2
```

This will produce the following code

```
        addis   r3,r0,yvar@ha   # macro expansion #1
        lwz     r3,yvar@l(r3)
        addis   r4,r0,xvar@ha   # macro expansion #2
        lwz     r4,xvar@l(r4)
```

Macro definition

Macro definition

label: **macro**[*.parameter*] [*parameter* ,...]
macro body
endm

where *label* is the name of the macro, without containing any period. The optional parameters can be referenced in the macro body in two different ways:

1. By using the parameter name:

```
madd:   macro   par1,par2,par3  # definition
        add     par1,par2,par3
        endm

        madd    r7,r8,r9        # invocation
```

produces

```
        add     r7,r8,r9
```

2. By using a \n syntax where **\1**, **\2**, ... **\9**, **\A**, ... **\Z** are the first, second, etc. actual parameters passed to the macro. When the \n syntax is used, formal parameters are optional in the macro definition. If present, both the named and numbered form may be freely mixed in the same macro body.

```
madd:   macro                   # definition
        add     \1,\2,\3
```

```
        endm

        madd    r7,r8,r9        # invocation
```

produces

```
        add     r7,r8,r9
```

The special parameter \0 denotes the actual parameter attached to the macro name with a '.' character in an invocation. Usually this is an instruction size.

```
move:   macro   dreg,sreg       # definition
        l\0z    r0,0(sreg)
        st\0    r0,0(dreg)
        endm

        move.h  r10,r11         # invocation
```

produces

```
        lhz     r0,0(r11)
        sth     r0,0(r10)
```

Separating parameter names from text

In the macro body, the characters '&&' can optionally precede or follow a parameter name to concatenate it with other text. This is useful when a parameter is to be part of an identifier:

```
xadd:   macro   par1,par2,hcnst # definition
        addi    par1,par2,0x&&hcnst
        endm

        xadd    r4,r5,ff00      # invocation
```

produces

```
        addi    r4,r5,0xff00
```

Generating unique labels

The special parameter \@ is replaced with a unique string to make it possible to create labels that are different for each macro invocation:

```
lstr:   macro   reg,string      # definition
        data
.Lm\@:
        byte    string,0
        previous
        addi    reg,r0,.Lm\@
        lwz     reg,.Lm\@(reg)
        endm

        lstr    r3,"abc"        # invocation
```

produces

```
        data
.Lm.0001:
        byte    "abc",0
        previous
        addi    r3,r0,.Lm.0001
        lwz     r3,.Lm.0001(r3)
```

NARG symbol

The special symbol NARG represents the actual number of non-blank parameters passed to the macro:

```
init:   macro   value           # definition
        if      NARG == 0
        byte    0
        else
        byte    value
        endc
        endm

        init                    # invocation #1
        init    10              # invocation #2
```

produces

```
        byte    0               # expansion #1
        byte    10              # expansion #2
```

Invoking a macro

A macro is invoked by using the macro name anywhere an instruction can be used. The macro body will be inserted at the place of invocation, and the formal parameters in the macro definition will be replaced with the actual parameters, or operands, given after the macro name.

Actual parameters are separated by commas. To pass a an actual parameter that includes special characters, such as blanks, commas and comment symbols, angle brackets '< >' can be used. Everything in between the brackets is regarded as one parameter.

Example:

```
init:   macro   command,list
        data
        command list
        previous
        endm

        init    byte,<0,1,2,3>
```

produces

```
        data
        byte    0,1,2,3
        previous
```

8 Example Listing

If the -l or -L option is specified, a listing is produced.

The listing contains the following columns:

Table 8-1 Listing Columns

Column	Description
Undefined flag	A 'U' is printed if the generated code from this line contains any undefined references.
Current location (Loc)	This hexadecimal value denotes the relative address of the generated code.
Relocation type (Plc)	If the code contains a relocatable symbol a character is printed to indicate what section the symbol belongs to. 'T' for **.text**, 'D' for **.data** and 'B' for **.bss**. An 'X' means that the symbol is undefined.
Generated code (Code)	The code is printed in hexadecimal.
Symbol value (Arg)	If a symbol is assigned a value on the current line, this value is printed in hexadecimal.
Line count (Lc)	Source line number.
Source statement	Source code lines.

At the end of the listing, a cross reference table for all symbols is printed in alphabetic order. A 'U' is printed in front of symbols with no assigned value. A '<' means that the symbol is undefined and a '>' means that the symbol is exported. After the name, the value of the symbol is printed, possibly followed by an 'A' if it is absolute or an 'X' if it is undefined. The next column shows the definition line number followed by a list of all the lines on which the symbol is used.

If the -H option is used, a header containing the source filename and the cumulative number of errors is displayed at the top of each page.

An example listing that is produced with the command `das -l -H example.s` follows: <<F: TO BE SUPPLIED>>

A Error Messages

When there are errors in an assembly, an error message appears on the standard error output (e.g., the terminal) listing the type of error and the source line number. If an assembly listing is requested, and there are errors, the error message appears after the offending statement. If there were no assembly errors, there is no message; indicating a successful assembly. Warnings indicate possible errors, but will not terminate assembly.

If an assembly listing was not requested, any source lines which caused an assembly diagnostic are displayed on the terminal (the standard error file). In addition, a list of assembly errors and their description are also displayed on the terminal.

The common error messages, and their probable cause, follows:

.bf .ef or .bb .eb does not match
A **.bf** / **.ef** pair or a **.bb** / **.eb** pair does not match (indicating beginning and end of a function or a block). Note: these assembler directives are generated by the debug symbol option of compilers.

End of memory
The assembler cannot allocate enough memory.

File open error
An invalid filename is specified.

Illegal character
Found an invalid character in a character constant or character string.

Invalid constant
Found an invalid character in a number, e.g., a hexadecimal digit in a decimal number.

Illegal operand
An illegal operand is utilized in the operand field.

Illegal number of operands
An instruction is coded with an incorrect number of operands.

Illegal register expression
A register is used in the wrong context.

Illegal line terminator
The expected line terminator <Newline> is not found on the line.

Illegal tag name
The definition of the structure tag given in a **.tag** directive cannot be found.

Illegal usage
A directive is used in an illegal context.

Missing parenthesis
A matching parenthesis was not found.

Missing or invalid symbol
A symbol is missing or does not have a valid form.

Multiple defined symbol
The same label occurs twice or is redefined with a **.def** directive.

Non-relocatable expression
A relocatable expression was expected.

Nesting not allowed
.def ... .endef nesting are not allowed.

Non-linkable expression
An illegal combination of relocatable and/or external symbols is used in an expression.

Not assigned a value
A label is referenced which has no value.

Not previously defined
An undefined symbol is used in an expression where an absolute value is expected.

Odd address
Data or an instruction requiring alignment on an even address is occurs at an odd address. Use the **.align** directive (see page 20).

Only one .file allowed
Only one **.file** directive is allowed in a source file.

Target out of range
A displacement cannot fit into the space provided by the instruction.

Too many sections
Too many **.section** directives have been used.

Undecodable statement
A symbol in the opcode field is not recognized as an instruction mnemonic or directive.

B Machine Instructions

This section describes the conventions used in D-AS to specify instruction mnemonics and addressing modes.

Instruction Mnemonics

See the *PowerPC 601 RISC Microprocessor: The programming Environments* for the available instructions.

D-AS handles all PowerPC instructions, including the simplified mnemonics described in the above literature.

Operand Addressing Modes

The only addressing modes available in PowerPC instructions are registers and expressions.

Registers

Registers can be specified in the following ways:

Register	Description
r0-r31 R0-R31 0-31	General purpose registers. Can only be used where a general purpose register is expected
f0-f31 F0-F31 0-31	Floating point registers. Can only be used where a floating point register is expected
cr0-cr31 CR0-CR31 0-31	Condition code registers. Can only be used where a condition code register is expected
sr0-sr15 SR0-SR15 0-15	Segment registers. Can only be used where a segment register is expected
1-1023 xer (1) ctr (9) lr (8)	Special purpose registers. Can only be used where a special purpose register is expected. Only the most common register names are shown. D-AS recognizes all special purpose registers for the supported targets

Expressions

See Chapter 5, "Expressions," beginning on page 17 for a complete description on how expressions are handled. There are no limits on the complexity of an expression as long as all the operands are constants. When a label is used in the expression, the assembler will generate a relocation entry so that the linker can patch the instruction with the correct address. See the COFF and ELF sections in the D-LD User's Manual for a complete list of accepted expressions and their corresponding relocation types.

The following table shows a few examples of expressions used for common addressing modes:

Example		Description
addis lwz	r3,r0,var@ha r3,r3,var@l	Load r3 with the value pointed to by the 32 bit address of *var*. *var*@**ha** will extract the higher 16 bits and adjust them so that when adding the sign extended lower 16 bits, *var*@**l**, the resulting value will be the address of *var*
addis lwz	r3,r0,(var+4)@ha r3,r3,(var+4)@l	Same as above, but use the address of *var* plus 4
lwz	r3,r0,svar@sdarx	Load r3 with the value of *svar* located in one of the Small Data Areas (SDA). There are three 64KB SDAs: • One area pointed to by register r13 for regular small data. Usually in the sections **.sdata** and **.sbss** • One area pointed to by register r2 for constant small data. Usually in the sections **.sdata2** • One area located around address 0 (register r0) The assembler will generate the appropriate relocation information and the D-LD linker will patch the instruction to use the correct register and the correct offset
addis lwz	r3,r0,var@sdarx@ha r3,r3,var@sdax@l	Load r3 with the value of var by adding a 32 bit offset to one of the base registers described above. This is a way of getting position independent data for data areas bigger than 64K

Utilities User's Manual

DIAB DATA

Defining Compiler Performance

Copyright Notice

Disclaimer

Diab Data makes no representations or warranties with respect to the contents of this publication, and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Diab Data reserves the right to revise this publication and make changes from time to time in the content hereof without obligation on the part of Diab Data to notify any person or company of such revision or changes.

In no event shall Diab Data, or others from whom Diab Data has a licensing right, be liable for any indirect, special, incidental, or consequential damages arising out of or connected with a customers possession or use of this product, even if Diab Data or such others has advance notice of the possibility of such damages.

Trademarks

Diab Data, alone and in combination with D-AS, D-C++, D-CC, D-F77, and D-LD are trademarks of Diab Data, Inc. All other trademarks used in this document are the property of their respective owners.

Diab Data, Inc.
323 Vintage Park Drive
Foster City, CA 94404
USA

Tel **415-571-1700**
Fax **415-571-9068**

Email **support@ddi.com**

CONTENTS

List of Tables

1 Introduction

This manual describes utility tools that accompany the Diab Data compiler tool suites. Please see the following manuals for information on the other tools in the suites.

- *D-CC Language User's Manual*
- *D-C++ Language User's Manual*
- *D-F77 Language User's Manual*
- *D-AS Assembler User's Manual*
- *D-LD Linker User's Manual*
- *C Library Manual*
- *C++ Class Reference Manual*

Document conventions

This manual uses the following typographic conventions:

Table 1-1 Document Conventions

Example	Description
`dcc -o test.c`	This font is used for file and program names, environment variables, examples, user input, and program output.
if, main( ), #pragma, __pack__	Bold type is used for keywords, operators and other tokens of the language, library routines and entry points, and section names. Some names begin or end with underscores. These underscores and special characters such as # shown in bold are required.
variable, filename	Italic type is used for placeholders for information which you must supply. Italics are also used for emphasis, to introduce new terms, and for titles.
[optional text]	An item enclosed in brackets is optional.
{ item1 | item2 }	Two or more items enclosed in braces and separated by vertical bars means that you *must* choose exactly one of the items.
item ... item ,...	An item followed by "..." means that items of that form may be repeated separated by whitespace (spaces or tabs). A character preceding the "..." means that the items are separated by the character, shown here as a comma, and optional whitespace. The item may be a single token, an optional item enclosed in [ ] brackets (meaning that the item may appear not at all, once, or multiple times), or a set of choices enclosed in { } braces (meaning that a choice must be made from the enclosed items one or more times).

2 D-AR Archiver

Synopsis

Create and maintain an archive of files of any type, with special features for object files.

Syntax

dar *command* [*position-name*] *archive-filename* [*name*] ...

Description

The **dar** command maintains files in an archive. Archives can contain files of any kind. However, COFF and ELF object files are handled in a special way. If any of the included files is a COFF or ELF file, the archiver will generate an invisible symbol table in the archive. This symbol table is used by the linker to search for missing identifiers without scanning through the whole archive.

command is composed of a hyphen (-) followed by a command letter. One or more optional modifier letters for some commands may either be concatenated to the command letter, or may be given as separate option arguments (see below for examples).

position-name is the name of a file in the archive used for relative positioning with the -r and -m commands.

archive-filename is the name of the archive file (*archive* for short).

name is one or more files in the archive. Multiple *name* arguments are separated by whitespace.

`dar` commands and modifiers are as follows. Modifiers are shown in brackets.

Table 2-1 dar commands

-d [lv]	Delete the named files from the archive.
-m [abiv]	Move the named files. If any of the [abi] modifiers are employed, the *position-name* argument must be present and the files will be positioned in the same manner as with the -r command. Otherwise the files are moved to the end of the archive.
-p [sv]	Print the contents of the named files on the standard output. This is useful only with text files in an archive; binary files, e.g., object files are not converted and so are not normally printable.
-q [cflv]	Quickly appends the named files at the end of the archive without checking whether the files already exists. If the archive contains any COFF or ELF files, the symbol table file will be updated. If the [f] modifier is used, the files will be appended without updating the symbol table file, which is considerably faster. Use the -s command when all files have been inserted in the archive to update the symbol table.

Table 2-1 dar commands

-r [abciluv]	Replace the named files in the archive. New files are placed at the end of the archive unless one of the [abi] modifiers is used. If so, *position-name* must be given to specify a position in the archive. With the [bi] modifiers, the named files will be positioned before *position-name*; with the [a] modifier, after it. If the archive does not exist, create it. If the [u] modifier is specified, then only files with a modification date later than the corresponding files in the archive will be replaced.
-s [lR]	Update the symbol table file in the archive. Utilized when the archive is created with the -qf command.
-t [sv]	List a table of contents for the archive on the standard output.
-V	Print the version number of `dar`.
-x [lsv]	Extract the named files from the archive and place them in the current directory. The archive is not changed.

Table 2-2 dar command modifiers

	Use With Commands	
a	-m -r	Insert the named files in the archive after the file *position-name*.
b	-m -r	Insert the named files in the archive before the file *position-name*.
c	-q -r	Does not display any message when a new archive *archive-filename* is created.
f	-q	Append files to the archive, without updating the symbol table file. If any of the files already exist, multiple copies will exist in the archive. The next time the -s command is used **dar** will delete all copies but the last of the files with the same name.
i	-m -r	Insert the named files in the archive before the file *position-name*.
l	-d -q -r -s -x	Place temporary files in the current directory instead of the directory specified by the environment variable `TMPDIR`, or in the default temporary directory.
s	-p -t -x	Same as the -s command.
u	-r	Replace those files that have a modification date later than the files in the archive.
v	-d -m -p -q -r -t -x	Verbose output.
R	-s	Sort COFF and ELF files in the archive so that the linker does not have to scan the symbol table in multiple passes.

Examples

Later examples build on earlier examples.

Create a new archive `lib.a` and add files `f.o` and `h.o` to it (the -r command could also be used):

```
dar -q lib.a f.o h.o
```

Replace file f.o, and insert file `g.o` in archive `lib.a`, and also display the version of `dar`. Without the [a] modifier, the new file `g.o` would be appended to the end of the archive. With the [a] modifier and the first `f.o` acting as the *position-name* in the command, new file `g.o` is inserted after the replaced `f.o`:

```
dar -raV f.o lib.a f.o g.o
```

The above can also be given in the following form with the modifier letters given as separate options. The first item following `dar` must always be the command from Table 2-1 on page 3.

```
dar -r -a -V f.o lib.a f.o g.o
```

Quickly append `f.o` to the archive `lib.a`, without checking if `f.o` already exists. This operation is very fast and can be used as long as the archive is later cleaned with the -sR command (see below):

```
dar -qf lib.a f.o
```

Cleanup archive `lib.a` by creating a new sorted symbol table and removing all but the last of files with the same name. This is useful after many files have been added with the -qf option:

```
dar -sR lib.a
```

Extract `file.c` from archive `source.a` and place it in the current directory. The archive is unchanged.

```
dar -x source.a file.c
```

Delete `file.c` files from archive `source.a`. The file is deleted without being written anywhere.

```
dar -d source.a file.c
```

3 D-BCNT Basic Block Counter

Synopsis

Display profile data collected from one or more runs of a program.

Syntax

dbcnt [-f *profile-file*] [-h *n*] [-l *n*] [-n] [-t *n*] *source-file* ,...

Description

The **dbcnt** command displays the number of times each line in a source program has been executed.

The files to be measured must be compiled with the -Xblock-count option. By definition, a basic block is a segment of code with exactly one entrance and one exit. Thus, all statements in a basic block will have the same count. Compiling with -Xblock-count causes the compiler to insert code into each basic block to record each execution of the block. Each time the resulting program is run, the profile data is stored in the file named in the environment variable `DBCNT`. If `DBCNT` is not set, the file `dbcnt.out` will be used. If the program is executed more than once, the new profile data will be added to the existing `DBCNT` file.

➤ For information on support for file I/O and environment variables in an embedded environment, See Chapter 8, "Use in an Embedded Environment," in the *Compiler Target User's Manual*, especially the sections "Profiling in an embedded environment", "File I/O", and "Command line arguments and environment variables".

After the profile data has been collected and returned to the host, to display one or more source files together with their line counts, enter the command

`dbcnt` [*options*] *source-file1*, *source-file2*, ...

If the name of the `DBCNT` file is not `dbcnt.out`, use the -f option to provide the actual filename with the line counting information. See below for examples.

dbcnt options are:

-f *filename*	Read profile data from *filename* instead of `dbcnt.out`.
-h *n*	Do not print lines executed more than *n* times.
-l *n*	Do not print lines executed fewer than *n* times.
-n	Print the line number of every source line.
-t *n*	Print the *n* most frequently executed lines.
-V	Print the version number of `dbcnt`.

Files

`dbcnt.out`	default output file for profile data
`DBCNT`	environment variable giving the name of the profile data file

Examples

The file *file.c* is compiled with:

```
dcc -Xblock-count -o file file.c
```

When executed, the following output is produced:

```
47 numbers are multiples of 3 or 5.
```

`dbcnt` is used to show how many times each line is executed:

```
dbcnt file.c
```

`dbcnt` produces the following output:

```
file.c (1 run(s)):
      main()
      {
1       int i = 100, n = 0;
1
101     while(i > 0) {
100       if ((i % 3) == 0 || (i % 5) == 0) {
67
47          n++;
47        }
100       i--;
100     }
1       printf("%d numbers are multiples of 3 or 5.\n",n);
      }
```

➤ When a source line contains more than one basic block, such as the **if** statement above, empty lines are added to show the count of the basic blocks after the first.

`dbcnt -h0 -l0 -n *.c` may be used to find all source lines which will never execute in a program.

`dbcnt -n -t100 *.c` may be used to find the 100 most frequently executed source lines in a program.

Notes

The functions **__dbini()** and **__dbexit()** must exist in the standard library in order for the linker to be able to link the files compiled with the -Xblock-count option.

4 D-DUMP File Dumper

Synopsis

Dump or convert all or parts of COFF or ELF object files and archive files.

Syntax

ddump [*command*] [*modifiers*] *file* ,...

Description

A COFF or ELF object file consists of several different parts which can be individually dumped or converted with the `ddump` command.

`ddump` accepts both object files and archive files. In the latter case each file in the archive will be processed by the `ddump` command.

`ddump` can also be used to convert from COFF or ELF to other object formats. See the -R command for Motorola S-Records and the -I command for IEEE 695 (for COFF files only).

command is composed of a hyphen (-) followed by one or more command letters. One or more optional modifier letters for some commands may either be concatenated to the command letter, or may be given as separate option arguments. Commands and options are all represented by unique letters and so may be mixed in any order. Typically modifiers consisting of a single letter are concatenated with commands, while modifiers taking a separate argument are given as separate options (e.g., -t *index*, +z *number2*).

`ddump` commands and modifiers are as follows.

Table 4-1 ddump commands

-a	Dump the archive header for all the files in an archive file.
-B	Convert a hexadecimal file to binary format. Each pair of hexadecimal numbers is translated to one byte in the output file. Whitespaces (spaces, tabs, and newlines) are ignored. Unless the -o modifier is used, the output file will be named `bin.out`.
-c	Dump the string table in each object file.
-f	Dump the file header in each object file.
-g	Dump the symbols in the global symbol table built into every archive file.
-H	Display the contents of any file in hexadecimal and ASCII formats. The -p modifier will display hexadecimal only.
-h	Dump the section headers in each object file.
-I	Convert a COFF file to IEEE 695 object format. The output file will be named `ieee.out` unless the -o modifier is used. The -p modifier below can be used to specify the processor name. (Not available for ELF files.)
-l	Dump the line number information in each object file.

Table 4-1 ddump commands

-N	Dump the symbol table information in each object file. Similar to the UNIX **nm** command. The following special modifiers are available: -x Display numbers in hexadecimal -o Display numbers in octal -u Display only undefined symbols -p Display symbols in BSD format -h Suppress header -r Display filename before symbol name
-o	Dump the optional header in each object file. (Not to be confused with the -o modifier defined in Table 4-2 on page 11.)
-R	Convert a COFF or ELF file to Motorola S-Record format. The output file will be named `srec.out` unless the -o modifier is used (see Table 4-2 on page 11). If the -p modifier is used, a plain ASCII file with the contents of the sections in hexadecimal will be written. If the -u modifier is used, a binary file with the contents of the sections in hexadecimal will be written. If the -v modifier is used, the **.bss** section will not be converted or output. -v can also be used together with -p and -u.
-r	Dump the relocation information in each object file.
-S	Display the size of the COFF or ELF sections. Similar to the UNIX **size** command. By using the -f modifier, the section names will be included in the output. By using the -v modifier all sections will be included.
-s	Dump the section contents in each object file.
-T	Remove the symbol table information in each object file. Similar to the UNIX **strip** command.
-t	Dump the symbol table information in each object file.
-t *index*	Dump the symbol table information for the symbol indexed by *index* in the symbol table.
+t *index*	Dump the symbol table information for the symbols in the range given by the -t option through the +t option. If no -t was given, 0 is used as the lower limit.
-V	Print the version number of `ddump`.
-z *name*	Dump the line number information for the function *name*.
-z *name,number*	Dump the line number information in the range *number* to *number2* given by +z for the function *name*.
+z *number2*	Provide the upper limit for the -z option.

Table 4-2 ddump command modifiers

	Use With Command	
-d *number*	-h -l -R -s	Dump information for sections greater than or equal to *number*. Sections are numbered 1, 2, etc.
+d *number*	-h -l -r -R -s	Dump information for sections less than or equal to *number*.
-n *name*	-h -l -R -s -t	Dump the information associated with *name*.
-o *name*	-I -R	Specify an output file name for the -B, -I, and -R commands.
-p	any	Suppress printing of headers. Special meaning with -R.
-u	any	Underline file names. Special meaning with -R.
-v	any	Dump information in verbose mode. Special meaning with -R.
-p name	-I only	Set the processor name in the "Module Begin" record. If this option is not specified the processor name is taken from the magic number of the input file. A list of processor names and magic numbers can be found in the IEEE 695 specification.

Examples

Dump the file header and symbol table from each object file in an archive in verbose mode:

```
ddump -ftv lib.a
```

Convert an object file named `test.out` to Motorola S-Record format, naming the output file `test.rom`. The **.bss** section is suppressed:

```
ddump -Rv -o test.rom test.out
```

Same as the prior example but convert and output only section numbers 5-7 and call the result `data.rom`.

```
ddump -R -d 5+d 7-o data.rom test.out
```