WIND RIVER

Wind River® Compiler for ARM/XScale

USER'S GUIDE

5.4

Corporate Headquarters
Wind River Systems, Inc.
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.

toll free (U.S.): (800) 545-WIND
telephone: (510) 748-4100
facsimile: (510) 749-2010

For additional contact information, please visit the Wind River URL:

http://www.windriver.com

For information on how to contact Customer Support, please visit the following URL:

http://www.windriver.com/support

Wind River Compiler for ARM/XScale User's Guide, 5.4

26 Apr 06
Part #: DOC-15788-ND-00

Contents

PART III: WIND RIVER ASSEMBLER

PART I

Introduction

1 Overview

1.1 Introduction

This manual describes all tools in the Wind River Compiler toolkit (formerly known as the Diab Compiler) for the ARM family of RISC microprocessors, including the Thumb variant and the Intel XScale architecture. It includes detailed information about each tool, optimization hints, and guidelines for porting existing code to the compilers and assembler.

For introductory information, including an example program, see the *Getting Started* manual.

1.2 Overview of the Tools

The compiler suite includes high-performance C and C++ tools designed for professional programmers. Besides the benefits of state-of-the-art optimization, they reduce time spent creating reliable code because the compilers and other tools are themselves fast, and they include built-in, customizable checking features that will help you find problems earlier.

With hundreds of command-line options and special pragmas, and a powerful linker command language for arranging code and data in memory, the tools can be customized to meet the needs of any device software project. Special options are provided for compatibility with other tools and to facilitate porting of existing code.

Important Compiler Features and Extensions

- Many compiler controls and options for greater flexibility over compiler operation and code generation.
- Many features and extensions targeted for the device programmer. See *15. Use in an Embedded Environment*.
- Optimizations and features tailored individually for each processor type within the ARM microprocessor family. See *4.3 Alternatives for Selecting a Target Configuration*, p.25 for information on how to specify the target processor.
- Extensive compile-time checking to detect suspicious and nonportable constructs. See *11. The Lint Facility*.
- Powerful profiling capabilities to locate bottlenecks in the code. The profiling information can also automatically be used as feedback to the compiler, enabling even more aggressive optimizations. See *10. Optimization*, and the discussion of **D-BCNT** in *28. D-BCNT Profiling Basic Block Counter*.
- C++ templates, exceptions, and run-time type information.

High Performance Optimizations

A wide range of optimizations, some of which are unique to the Wind River Compiler, produce fast and compact code as measured by independent benchmarks. Special optimizations include superior interprocedural register allocations, inlining, and reaching analysis.

Optimizations fall into three categories: local, function-level, and program-level, as listed next. See *10. Optimization*.

- Local optimizations within a block of code:

  Constant folding
  Integer divide optimization
  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
  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
  Global variable store delay
  Lifetime analysis (coloring)
  Link register optimization
  Loop count-down 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 and bit-field 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
  Interprocedural optimizations
  Literal synthesis optimization
  Local data area optimization
  Profiling feedback optimization

Portability

The compiler implements the ANSI C++ standard (ISO/IEC FDIS 14882) as described in *13. C++ Features and Compatibility*. Exceptions, templates, and run-time type Information (RTTI) are fully implemented.

For C modules, the compiler conforms fully to the ANSI X3.159-1989 standard (called ANSI C), with extensions for compatibility with other compilers to simplify porting of legacy 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. See *BCompatibility Modes: ANSI, PCC, and K&R C*, p.559 for operational details when compiling in different modes.

Wind River tools produce identical binary output regardless of the host platform on which they run. The only exceptions occur when symbolic debugger information is generated (that is, when **-g** options are enabled), since path information differs from one build environment to another.

1.3 Documentation

This User's Guide

This guide contains all information necessary to use the tools effectively. Please see the table of contents for a detailed overview.

Table 1-1 **User's Guide Parts**

Part	Contents
Part I. Introduction	Overview, configuration, directory structure, subprograms, selecting a target for compilation.
Part II. Wind River Compiler	The compilers, including invocation, options, additions to C and C++ for device programming, internal data representation, calling conventions, and optimizations.
Part III. Wind River Assembler	The assembler, including invocation, options, syntax rules, expression syntax, and all assembler directives. See manufacturer's manuals for details on ARM instructions.
Part IV. Wind River Linker	The linker, including invocation, options, the linker command language, and object module format.
Part V. Wind River Compiler Utilities	The **D-AR** library archiver; the **D-DUMP** utility for converting and examining object, executable, and archive files; and others.
Part VI. C Library	The structure of the C libraries provided with the compiler for use in different environments, and the details of the functions in the libraries.
Part VII. Appendices	Configuration files, limits, implementation-defined behavior, assembler coding notes, object modules format details, **-X** options by number, and messages.

This manual does not explain the C or C++ language. See*Additional Documentation*, p.8 below, for references to standard works.

Additional Documentation

Changes made for this release and information developed after publication of this manual may be found in the release notes.

The following C++ references are recommended: the ANSI C++ standard (ISO/IEC FDIS 14882), *The C++ Programming Language* by Bjarne Stroustrup, *The Annotated C++ Reference Manual* by Margaret A. Ellis and Bjarne Stroustrup, and *The C++ Standard Template Library* by P.J. Plauger et al.

For C, see the ANSI C standard *X3.159-1989* and *The C Programming Language* by Brian Kernighan and Dennis Ritchie (K&R).

The following manuals from ARM Limited may be consulted for details about microprocessor architecture and instructions:

- *ARM Architecture Reference Manual*
- *ARM Developer Guide*

2

Configuration and Directory Structure

2.1 Components and Directories

All files are located 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 pathname of the root directory. The root directory contains *version_path* 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 of the complete path 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. **Win32** or **SUNS** (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. The following table assumes that the version number is 5.3.*x* and shows examples for

common installations. For other systems, see the installation procedures shipped with the media.

Table 2-1 **Example Default Installation Pathnames**

System	Default *version_path*	Default with *host_dir*
UNIX	**/usr/lib/diab/5.3.***x*	**/usr/lib/diab/5.3.***x***/***host*
HP-UX		**/usr/lib/diab/5.3.***x***/HPUX**
Solaris		**/usr/lib/diab/5.3.***x***/SUNS**
Linux		**/usr/lib/diab/5.3.***x***/LINUX386**
PCs	**C:\diab\5.3.***x*	**C:\diab\5.3.***x******op-sys*
Windows		**C:\diab\5.3.***x***\WIN32**

NOTE: In this manual, instructions and examples for Windows apply to all supported versions of Microsoft Windows.

Also, in cases where the Windows and UNIX pathnames are identical except for the path separator character, only one pathname is shown using the UNIX separator "/".

The following table 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
Programs:	
*host_dir***/bin/**	Programs intended for direct use by the user:
dcc	Main driver—assumes C libraries and headers.
dplus	Main driver—assumes C++ libraries and headers.
das	The assembler. A separate ARM-specific description file controls assembly.
dld	The linker. Generates executable files from one or more object files and object libraries (archives).

Table 2-2 **Version_path Subdirectories and Important Files** (cont'd)

Subdirectory or File	Contents or Use
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.**
dctrl	Utility to set default target for compiler, assembler, and linker.
ddump	**D-DUMP** object file utility. Examines or converts object files, e.g. ELF to Motorola S-Records.
dmake	"make" utility; extended features are required to re-build the libraries. Not for use with VxWorks development tools.
flexlm* **lm***	Programs and files for the license manager used by all Wind River tools.
reorder	This program is started by the driver. It reschedules the instruction sequence to avoid stalls in the processor pipeline and does some peephole optimizations. See *10. Optimization*.
host_dir/**lib/**	Programs and files used by programs in **bin**.
ctoa	C and C++ compilers. A separate ARM-specific description file directs code generation. (The preferred C++ compiler is **etoa**; **dtoa** is an older version.)
etoa, **dtoa**	
Configuration, header, and source files	
conf/	Configuration files for compilers, assembler, and linker.
dtools.conf **default.conf** **user.conf**	Configuration files read by the compiler drivers at startup, primarily to supply command-line options. See *A. Configuration Files* for details. Other **.conf** files for particular boards or operating systems may also be present.
default.dld	Default linker command file. Other sample **.dld** linker command files are also found here. See *24.2 Defaults*, p.360 in the Linker section of this manual.
dmake/	**dmake** startup files. See *30. dmake Makefile Utility*.
example/	Example files used in the *Getting Started* manual and elsewhere.

Table 2-2 **Version_path Subdirectories and Important Files** (cont'd)

Subdirectory or File	Contents or Use
include/	Standard and other header files for use in user programs, plus HP/SGI STL library header files.
libraries/	Library sources and build files. See *32.3 Library Sources, Rebuilding the Libraries*, p.453 for details.
pdf/	PDF form for all manuals.
relhist/	Older Release Notes.
src/	Source code for replacement routines for system calls. These functions must be modified before they can be used in an embedded environment. See *15. Use in an Embedded Environment*.
ARM startup module and libraries	
ARME/	ELF library and startup code directories (big-endian).
crt0.o	Start up code to initialize the environment and then call **main**. The source for **crt0.o** is **src/crtarm/crt0.s**.
libc.a **cross/libc.a** **simple/libc.a**	ELF standard C libraries. Each **libc.a** is actually a short text file of **-l** options listing other libraries to be included. A **libc.a** file is selected based on the library search path (See *4.2 Selected Startup Module and Libraries*, p.24). **ARME/libc.a** is a generic C library with no input/output support. It includes sublibraries **libi.a**, **libcfp.a**, **libimpl.a**, **libimpfp.a**, all described below. **ARME/simple/libc.a** includes the above four sublibraries plus **libchar.a** providing basic character I/O. **ARME/cross/libc.a** includes the above four sublibraries plus **libram.a**, which adds RAM-disk-based file I/O. For details, see *32.2 Library Structure*, p.444.
libchar.a	Basic character input/output support for **stdin** and **stdout** (**stderr** and named files are not supported); an alternative to **libram.a**.
libram.a	Adds to **libchar.a** RAM-disk-based file I/O for **stdin** and **stdout** only; an alternative to **libchar.a**.

Table 2-2 **Version_path Subdirectories and Important Files** (cont'd)

Subdirectory or File	Contents or Use
libi.a	General library containing standard ANSI C functions.
libimpl.a	Utility functions called by compiler generated or runtime code, typically for constructs not implemented in hardware, e.g., low-level software floating point support, multiple register save and restore, and 64-bit integer support.
libd.a	Additional standard library functions for C++ (**libc.a** is also required).
libg.a	Functions to generate debug information for some debug targets.
windiss/libwindiss.a	Support library for Wind ISS instruction-set simulator when supplied. Note: implicitly also uses **cross/libc.a**.
Floating point-specific libraries and sub-libraries	
ARMEN	ELF floating point stubs for floating point support of "None".
libcfp.a	Stubs to avoid undefined externals.
libimpfp.a	Empty file required by different versions of **libc.a**.
libstl.a, **libstlstd.a**	Support library for C++. Includes **iostream** and complex math classes.
ARMES/	ELF software floating point libraries:
libcfp.a	Floating point functions called by user code.
libcomplex.a	C++ complex math class library.
libimpfp.a	Conversions between floating point and other types.
libios.a	C++ **iostream** class library.
libm.a	Math library.
libpthread.a	Unsupported implementation of POSIX threads for use with the example programs. Text file which includes sub-libraries **libdk*.a**.
ARMEV/	ELF vector floating point libraries:

Table 2-2 **Version_path Subdirectories and Important Files** (cont'd)

Subdirectory or File	Contents or Use
ARML*f*/	Parallel little-endian ELF library and startup code directories.
ARMT*of*/	Parallel directories for the Thumb instruction set.
ARMX*of*/	Parallel directories for Xscale.

2.2 Accessing Current and Other Versions of the Tools

The driver **(dcc** or **dplus**) automatically finds the subprograms it calls (it is modified with the directory selected during installation). Thus, running the compiler requires only that driver be accessed in any of the usual ways:

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

If the tools are installed on a remote server, Windows users should map a drive letter to the remote directory where they reside and use that drive letter when setting their path variable.

You can invoke an older copy of a driver as follows:

- Rename the main driver for the older version. For example, to execute version 4.4a of the C++ driver, rename **dplus** in the **bin** directory for version 4.4a **dplus44a**. Then access **dplus44a** in any of the usual ways described above.
- Modify your path to put the directory containing the desired version before the directory containing any other version. The driver command will then access the desired version.
- Create an alias or batch file that includes the complete path of the desired version.

2.3 Environment Variables

NOTE: This section is for unusual cases. It is usually sufficient to override the default setting by using the **-t** option on a command line when invoking a tool, or to use one of the other methods, all as described under *4.3 Alternatives for Selecting a Target Configuration*, p.25.

The configuration information which controls default operation of the tools is usually stored as *configuration variables* in **default.conf** in the **conf** subdirectory of the *version_path* directory by the **dctrl** program. These configuration variables include **DTARGET**, **DFP**, **DOBJECT**, and **DENVIRON**. However, if an environment variable having the same name as a configuration variable is set, the value of the environment variable will override the value stored in **default.conf**. (This can in turn be overridden by using a **-t** or **-WD** option on the command line when invoking a tool.)

The method used to set environment variables depends on the operating system as shown in the following table.

Table 2-3 **Setting Environment Variables**

System	Command
UNIX	*variable*`=`*value*`;export` *variable*
Windows	`set` *variable*`=`*value*

2.3.1 Environment Variables Recognized by the Compiler

This section describes the environment variables recognized by the compiler.

DCONFIG

Specifies the configuration file used to define the default behavior of the tools. documents the configuration file. If neither **DCONFIG** nor the **-WC** option is used (see *A.2.2 Startup*, p.547), the drivers use:

version_path`/conf/dtools.conf` (UNIX)
`%`*version_path*`%\conf\dtools.conf` (Windows)

DTARGET
DOBJECT

DFP

DENVIRON

These four environment variables specify, respectively, the target processor, object file format and mnemonic type, floating point method, and execution environment. They may be used to override the values set in **default.conf** (and will in turn be overridden by a **-t** option on the command line). **DENVIRON** may also refer to an additional configuration file, for example to set options for a particular target operating system. For details, see:

- *4.3 Alternatives for Selecting a Target Configuration*, p.25.
- *4.1 Selecting a Target*, p.21 for valid settings for the four variables.
- *A.3.1 DENVIRON Configuration Variable*, p.549 regarding **DENVIRON**.

DFLAGS

Specifies extra options for the drivers and is a convenient way to specify **-XO**, **-O** or other options with an environment variable (e.g., to avoid changing several makefiles or to override options given in a configuration file). The options in **DFLAGS** are evaluated before the options given on the command line. See *A.3 Standard Configuration Files*, p.548, especially Figure A-2 for details.

DIABLIB

Formerly used to tell the compiler and drivers where to look for the tools. If **DIABLIB** is defined, it should be set to the *version_path* selected during installation. If **DIABLIB** is not defined, the compiler and drivers are found on the user's path variable or from an absolute directory path specified on the command line.

NOTE: DIABLIB is deprecated and is maintained for backward compatibility only.

DIABTMPDIR

Specifies the directory for all temporary files generated by all tools in the tool suite.

DCXXOLD

If set to **YES**, tells the compiler to use the old C++ parser (**-Xc++-old** option) by default.

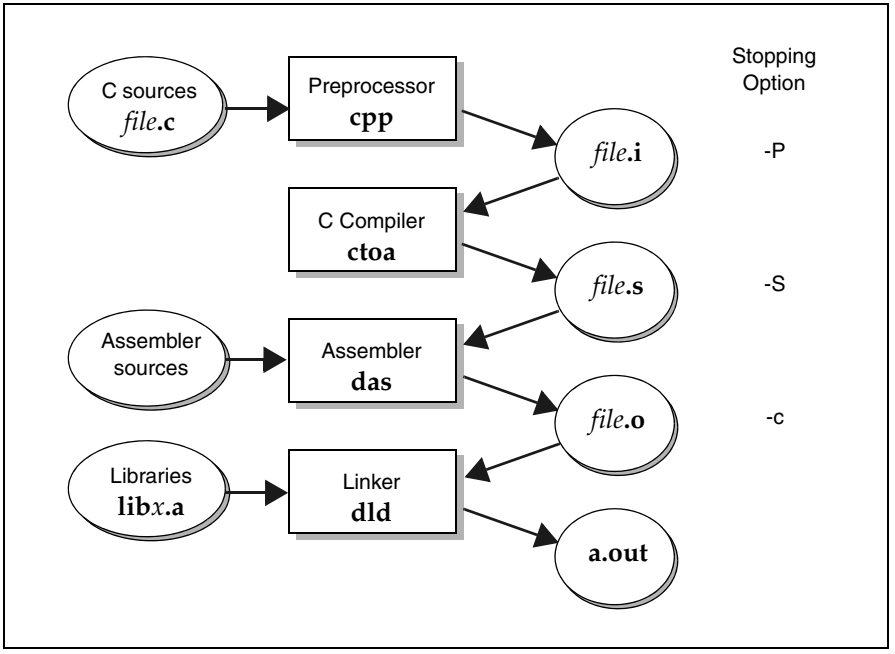
3

Drivers and Subprogram Flow

The Wind River tools are most easily invoked using the **dcc** and **dplus** *driver programs*. Depending on the input files and options on the command line, the driver may run up to five subprograms: the C preprocessor, either or both compilers, the assembler, and the linker.

The following figure shows the subprogram flow graphically for a C file. A C++ file is processed similarly except **dplus** invokes the C++ **etoa** compiler instead of **ctoa**. The subprograms and the *stopping options* are described following the figure.

Figure 3-1 **Subprogram Flow and Intermediate Files**



Driver command lines are described in detail in *5. Invoking the Compiler*. The general form is:

dcc	[*options*] [*input-files*]	Assumes Wind River C libraries.
dplus	[*options*] [*input-files*]	Assumes Wind River C++ libraries.

The driver determines the starting subprogram to be applied to each *input-file* based on the file's extension suffix; for example, by default a file with extension **.s** is assembled and linked but not preprocessed or compiled. Command-line options may be used to stop processing early. The subprograms and stopping options are as follows.

Table 3-1 **Driver Subprograms, Default Input and Output Extensions, and Stopping Options**

Sub-program	Default Input Extension		Stopping Option	Default Output Extension	Function and Stopping Option
`cpp`			`-P`	`.i`	The preprocessor; takes a C or C++ module as input and processes all **#** directives. This program is included in the main compiler program. The **-P** option halts the driver after this phase, producing a file with the **.i** suffix. (The **.i** file is not produced unless **-P** is used.)
`ctoa`	`.c`		`-S`	`.s`	The C-to-assembly compiler; consists of several internal stages (parser, optimizer, and code generator), and generates assembly source from preprocessed C source.

`etoa`	`.cpp`
	`.cxx`
	`.cc`
	`.C`
		(capital, UNIX)	`-S`	`.s`	The C++-to-assembly compiler; generates assembly source from preprocessed C++ source.

`das`	`.s`		`-c`	`.o`	The assembler; generates linkable object code from assembly source.
`dld`	`.o`	(object)		`a.out` (default)	The linker; generates an executable file from one or more object files and object libraries, as directed by a **.dld** linker command file (obsolete: **.lnk**). The default output name is **a.out** if the **-o** *outputfile* option is not given.
	`.a`	(archive)			
	`.dld` `.lnk`	(commands)			

4

Selecting a Target and Its Components

4.1 Selecting a Target

The compiler, assembler, and linker all require specification of a *target configuration*.

A complete target configuration specifies the target processor, the type of floating point support, the object module format (ELF), and the execution environment (default libraries for input/output and target operating system support). To determine the current default, execute the command:

```
dcc -Xshow-target
```

or print the file **default.conf** in the *version_path*/**conf** subdirectory.

The easiest methods for selecting a target configuration are as follows. The first method is preferred. For special cases or more details, see *4.3 Alternatives for Selecting a Target Configuration*, p.25.

- Use the **-t***tof* or **-t***tof***:***environ* option when invoking the compiler, assembler, or linker. The table below describes this option.
- Invoke the **dctrl** command with the **-t** option to set the defaults used when no **-t** option is present on the compiler, assembler, or linker command line. Note that this sets the default for all users.

The *tof***:***environ* string given with the **-t** option has four parts, as follows. See *4.2 Selected Startup Module and Libraries*, p.24 for examples.

Table 4-1 **-t Option Values**

t — Target processor, a several-character code — *see the Notes following the table* (sets **DTARGET**):

ARM	ARM
ARMV5	ARMV5
ARMT	ARMT incorporates the Thumb instruction set enabling a 16-bit mode
ARMV5T	ARMV5T incorporates the Thumb instruction set enabling a 16-bit mode
ARMX	Intel XScale
ARMXT	ARMXT incorporates the Thumb instruction set enabling a 16-bit mode

o — Object format (sets **DOBJECT**):

E for ELF big-endian

L for ELF little-endian

f — Floating point support — one character (sets **DFP**):

S for Software floating point emulation provided with the compiler — default on targets without internal floating point.

V for Vector floating point support. The compiler assumes that the VFP unit has been initialized with stride=1. See the Note below.

N for No floating point support (minimizes the required runtime).

Table 4-1 **-t Option Values** (cont'd)

environ	Execution environment (sets **DENVIRON**). Determines paths searched for libraries (see *4.2 Selected Startup Module and Libraries*, p.24). Two standard values used with the libraries delivered with the tools are: **cross** to include **libram.a** for RAM-disk input/output **simple** to include **libchar.a** for basic character input/output *environ* may also be the name of a target operating system supported by Wind River. In this case, in addition to specifying the library search path, the value will be used to include a special configuration file, *environ***.conf** in the **conf** subdirectory, to set options required by the target operating system. For further details, see *A.3.1 DENVIRON Configuration Variable*, p.549, *VxWorks Application Development*, p.24, and the release notes and available application notes for particular target operating systems. *environ* is optional. If not given by **-t**, a **-WDDENVIRON** option, or a **DENVIRON** environment variable, the value set by **dctrl** is used.

Notes for the Target Processor Component of the -t Option

- In the **-t***tof* option, "*t*" is the part not including the final two parts, each of which is always a single character (the *o* and *f* parts).
- Each target in the table which is not preceded by an **"="** sign causes the invoked tool to operate in a manner unique to that target. The unique operating characteristics are selected via the options used to invoke the tool plus the options which the tool extracts from the built-in configuration files.

  To see the options associated with a particular **-t** option, invoke a compiler driver with the **-t** option, the **-# option (causes the driver to show the command line used to invoke each tool), and the -Wa, -#** option (causes the assembler, when invoked by the driver, to show options which it extracts from the configuration files).
- Each implemented ARM DMA channel has its own register for control of DMA operations. Part of the register format includes the *stride*. The stride indicates the increment on the external address between each consecutive access of the DMA; a stride of zero indicates that the external address should not be incremented. By default, the Wind River Compiler for ARM/XScale sets the stride to 1; adjust your startup code appropriately if necessary.

- This table may not be up-to-date. Invoke **dctrl -t** to construct any valid **-t** option supported by the tools as installed, or look in **ARM.conf** for a complete list of target processor codes.

VxWorks Application Development

To build VxWorks applications, specify the appropriate execution environment with the **-t** option. Usually this will be **:rtp** for user (real-time process) mode or **:vxworks***x.x* for kernel mode. For example, -tARME**N:rtp** selects user mode, while -tARME**N:vxworks6.2** selects VxWorks 6.2 kernel mode. For more information, see the documentation that accompanied your VxWorks development tools.

NOTE: If you specify a VxWorks execution environment (**:rtp** or **:vxworks***x.x*), the standard C libraries linked to your application will be different from the compiler's native C libraries documented in this manual.

Specifying a VxWorks execution environment turns on **-Xieee754-pedantic** by default.

4.2 Selected Startup Module and Libraries

The parts of **-t***tof***:***environ* option (or its equivalents as described in *4.1 Selecting a Target*, p.21) are used to construct a directory name and to select the desired startup module and libraries per Table 4-1.

Examples:

-t Option	Startup Module, Libraries
`-tARMEN:simple`	**ARME/crt0.o** **ARME/simple/libc.a** with **ARMEN/libcfp.a** and **ARME/libchar.a** ARM, ELF objects, no floating point, character input/output

-t Option	Startup Module, Libraries
`-tARMES:cross`	**ARME/crt0.o** **ARME/cross/libc.a** with **ARMES/libcfp.a** and **ARME/libram.a** ARM, ELF objects, software floating point, RAM-disk input/output

The library archive files themselves, and the detailed mechanics for selection of the appropriate subdirectories and libraries, are fully described in *32.2 Library Structure*, p.444.

Briefly, the main driver programs select the startup module and libraries by invoking the linker with the following partial command line, using UNIX path notation, written on multiple lines and spaced for readability, and where *f* is as described above:

```
dld -Y P,version_path/ARMEf/environ : version_path/ARMEf :
          version_path/ARME/environ : version_path/ARME ...
-l:crt0.o ... -lc
```

The **-Y P** option sets a list of directories. Then the **-l:crt0.o** option causes the linker to look in those directories for file **crt0.o**, the startup file, without modification, while the **-lc** option causes the linker to construct filename **libc.a** and look in those directories for it.

4.3 Alternatives for Selecting a Target Configuration

There are five ways to change the target configuration. *As noted at the beginning of this chapter, the first method is preferred, especially when multiple engineers work with multiple targets.* This section is provided for backward compatibility and special cases.

Using **-t** sets four *configuration variables*: **DTARGET** for the processor, **DOBJECT** for the object module format, **DFP** for the type of floating point support, and **DENVIRON** for the target execution environment.

These configuration variables are stored in *version_path*/**conf/default.conf**. A configuration variable may be overridden by an environment variable of the same name, or by a **-t** or **-WD** *variable* option on the command used to invoke the

compiler, assembler, or linker. The environment variable is checked first and then the command line; the last instance found is used.

Change the target for a single invocation of a tool by using the **-t** option on the command line; this applies to **dcc**, **dplus**, **das**, and **dld**. The **-t** option takes one of the *tof* or *tof*:*environ* codes described in *4.1 Selecting a Target*, p.21 and displayed by the **dctrl -t** program (see below).

Example:

```
dplus -ttof -c file.cpp
```

Other methods involve changing or overriding four configuration variables stored in the configuration file **default.conf**. (See *A.3 Standard Configuration Files*, p.548.)

- 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 prompts you for the desired target processor, object format, floating point support, and target execution environment. If you already know the exact target configuration you want, you can skip the interactive program by specifying the target after **-t** on the command line:

  ```
  dctrl -ttof:environ
  ```

  Upon success, **dctrl** displays the new default target and modifies **default.conf**.

- Manually edit the **default.conf** configuration file to change the default settings for any of the **DTARGET** (the processor), **DOBJECT** (object module format), **DFP** (floating point support), and **DENVIRON** (target execution environment) configuration variables.

- Set any of the **DTARGET**, **DFP**, **DOBJECT**, and **DENVIRON** environment variables. This overrides the values of the configuration variables having these names in **default.conf**.

- Use the command-line option **-WD** *environment_variable* (see *5.3.26 Define Configuration Variable (-W Dname=value)*, p.42). This overrides both the values of the variables in **default.conf** and any environment variables. Example:

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

NOTE: For additional explanation, and order of precedence when more than one of these methods is used, See *A. Configuration Files*, and especially *A.2.1 Configuration Variables and Precedence*, p.546.

PART II

Wind River Compiler

5 Invoking the Compiler

5.1 The Command Line

As noted in *3. Drivers and Subprogram Flow*, the compiler is best executed via one of the driver programs as follows:

`dcc`	[*options*]	[*input-files*]	Assumes Wind River C libraries.
`dplus`	[*options*]	[*input-files*]	Assumes Wind River C++ libraries.

where:

`dcc`
`dplus`
: Invokes the main driver program for the compiler suite. See *2.2 Accessing Current and Other Versions of the Tools*, p.14 for details on how the driver program is found.

Both the **dcc** and **dplus** drivers are used in examples this manual. Please substitute **dcc** for **dplus** if you are using only the C compiler.

options
Command-line options which change the behavior of the tools. See the remainder of this chapter for details. Options and filenames may occur in any order.

input-files
A list of pathnames, each specifying a file, separated by whitespace. The suffix of each filename indicates to the driver which actions to take. See Table 3-1 for details.

For example, process a single C++ file, stopping after compilation, with standard optimization:

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

The form **-@***name* can also be used for either *options* or *input-files*. The name must be that of an environment variable or file (a path is allowed), the contents of which replace **-@***name*. See *A.2 How Commands, Environment Variables, and Configuration Files Relate*, p.546 for details.

5.2 Rules for Writing Command-Line Options

Same Option More Than Once

Options can come from several sources: the command line, environment variables, configuration files, and so forth as described in *A.2 How Commands, Environment Variables, and Configuration Files Relate*, p.546.

If an option appears more than once from whatever source, the final instance is taken unless noted otherwise in the individual option descriptions in the next sections.

Command-Line Options are Case-sensitive

Command-line options are case-sensitive. For example, **-c** and **-C** are two unrelated options. This is true even on Windows; however filenames on Windows remain case-insensitive as usual.

Spaces In Command-Line Options

For easier reading, command-line options may be shown with embedded spaces in documentation, although they are not typically written this way in use. In writing options on the command line, space is allowed only following the option letter, not elsewhere. For example:

```
-D DEBUG=2
```

is valid, and is exactly equivalent to:

```
-DDEBUG=2
```

However,

```
-D DEBUG = 2
```

is not valid because of the spaces around the "**=**".

Quoting Values

When a command-line option can take a string as a value, it does *not* require quotes. For example:

```
-prof-feedback=rta-db  -Xname-code=.code
```

Enclosing the value in quotes has no effect. Thus,

```
-DSTRING="test"
```

is equivalent to:

```
-DSTRING=test
```

Using "\" to escape the quotes will pass the quotes into the compiler. Given file **test.c** containing:

```
void main() {
    printf(STRING);
}
```

compiling with:

```
dcc test.c -DSTRING="test"
```

the **printf** statement becomes:

```
printf( test );
```

(and will fail because **test** is undefined). But compiled with:

```
dcc test.c -DSTRING=\"test\"
```

the **printf** statement becomes:

```
printf( "test" );
```

Unrecognized Options, Passing Options to the Assembler or Linker

Ordinary options beginning with a letter other than "**X**" and which are not listed in this section are automatically passed by the driver to the linker. All **-X** options are processed first by the compiler.

When invoking the **dcc** or **dplus** driver program, it is sometimes important to pass an option explicitly to the assembler or linker—for example, a **-X** option or an option identified by the same letter as a driver or compiler option. The driver options **-W a,***arguments* and **-W l,***arguments* pass *arguments* to the assembler and linker respectively.

Length Limit

The length of the command line is limited by the drivers' 1000-byte internal buffer. To pass longer commands to the tools, see *5.3.39 Read Command-Line Options from File or Variable (-@name, -@@name)*, p.48.

The following example is written on several lines for clarity. The individual options shown are fully documented in this chapter or in the *16.4 Assembler -X Options*, p.283 and in *24.5 Linker -X options*, p.369.

```
dcc -D DEBUG=2 -XO
    -Wa,-DDEBUG=3
    -Wl,-Xdont-die
    -Llibs
    -WA.asm
    f.c a.asm
```

`-D DEBUG=2 -XO`
: The driver invokes the compiler with these options. A space is allowed after the option letter **-D**.

`-Wa,-DDEBUG=3`
The driver invokes the assembler with the option **-DDEBUG=3**, perhaps for use in the **a.asm** file. Without the -Wa, the driver would have passed this option to the compiler, resetting **DEBUG** to 3.

No space is allowed after the **-D** because it would have ended the **-Wa** option; **-W a**, **-DDEBUG=3** would also have been valid.

`-Wl,-Xdont-die`
The driver invokes the linker with the option **-Xdont-die**. Without the **-Wl**, the driver would have passed this linker option **-Xdont-die** to the compiler.

`-Llibs`
This option is not recognized by the driver as a driver or compiler option, so it is passed to the linker.

`-WA.asm`
Instructs the driver that files having the extension **.asm** are to be preprocessed and then assembled. If this extension is a project standard, it can more conveniently be set in user configuration file **user.conf** as follows (see *A.3.2 UFLAGS1, UFLAGS2, DFLAGS Configuration Variables*, p.551):

```
UFLAGS1=-WA.asm
```

`f.c a.asm`
An input file to be compiled (**f.c**) and, because of the **-WaA.asm** option, an input file to be preprocessed and assembled (**a.asm**).

The next sections document the command-line options recognized by the driver and compiler.

5.3 Compiler Command-Line Options

This section shows all general command-line options. New options added after publication may also be in the most recent release notes.

5.3.1 Show Information About Compiler Options (-?, -?..., -h, -h..., --help)

-?
-h
--help
Show synopsis of commonly used compiler options. Available for other tools (assembler, linker) as well.

-??
-h?
Show synopsis of less frequently used options.

-?W
-hW
Show synopsis of **-W** options (see *5.3.25 Pass Arguments to the Assembler (-W a,arguments, -W :as:,arguments)*, p.42).

-?X
-hX
Show synopsis of **-X** options (see *5.4 Compiler -X Options*, p.48).

-?X*string*
Show synopsis of **-X** options whose names contain the specified string. For example, entering **dcc -?Xbss** returns information about **-Xbss-off** and **-Xbss-common-off**.

5.3.2 Ignore Predefined Macros and Assertions (-A-)

-A-
Cause the preprocessor to ignore all predefined macros and assertions.

5.3.3 Define Assertion (-A assertion)

-A *pred***(***ident1***)** **(***ident2***)**
Cause the assertion *pred***(***ident***)** to be defined. See *#assert and #unassert Preprocessor Directives*, p.120.

5.3.4 Pass Along Comments (-C)

-C
Cause the C processor to pass along all comments. Useful only in conjunction with **-E** or **-P**.

NOTE: The preprocessor may be used with any language supported by Wind River.

-C is not necessary when **-Xpass-source** is used to output source as comments when generating assembly output because in that case the source code is taken before preprocessing.

5.3.5 Stop After Assembly, Produce Object (-c)

-c

Stop after the assembly step and produce an object file with default file extension **.o** (unless modified by **-o**, see *5.3.18 Specify Output File (-o file)*, p.40).

5.3.6 Define Preprocessor Macro Name (-D name=definition)

-D *name* [**=***definition*]

Define the preprocessor macro *name* as if by the **#define** directive. If no *definition* is given, the value 1 is used.

Macros may be either *function-like* macros or *object-like* macros. Function-like macros take arguments; this sample macro converts inches to centimeters:

```
dcc -DIN_TO_CM(x)=((x)*2.54) foo.c
```

Note that, to prevent unexpected results, both the argument and the entire macro expression should be enclosed in parentheses.

Object macros do not take arguments:

```
dcc -DYEAR_LENGTH=366 bar.c
```

See *5.2 Rules for Writing Command-Line Options*, p.30, for rules about using spaces, quotations, and the like on the command line.

5.3.7 Stop After Preprocessor, Write Source to Standard Output (-E)

-E

Run only the 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. (To suppress line-number information, use

-Xpreprocessor-lineno-off.) The source files do not require any particular suffix.

When **-E** is invoked, the preprocessor implicitly includes the **lpragma.h** file. To suppress inclusion of **lpragma.h**, use **-Xclib-optim-off**. For more on **lpragma.h**, see *5.4.22 Disregard ANSI C Library Functions (-Xclib-optim-off)*, p.64.

See also *5.3.19 Stop After Preprocessor, Produce Source (-P)*, p.41.

5.3.8 Change Diagnostic Severity Level (-e)

-e*sn*[,*n*...]

For each of one or more diagnostic message numbers *n* in the comma-separated list, change the severity level of the message to *s* where *s* is one of:

i
: Information, equivalent to ignore.

w
: Warning.

e
: Error (continue compilation).

f
: Fatal error (terminate immediately).

Each diagnostic message has the form:

"file", line #: *severity-level* (*compiler*:*error* #): *message*

Example:

```
"err1.c", line 2: warning (dcc:1025): division by zero
```

To raise the severity level of this message from "warning" to "error", invoke the compiler with the option **-ee1025**. To reduce the level to "ignore", use **-ei1025**.

NOTE: Some messages have a minimum severity level. The severity level of a message may be raised above its minimum but not lowered below it. Attempting to do so will generate warning 1641.

NOTE: -Xmismatch-warning and **-Xmismatch-warning=2** override the **-e** option. If either form of **-Xmismatch-warning** is used, mismatched types will only produce a warning, even if **-e** is used to increase the severity level of the diagnostic. See *5.4.92 Warn On Type and Argument Mismatch (-Xmismatch-warning)*, p.94.

5.3.9 Generate Symbolic Debugger Information (-g)

The several **-g***n* options enable generation of varying levels of debugging information. If optimization options are also present (**-O** or **-XO**), optimization will be affected as described.

`-g`
Same as **-g2**.

`-g0`
Do not generate symbolic debugger information. This is the default. No effect on optimization.

`-g1`
Generate symbolic debugger information, but leave out line number information. No effect on optimization.

`-g2`
Generate symbolic debugger information.

Do most target-independent optimizations, but do not do the following optimizations, since most object formats have no way to describe them. Hexadecimal numbers indicate the mask for **-Xkill-opt (***5.4.79 Disable Individual Optimizations (-Xkill-opt=mask, -Xkill-reorder=mask)*, p.87**)**.

- Function inlining (*Inlining (0x4)*, p.191)
- Structure member optimization (*Structure Members to Registers (0x10)*, p.193)
- Split optimization (*Variable Live Range Optimization (0x400)*, p.195)
- *Complex Branch Optimization (0x1000)*, p.196
- *Loop Count-Down Optimization (0x4000)*, p.197
- *Minor Transformations to Simplify Code Generation (0x80000)*, p.198
- *Live-Variable Analysis (0x40000000)*, p.201

Also, disable most target-dependent optimizations: option **-g2** also disables basic reordering and all peephole optimizations (see 203).

See *10. Optimization* for details on these optimizations (the optimizations are ordered by the hex values in that chapter).

See also **-Xoptimized-debug-off** (*5.4.98 Disable Most Optimizations With -g (-Xoptimized-debug-...)*, p.97) on how to disable optimizations which interfere with debugging.

g3

Generate symbolic debugger information and do all optimizations. Highly optimized code can be difficult to debug. For example, there is no way to break on inlined functions (except at the assembly level). Hence, when debugging is required, **-g2** is usually a better choice.

NOTE: The **-g***n* options may also be specified at the beginning of a source files using:

```
#pragma option -gn
```

5.3.10 Print Pathnames of Header Files (-H)

-H

Print the pathnames of all header files to the standard error output.

5.3.11 Specify Directory for Header Files (-I dir)

-I *dir*

Add *dir* to the list of directories to be searched for header files. A full pathname is allowed. More than one **-I** option can be given.

For an **#include** "*file*" directive, search for the file in the following locations:

- First, the directory of the file containing the **include** directive. Thus, if an **#include** directive includes a path, that path defines the current directory for **#include** directives in the included file. Example (using UNIX notation):

  Assume file **f1.c** contains:

  ```
  #include "p1/h1.h"
  #include "h3.h"
  ```

  and file **h1.h** contains:

  ```
  #include "h2.h"
  ```

  The search for **h2.h** will begin in directory **p1**; the search for **h3.h** will begin in the directory containing **f1.c**.

- Second, directories given by the **-I** *dir* option, in the order encountered.
- Third, the directory given by either:

  any **-Y I** option appearing prior to the **-I** option

  – or –

  version_path`/include` (`UNIX`)
  version_path`\include` (`Windows`)

  (The **-Y I** option effectively replaces the *version_path* directory.)

For an **#include <***file***>** directive, search only the second and third locations.

5.3.12 Control Search for User-Defined Header Files (-I@)

`-I@`

C only. Search for user-defined header files (those enclosed in double quotes (") in the order specified only by **-I** options (modified by **-Y I** options if any). That is, do not search the current directory by default; search the current directory only when an **-I@** option is encountered. Example:

```
dcc -Iabc -I@ -Idef file.c
```

will result in a search order of:

the directory **abc**
the current directory
the directory **def**

5.3.13 Modify Header File Processing (-i file1=file2)

`-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.

The **-i** option is disabled by **-P**.

5.3.14 Specify Directory For -l Search List (-L dir)

This is a linker option. See *Specify Directory for -l search List (-L dir)*, p.365.

5.3.15 Specify Library or Process File (-l name)

This is a linker option. See *Specify Library or File to Process (-lname, -l:filename)*, p.365.

5.3.16 Specify Pathname of Target-Spec File (-M target-spec)

-M *target-spec*

NOTE: This option is primarily for use by Wind River.

Specify the pathname of the *target-spec* file to the compiler (see **target.cd** in Table 2-2). This file contains the target description and is read by the compiler at startup. If the **-M** option is set more than once, the final setting is used.

5.3.17 Optimize Code (-O)

-O

Optimize code. Either this or **-XO** must be present to enable optimization and to invoke the **reorder** program. See the **-XO** option in *5.4.95 Enable Extra Optimizations (-XO)*, p.96 for the difference between these options and *10. Optimization* for more information about optimizations.

This option can also be specified at the beginning of a source file using:

```
#pragma option -O
```

5.3.18 Specify Output File (-o file)

-o *file*

Output to the given file instead of the default. This option works with the **-P**, **-S** and **-c** options as well as when none of these are specified. When compiling *file.ext* the following filenames are used by default if the **-o** option is not given:

-P	*file*.**i**
-S	*file*.**s**
-c	*file*.**o**

```
not -P, -S, or -c       a.out
```

5.3.19 Stop After Preprocessor, Produce Source (-P)

-P

Stop after the preprocessor step and produce a source file with default file extension **.i** (unless modified by **-o**).

Unlike with the **-E** option, the output will not contain any preprocessing directives, and the output does not go to standard out (see **-o** for the output filename). The source files do not require any particular suffix.

When this option is used, the compiler driver does not invoke the assembler or linker. Thus, any switches intended for the assembler or linker must be given separately on command lines which invoke them. The **-P** option also disables **-i**.

When **-P** is invoked, the preprocessor implicitly includes the **lpragma.h** file. To suppress inclusion of **lpragma.h**, use **-Xclib-optim-off**. For more on **lpragma.h**, see *5.4.22 Disregard ANSI C Library Functions (-Xclib-optim-off)*, p.64.

5.3.20 Stop After Compilation, Produce Assembly (-S)

-S

Stop after the compilation step and produce an assembly source code file with the default file extension **.s** (unless modified by **-o**). If **-Xshow-configuration=1** is enabled, the assembly file contains a list of options in effect during compilation.

5.3.21 Select the Target Processor (-t tof:environ)

-t *tof*:*environ*

Select the target processor with *t* (a several character code), the object format with **o** (a one letter code), the floating point support with *f* (**H** for hardware, **S** for software, and **N** for none), and libraries suitable for the target environment with *environ*.

To determine the proper *tof*, execute **dctrl -t** to interactively display all valid combinations. See also *4.2 Selected Startup Module and Libraries*, p.24.

5.3.22 Undefine Preprocessor Macro Name (-U name)

-U name
Undefine the preprocessor macro *name* as if by the **#undef** directive.

5.3.23 Display Current Version Number (-V, -VV)

-V
Display the current version number of the driver.

-VV
Display the current version number of the driver and the version number of all subprograms. Do not complete the compilation.

5.3.24 Run Driver in Verbose Mode (-v)

-v
Run the main drive program in verbose mode, printing a message as each subprogram is started.

5.3.25 Pass Arguments to the Assembler (-W a,arguments, -W :as:,arguments)

-W a,*arg1*[,*arg2*...]
-W :as:,*arg1*[,arg2...]
Pass the arguments to the assembler. Example:

```
-Wa,-l  or  -W:as:,-l
```

Pass the option "**-l**" (lower case letter L) to the assembler to get an assembler listing file.

5.3.26 Define Configuration Variable (-W Dname=value)

-W D*name=value*
Set a configuration variable equal to a value for use during configuration file processing.

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.

5.3.27 Pass Arguments to Linker (-W l,arguments, -W :ld:,arguments)

-W l,*arg1*[,*arg2*...]
-W :ld:,arg1[,*arg2*...]
Pass the arguments to the linker.

Any option which is not recognized by the driver or compiler is automatically passed to the linker. **-Wl** may be used to pass options to third-party linkers in cases where such an option resembles a driver or compiler option. See *5.4.56 Suppress Assembler and Linker Parameters (-Xforeign-as-ld)*, p.78. Example:

```
-Wl,-m  or  -W:ld:,-m
```

Pass the option **-m** to the linker to get a link map.

5.3.28 Specify Linker Command File (-W mfile)

-W m*file*
Use the given linker command file instead of the default *version_path***/conf/default.dld**.

NOTE: To suppress use of the **default.lnk** file, specify just **-Wm** with no *file* on the command line.

5.3.29 Specify Startup Module (-W sfile)

-W s*file*
Use the given object file instead of the default startup file (**crt0.o**). Additional object files to be loaded along with the startup file and before any other files can be given separated by commas.

NOTE: To provide a **crt0.s** file or substitute to be assembled on the command line, or to use an existing non-default **crt0.o** file or substitute, specify just **-Ws** with no name to suppress use of the default.

5.3.30 Substitute Program or File for Default (-W xfile)

NOTE: Except for the common cases **-W m** and **-W s** documented above, this option is primarily for use by Wind River.

-W x*file*

Use the given program or file instead of the default program or file for the case indicated by *x*. Some cases take the form **-W** *xname*=*value*. *x* is one of the following:

:as:, a
The assembler.

C
The configuration file to be used. The default is **dtools.conf** (**DTOOLS.CON** for Windows) in the *version_path*/**conf** subdirectory.

:cpp:, p
The C preprocessor. The preprocessor is incorporated in the compiler, so this becomes a synonym for 0.

:c:
The C compiler.

:c++:
The C++ compiler.

c
Pass the string following the **-Wc** exactly as is as an option to the linker. More than one option can be given following **-Wc**, separated by commas. For example, **-Wc-lc,-lproj** would cause the linker to search for missing symbols in libraries **libc.a** and **libproj.a**.

The linker **-l** option is the more usual way to specify libraries.

D
See *5.3.26 Define Configuration Variable (-W Dname=value)*, p.42.

d
The C++ library. The default is **-ld**. See "**c**" for the meaning of **-ld** and additional rules.

:ld:, l
The linker.

L
The object converter; will execute after the linker.

m
See *5.3.28 Specify Linker Command File (-W mfile)*, p.43.

s
See *5.3.29 Specify Startup Module (-W sfile)*, p.43.
The compiler implied by the extension of the source file.

1
The **reorder** program. Specifying **-W1** with no substitute program name will disable the **reorder** program.

2 - 6
Other filter programs. **-W1** and **-W2** execute 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.

5.3.31 Pass Arguments to Subprogram (-W x,arguments)

-W x,*arg1*[,*arg2*...]
Pass the arguments to the subprogram designated by *x*. *x* is one of the following:

:cpp:, p
The preprocessor. The preprocessor is incorporated in the compiler, so this becomes a synonym for 0.

0
The compiler implied by the extension of the source file.

:c:
The C compiler.

:c++:
The C++ compiler.

a, :as:
The assembler. See *5.3.25 Pass Arguments to the Assembler (-W a,arguments, -W :as:,arguments)*, p.42.

l, :ld:
The linker. See *5.3.27 Pass Arguments to Linker (-W l,arguments, -W :ld:,arguments)*, p.43.

L
The object converter. Usually not implemented. If given, it will execute after the linker.

l

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.

5.3.32 Associate Source File Extension (-W x.ext)

-W *x.ext*
Associate a source file extension with a tool; that is, indicate to the main driver program **dcc** or **dplus** which tool should be invoked for an input file with a particular extension. *ext* specifies the extension and *x* specifies a tool, as follows:

0
The compiler implied by the extension of the source file.

:c:
The C compiler.

:c++:
The C++ compiler.

:as: a
The assembler.

:pas:, A
Preprocessor and assembler: both the preprocessor and assembler will be applied to the source. Allows use of preprocessor directives with assembly language.

Example:

```
-W:as:.asm
```

Specify that **file.asm** is an assembly source file.

5.3.33 Suppress All Compiler Warnings (-w)

-w
Suppress all compiler warnings. (Does not apply to assembler or linker.)

5.3.34 Set Detailed Compiler Control Options (-X option)

See *5.4 Compiler -X Options*, p.48.

5.3.35 Specify Default Header File Search Path (-Y I,dir)

-Y I,*dir*
Use *dir* as the default directory to search for header files specified with the **-I** option. A full pathname is allowed. Must occur prior to a **-I** option to be effective for that option.

5.3.36 Specify Search Directories for -l (-Y L, -Y P, -Y U)

These are linker options. See *Specify Search Directories for -l (-Y L, -Y P, -Y U)*, p.368.

5.3.37 Specify Search Directory for crt0.o (-Y S,dir)

Use *dir* as the default directory to search for **crt0.o**. This option is provided as a convenience for older makefiles; users should use the **-W s***file* option instead, as it enables you to specify both the search directory *and* the name of the startup file. See *5.3.29 Specify Startup Module (-W sfile)*, p.43.

5.3.38 Print Subprograms With Arguments (-#, -##, -###)

-#
Print subprogram command lines with arguments as executed.

-##
Print subprogram command line with arguments without actually executing them.

-###
Print subprogram command lines with arguments inside quotes without executing them.

5.3.39 Read Command-Line Options from File or Variable (-@name, -@@name)

-@*name*
Read command-line options from either a file or an environment variable. When **-@***name* is encountered on the command line, the driver first looks for an environment variable with the given *name* and substitutes its value. If an environment variable is not found then the driver 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 the driver terminates.

-@@*name*
Same as **-@***name*; also prints all command-line options on standard output.

5.3.40 Redirect Output (-@E=file, -@E+file, -@O=file, -@O+file)

-@E=*file*
Redirect any output to standard error to the given file.

-@O=*file*
Redirect any output to standard output to the given file.
Use of "**+**" instead of "**=**" will append the output to the file.

5.4 Compiler -X Options

Compiler command-line **-X** options provide fine control over many aspects of the compilation process when behavior other than the default is needed.

Most **-X** options can be set 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 (leading 0x), by using an equal sign: **-X***name***=***m* or **-X***n***=***m*. Some options can be set to an unquoted string, e.g. **-Xfeedback**=*file*.

Many options have multiple names corresponding to different values. For example, **-Xchar-signed** is equivalent to **-X23=0**, and **-Xchar-unsigned** is equivalent to **-X23=1**. Please note that if a value is provided, it is always dominant, regardless of which name is used. Thus, **-Xchar-signed=1** is equivalent **-X23=1**, which is equivalent to **-Xchar-unsigned**. Internally, the name is translated to its number (23 in this case), and then the value is assigned regardless of which name was used.

5.4.1 Option Defaults

If an option is not provided, it defaults to a value of 0 unless otherwise stated. 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**.

To turn off an option which is on by default, or which was set using an environment variable or **-@** option, and for which there is no name for the "=0" case, set it to zero: **-X***name***=0**.

To determine the default for an option, compile a test module without the option using the **-S and -Xshow-configuration=1** options and examine the resulting **.s** assembly language file. All **-X** options used are given in numeric form near the beginning of the file. An option not present defaults to 0.

G. Compiler -X Options Numeric List lists all options having numeric equivalents in numeric order.

-X options can also be specified at the beginning of a source file using:

```
#pragma option -X...
```

The remainder of this section shows all general **-X** options in both forms (name and number).

As noted above, the **-X** options used for a compilation are given as comments in the assembly listing in numeric form. These include both options specified by the user and also some options generated by the compiler. Some of the latter may be undocumented and are present for use by Customer Support.

5.4.2 Compiler -X Options by Function

Below is a list of functional groups of **-X** options. This is followed by the **-X** options in each functional group.

Checking and Profiling

Debugging

- *5.4.71 Define Initial Value for -Xinit-locals (-Xinit-value=n)*, p.84
- *5.4.98 Disable Most Optimizations With -g (-Xoptimized-debug-...)*, p.97
- *5.4.123 Enable Stack Checking (-Xstack-probe), p.106*

Diagnostic and Lint

- *5.4.44 Control Use of Type "double" (-Xdouble...)*, p.72
- *5.4.55 Generate Warnings on Undeclared Functions (-Xforce-declarations, -Xforce-prototypes), p.78*
- *5.4.81 Generate Warnings On Suspicious/Non-portable Code (-Xlint=mask), p.88*
- *5.4.86 Warn On Undefined Macro In #if Statement (-Xmacro-undefined-warn), p.90*
- *5.4.92 Warn On Type and Argument Mismatch (-Xmismatch-warning)*, p.94
- *5.4.124 Diagnose Static Initialization Using Address (-Xstatic-addr-...)*, p.107
- *5.4.126 Buffer stderr (-Xstderr-fully-buffered)*, p.107
- *5.4.127 Terminate Compilation on Warning (-Xstop-on-warning)*, p.107
- *5.4.131 Warn on Large Structure (-Xstruct-arg-warning=n)*, p.108
- *5.4.134 Suppress Warnings (-Xsuppress-warnings), p.109*

Driver

- *5.4.17 Use Old C++ Compiler (-Xc++-old)*, p.62
- *5.4.56 Suppress Assembler and Linker Parameters (-Xforeign-as-ld)*, p.78
- *5.4.61 Control GNU Option Translator (-Xgcc-options-...)*, p.80
- *5.4.67 Ignore Missing Include Files (-Xincfile-missing-ignore), p.82*
- *5.4.77 Create and Keep Assembly or Object File (-Xkeep-assembly-file, -Xkeep-object-file)*, p.86
- *5.4.87 Show Make Rules (-Xmake-dependency)*, p.91
- *5.4.88 Specify Dependency Name or Output File (-Xmake-dependency-...)*, p.92
- *5.4.100 Output Source as Comments (-Xpass-source)*, p.98
- *5.4.105 Preprocess Assembly Files (-Xpreprocess-assembly)*, p.99
- *5.4.107 Use Old Preprocessor (-Xpreprocessor-old)*, p.100
- *5.4.120 Show Target (-Xshow-target)*, p.106

Instruction

- *5.4.3 Prefix Function Identifiers With Underscore (-Xadd-underscore)*, p.57
- *5.4.74 Enable Interworking (-Xinterwork)*, p.85
- *5.4.75 Enable Intrinsic Functions (-Xintrinsic-mask)*, p.86
- *5.4.122 Select Software Floating Point Emulation (-Xsoft-float)*, p.106

Memory

- *5.4.5 Align Functions On n-byte Boundaries (-Xalign-functions=n)*, p.57
- *5.4.6 Specify Minimum Alignment for Single Memory Access to Multi-byte Values (-Xalign-min=n)*, p.58
- *5.4.8 Specify Minimum Array Alignment (-Xarray-align-min)*, p.59
- *5.4.34 Align .debug Sections (-Xdebug-align=n)*, p.69
- *5.4.45 Generate Initializers for Static Variables (-Xdynamic-init)*, p.73
- *5.4.62 Treat All Global Variables as Volatile (-Xglobals-volatile)*, p.81
- *5.4.69 Control Generation of Initialization and Finalization Sections (-Xinit-section)*, p.83
- *5.4.70 Control Default Priority for Initialization and Finalization Sections (-Xinit-section-default-pri)*, p.84
- *5.4.90 Set Maximum Structure Member Alignment (-Xmember-max-align=n)*, p.93
- *5.4.91 Treat All Variables As Volatile (-Xmemory-is-volatile, -X...-volatile)*, p.93
- *5.4.125 Treat All Static Variables as Volatile (-Xstatics-volatile)*, p.107
- *5.4.103 Treat All Pointer Accesses As Volatile (-Xpointers-volatile)*, p.99
- *5.4.130 Align Strings on n-byte Boundaries (-Xstring-align=n)*, p.108
- *5.4.133 Set Minimum Structure Member Alignment (-Xstruct-min-align=n)*, p.109

Optimization

- *5.4.7 Assume No Aliasing of Pointer Arguments (-Xargs-not-aliased)*, p.59
- *5.4.18 Optimize Global Assignments in Conditionals (-Xcga-min-use)*, p.62
- *5.4.22 Disregard ANSI C Library Functions (-Xclib-optim-off)*, p.64
- *5.4.23 Enable Cross-module Optimization (-Xcmo-...)*, p.64

- 5.4.49 Control Inlining Expansion (-Xexplicit-inline-factor), p.75
- 5.4.72 Inline Functions with Fewer Than n Nodes (-Xinline=n), p.84
- 5.4.73 Allow Inlining of Recursive Function Calls (-Xinline-explicit-force), p.85
- 5.4.79 Disable Individual Optimizations (-Xkill-opt=mask, -Xkill-reorder=mask), p.87
- 5.4.84 Do Not Assign Locals to Registers (-Xlocals-on-stack), p.90
- 5.4.95 Enable Extra Optimizations (-XO), p.96
- 5.4.97 Execute the Compiler's Optimizing Stage n Times (-Xopt-count=n), p.97
- 5.4.99 Specify Optimization Buffer Size (-Xparse-size), p.97
- 5.4.112 Restart Optimization From Scratch (-Xrestart), p.103
- 5.4.121 Optimize for Size Rather Than Speed (-Xsize-opt), p.106
- 5.4.132 Control Optimization of Structure Member Assignments (-Xstruct-assign-split-...), p.108
- 5.4.139 Specify Loop Test Location (-Xtest-at-...), p.111
- 5.4.142 Control Loop Unrolling (-Xunroll=n, -Xunroll-size=n), p.112

Output

- 5.4.15 Control Allocation of Uninitialized Variables in "COMMON" and bss Sections (-Xbss-off, -Xbss-common-off), p.62
- 5.4.30 Dump Symbol Information for Macros or Assertions (-Xcpp-dump-symbols), p.67
- 5.4.59 Generate .frame_info for C functions (-Xframe-info), p.80
- 5.4.63 Do Not Pass #ident Strings (-Xident-off), p.81
- 5.4.67 Ignore Missing Include Files (-Xincfile-missing-ignore), p.82
- 5.4.87 Show Make Rules (-Xmake-dependency), p.91
- 5.4.88 Specify Dependency Name or Output File (-Xmake-dependency-...), p.92
- 5.4.100 Output Source as Comments (-Xpass-source), p.98
- 5.4.106 Suppress Line Numbers in Preprocessor Output (-Xpreprocessor-lineno-off), p.100
- 5.4.117 Disable Generation of Priority Section Names (-Xsect-pri-...), p.105

- *5.4.116 Generate Each Function in a Separate CODE Section Class (-Xsection-split)*, p.104
- *5.4.118 Control Listing of -X Options in Assembly Output (-Xshow-configuration=n)*, p.105
- *5.4.141 Append Underscore to Identifier (-Xunderscore-...)*, p.111

Position-independent data

- *5.4.33 Generate Position-independent Data (PID) (-Xdata-relative...)*, p.68
- *5.4.102 Generate Position-Independent Code for Shared Libraries (-Xpic)*, p.99

Precompiled Headers

- *5.4.101 Use Precompiled Headers (-Xpch-...)*, p.98

Sections

- *5.4.4 Set Addressing Mode for Sections (-Xaddr-...)*, p.57
- *5.4.15 Control Allocation of Uninitialized Variables in "COMMON" and bss Sections (-Xbss-off, -Xbss-common-off)*, p.62
- *5.4.25 Use Absolute Addressing for Code (-Xcode-absolute...)*, p.65
- *5.4.29 Locate Constants With "text" or "data" (-Xconst-in-text, -Xconst-in-data)*, p.67
- *5.4.32 Use Absolute Addressing for Code (-Xdata-absolute...)*, p.68
- *5.4.34 Align .debug Sections (-Xdebug-align=n)*, p.69
- *5.4.82 Allocate Static and Global Variables to Local Data Area (-Xlocal-data-area=n)*, p.89
- *5.4.83 Restrict Local Data Area Optimization to Static Variables (-Xlocal-data-area-static-only)*, p.90
- *5.4.93 Specify Section Name (-Xname-...)*, p.94
- *5.4.104 Control Interpretation of Multiple Section Pragmas (-Xpragma-section-...)*, p.99
- *5.4.115 Pad Sections for Optimized Loading (-Xsection-pad)*, p.104

Syntax

- *5.4.14 Parse Initial Values Bottom-up (-Xbottom-up-init)*, p.61

- *5.4.31 Suppress Preprocessor Spacing (-Xcpp-no-space)*, p.68
- *5.4.24 Use the 'new' Compiler Frontend (-Xcnew)*, p.65
- *5.4.41 Specify C Dialect (-Xdialect-...)*, p.71
- *5.4.42 Disable Digraphs (-Xdigraphs-...)*, p.72
- *5.4.43 Allow Dollar Signs in Identifiers (-Xdollar-in-ident)*, p.72
- *5.4.66 Treat #include As #import (-Ximport)*, p.82
- *5.4.75 Enable Intrinsic Functions (-Xintrinsic-mask)*, p.86
- *5.4.78 Enable Extended Keywords (-Xkeywords=mask)*, p.86
- *5.4.85 Expand Macros in Pragmas (-Xmacro-in-pragma)*, p.90
- *5.4.107 Use Old Preprocessor (-Xpreprocessor-old)*, p.100
- *5.4.128 Compile C/C++ in Pedantic Mode (-Xstrict-ansi)*, p.107
- *5.4.135 Swap '\n' and '\r' in Constants (-Xswap-cr-nl)*, p.110
- *5.4.140 Truncate All Identifiers After m Characters (-Xtruncate)*, p.111
- *5.4.144 Void Pointer Arithmetic (-Xvoid-ptr-arith-ok)*, p.113

Type

- *5.4.9 Change bit-field type to reduce structure size (-Xbit-fields-compress-...), p.59*
- *5.4.10 Specify Sign of Plain Bit-field (-Xbit-fields-signed, -Xbit-fields-unsigned)*, p.60
- *5.4.20 Specify Sign of Plain Char (-Xchar-signed, -Xchar-unsigned)*, p.63
- *5.4.19 Generate Code Using ASCII Character Set (-Xcharset-ascii)*, p.63
- *5.4.44 Control Use of Type "double" (-Xdouble...)*, p.72
- *5.4.47 Specify enum Type (-Xenum-is-...)*, p.73
- *5.4.50 Force Precision of Real Arguments (-Xextend-args)*, p.75
- *5.4.51 Specify Degree of Conformance to the IEEE754 Standard (-Xfp-fast, -Xfp-normal, -Xfp-pedantic)*, p.76
- *5.4.96 Use Old Inline Assembly Casting(-Xold-inline-asm-casting)*, p.96
- *5.4.57 Convert Double and Long Double (-Xfp-long-double-off, -Xfp-float-only)*, p.79
- *5.4.58 Specify Minimum Floating Point Precision (-Xfp-min-prec...), p.79*

- *5.4.64 Enable Strict implementation of IEEE754 Floating Point Standard (-Xieee754-pedantic)*, p.81
- *5.4.129 Ignore Sign When Promoting Bit-fields (-Xstrict-bitfield-promotions)*, p.108
- *5.4.145 Define Type for wchar (-Xwchar=n)*, p.113
- *5.4.146 Control Use of wchar_t Keyword (-Xwchar_t-...)*, p.113

C++

- *5.4.12 Set Type for Bool (-Xbool-is-...)*, p.61
- *5.4.13 Control Use of Bool, True, and False Keywords (-Xbool-...)*, p.61
- *5.4.16 Use Abridged C++ Libraries (-Xc++-abr)*, p.62
- *5.4.17 Use Old C++ Compiler (-Xc++-old)*, p.62
- *5.4.21 Use Old for Scope Rules (-Xclass-type-name-visible), p.64*
- *5.4.26 Mark Sections as COMDAT for Linker Collapse (-Xcomdat)*, p.66
- *5.4.27 Maintain Project-wide COMDAT List (-Xcomdat-info-file), p.66*
- *5.4.42 Disable Digraphs (-Xdigraphs-...)*, p.72
- *5.4.48 Enable Exceptions (-Xexceptions-...), p.74*
- *5.4.54 Use Old for Scope Rules (-Xfor-init-scope-...), p.78*
- *5.4.59 Generate .frame_info for C functions (-Xframe-info)*, p.80
- *5.4.65 Control Template Instantiation (-Ximplicit-templates...)*, p.82
- *5.4.76 Set longjmp Buffer Size (-Xjmpbuf-size=n)*, p.86
- *5.4.89 Set Template Instantiation Recursion Limit (-Xmax-inst-level=n)*, p.93
- *5.4.94 Disable C++ Keywords namespace and Using (-Xnamespace-...)*, p.96
- *5.4.101 Use Precompiled Headers (-Xpch-...)*, p.98
- *5.4.114 Enable Run-time Type Information (-Xrtti, -Xrtti-off)*, p.103
- *5.4.119 Print Instantiations (-Xshow-inst)*, p.105
- *5.4.128 Compile C/C++ in Pedantic Mode (-Xstrict-ansi)*, p.107
- *5.4.137 Disable Certain Syntax Warnings (-Xsyntax-warning-...)*, p.110
- *5.4.143 Runtime Declarations in Standard Namespace (-Xusing-std-...)*, p.112
- *5.4.145 Define Type for wchar (-Xwchar=n)*, p.113

- *5.4.146 Control Use of wchar_t Keyword (-Xwchar_t-...)*, p.113

The sections that follow present **-X** options in alphabetic order.

5.4.3 Prefix Function Identifiers With Underscore (-Xadd-underscore)

-Xadd-underscore
-X34
Prefix an underscore to function names only. Concatenation of underscore is useful when compiling libraries, to avoid using the same namespace as user programs.

5.4.4 Set Addressing Mode for Sections (-Xaddr-...)

-Xaddr-code=n
-X105=n
Specify addressing for code.

-Xaddr-const=*n*
-X102=*n*
Specify addressing for constant static and global variables.

-Xaddr-data=n
-X100=n
Specify addressing for non-constant static and global variables.

-Xaddr-string=*n*
-X104=*n*
Specify addressing for strings.

-Xaddr-user=*n*
-X106=*n*
Specify addressing for user-defined sections.

See the discussion of *addr-mode* in *14.2 Addressing Mode — Functions, Variables, Strings*, p.238 for more information.

5.4.5 Align Functions On n-byte Boundaries (-Xalign-functions=n)

-Xalign-functions=*n*
-X54=*n*
Align each function on an address boundary divisible by *n* (which must be greater than or equal to the default alignment for the target microprocessor). If *n* is absent, the option has no effect. This option is designed for targets having

some type of burst-mode memory access, for example a target that can fetch multiple instructions if aligned on a 32-byte boundary.

5.4.6 Specify Minimum Alignment for Single Memory Access to Multi-byte Values (-Xalign-min=n)

-Xalign-min=*n*
-X93=*n*

Set the minimum alignment required by the target processor to access a multi-byte value (e.g., **short**, **long**) in memory as an atomic unit, that is, in a single memory access. This option is set automatically by the compiler based on the target processor and should seldom be set by the user.

NOTE: This option does not change how data is aligned; it changes the instructions which the compiler generates to access multi-byte unaligned objects.

Technical details: if the target processor can access objects at any alignment with a single instruction, *n* is set to 1. For a processor which requires that multi-byte objects be aligned on even-byte boundaries for direct access, *n* is set to 2. Unaligned objects on such a processor must be accessed byte-by-byte. For a processor that requires 4-byte objects be on a 4-byte boundary, *n* is set to 4 (2-byte objects aligned on 2-byte boundaries can still be accessed with a single instruction).

The default value of *n* equals the maximum alignment restriction as given in the manufacturer's documentation for the processor. Note that it may differ among processors in a family. As of this writing, the default is 4.

NOTE: Because **-Xalign-min is > 1**, in a packed structure (a) bit-fields members are not allowed, (b) **volatile** members will not be accessed atomically, and (c) compound operators (for example, "**+=**") cannot be used with **volatile** members. See *Restrictions and Additional Information*, **p.130** for details.

Synonym: **-Xmin-align=***n*.

5.4.7 Assume No Aliasing of Pointer Arguments (-Xargs-not-aliased)

-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. Example:

```
int g;

func(int* a1, int* a2);

void main () {
    int i = 1;
    int j = 2;

    func(&i,&j);    /* OK     */
    func(&i,&i);    /* not OK */
    func(&i,&g);    /* not OK */
}
```

See also *no_alias Pragma*, p.126.

5.4.8 Specify Minimum Array Alignment (-Xarray-align-min)

-Xarray-align-min=*n*
-X161=*n*

Align arrays on the larger of *n* or the default alignment for the type of the array elements. *n* should be a power of 2. When this option is used, values given for **-Xstring-align** are ignored.

5.4.9 Change bit-field type to reduce structure size (-Xbit-fields-compress-...)

-Xbit-fields-compress
-X135=1
-Xbit-fields-compress-off
-X135=0

C only. Change the type of a bit-field if possible to generate more compact storage. The default is off.

The algorithm is as follows:

Examine all structure members before assigning offsets. Record:

BitFieldMaxAlign = maximum alignment of any bit-field.

NonBitFieldMaxAlign = maximum alignment of any non bit-field.

WidthMaxBitField = number bits in largest bit-field.

IF **BitFieldMaxAlign > NonBitFieldMaxAlign** THEN

NewType = **unsigned** integer type having the same alignment as that of the **NonBitFieldMaxAlign**.

IF **WidthMaxBitField <=** bits in **NewType** THEN

Change the type of each **unsigned** bit-field larger than **NewType** to **NewType** and each **signed** bit-field larger than **NewType** to **signed NewType**.

This option is intended for legacy code. The same effect may be achieved in new code by using the smallest types having the required alignments.

Synonym: **-Xbitfield-compress.**

5.4.10 Specify Sign of Plain Bit-field (-Xbit-fields-signed, -Xbit-fields-unsigned)

-Xbit-fields-signed
-X12=0

C only. Handle bit-fields without the **signed** or **unsigned** keyword as signed integers.

Synonym: **-Xsigned-bitfields**.

-Xbit-fields-unsigned
-X12

C only. Treat bit-fields without the **signed** or **unsigned** keyword as unsigned integers. This is the default setting.

Synonym: **-Xunsigned-bitfields**.

See also *5.4.129 Ignore Sign When Promoting Bit-fields (-Xstrict-bitfield-promotions)*, p.108.

5.4.11 Insert Profiling Code (-Xblock-count)

-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 *28. D-BCNT Profiling Basic Block Counter* for details, and also *5.4.52 Optimize Using Profile Data (-Xfeedback=file)*, p.76.

5.4.12 Set Type for Bool (-Xbool-is-...)

-Xbool-is-char
-X119=44
Implement type **bool** as a plain **char**. This is the default.

-Xbool-is-int
-X119=4
C++ only. Implement type **bool** as a **signed int**. This may produce less code on some architectures but will require more data space.

5.4.13 Control Use of Bool, True, and False Keywords (-Xbool-...)

-Xbool-on
-X213=0
Enable the **bool**, **true**, and **false** keywords. This is the default.

-Xbool-off
-X213
C++ only. Disable the **bool**, **true**, and **false** keywords.

Synonym: **-Xno-bool**.

5.4.14 Parse Initial Values Bottom-up (-Xbottom-up-init)

-Xbottom-up-init
-X21
C only. 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 **-Xdialect-pcc** is set.

5.4.15 Control Allocation of Uninitialized Variables in "COMMON" and bss Sections (-Xbss-off, -Xbss-common-off)

-Xbss-common-off
-X83=3
Disable use of the "**COMMON**" feature so that the compiler or assembler will allocate each uninitialized public variable in the **.bss** section for the module defining it, and the linker will require exactly one definition of each public variable. See *23.4 COMMON Sections*, p.352**.**

Synonym: **-Xno-common**.

-Xbss-off
-X83=1
Put all variables in the **.data** section instead of allocating uninitialized variables in the **.bss** section.

Synonym: **-Xno-bss.**

5.4.16 Use Abridged C++ Libraries (-Xc++-abr)

-Xc++-abr
Link to the abridged C++ libraries. Automatically disables exception-handling (**-Xexceptions=off**). See *13.2 C++ Standard Libraries*, p.220.

5.4.17 Use Old C++ Compiler (-Xc++-old)

-Xc++-old
Invoke the older C++ compiler that preceded version 5.0. Useful for compiling legacy code that is not ANSI-compliant. See *Older Versions of the Compiler*, p.214.

5.4.18 Optimize Global Assignments in Conditionals (-Xcga-min-use)

-Xcga-min-use=*n*
When a global variable is accessed repeatedly within a conditional statement, the compiler can replace the global variable with a temporary local copy (which can be stored in a register), then reassign the local variable to the global variable when the conditional finishes execution.

If conditional global assignment is enabled, the compiler determines whether to copy a global variable by estimating the number of times the global variable is accessed within the conditional block at runtime. (The exact number of accesses may depend on factors, such as the value of a loop counter, that cannot be known at compile time.) If the global variable is accessed *n* or more times, the compiler performs the optimization. The default value of *n* is 20.

Conditional global assignment is enabled by default (**-Xcga-min-use=20**) whenever optimizations are enabled (**-O** or **-XO**). To disable conditional global assignment, set *n* to 0 (**-Xcga-min-use=0**). Conditional global assignment is never performed on variables declared or treated as volatile (see *5.4.91 Treat All Variables As Volatile (-Xmemory-is-volatile, -X...-volatile)*, p.93) and should be used with caution in multi threaded environments.

5.4.19 Generate Code Using ASCII Character Set (-Xcharset-ascii)

-Xcharset-ascii
-X60=1

Generate code 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.

Synonym: **-Xascii-charset**.

5.4.20 Specify Sign of Plain Char (-Xchar-signed, -Xchar-unsigned)

-Xchar-signed
-X23=0

Treat variables declared **char** without either of the keywords **signed** or **unsigned** as signed characters.

Synonym: **-Xsigned-char**.

-Xchar-unsigned
-X23

Treat variables declared **char** without either of the keywords **signed** or **unsigned** as unsigned characters.

Synonym: **-Xunsigned-char**.

The default setting is **unsigned**. See also Table 8-1 and **__SIGNED_CHARS__** in *6.1 Preprocessor Predefined Macros*, p.117.

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.

5.4.21 Use Old for Scope Rules (-Xclass-type-name-visible)

-Xclass-type-name-visible
-X218=1

C only. Direct the compiler not to hide **struct** or **union** names when other identifiers with the same names are declared in the same scope. For example, consider the following statement:

```
struct S {...} S[10];
```

With or without this option, the form **struct S** may always be used later to declare additional variables of type **struct S**. However, without the option, **sizeof(S)** will refer to the size of the array, while with this option, **sizeof(S)** will refer to the size of the structure.

5.4.22 Disregard ANSI C Library Functions (-Xclib-optim-off)

-Xclib-optim-off
-X66

Direct the compiler to disregard all knowledge of ANSI C library functions.

By default, the compiler automatically includes, before all other header files, the file **lpragma.h**, which contains **pure_function**, **no_return**, and **no_side_effects** pragmas and other statements that allow optimization of calls to C library functions. (If the default include directory *version_path***/include** exists, the compiler looks for **lpragma.h** only in this directory. If *version_path***/include** does not exist, the compiler searches for **lpragma.h** in other user-specified directories.)

The option disables use of **lpragma.h**.

Synonym: **-Xno-recognize-lib**.

5.4.23 Enable Cross-module Optimization (-Xcmo-...)

-Xcmo-gen=name

Generate a database, in file *name*, for cross-module optimization.

-Xcmo-use=name
Compile with cross-module optimization using information in database *name*; update database.

-Xcmo-exclude-inline=list
Combined with **-Xcmo-use**, tells the compiler *not* to inline specified functions. *list* is a comma-delimited list of functions which should not be inlined across modules. For C++, use mangled function names.

-Xcmo-verbose
Combined with **-Xcmo-gen** or **-Xcmo-use**, lists all functions that are inlined across modules. Useful for tracking dependencies.

These options enable cross-module optimization, which allows the compiler to optimize calls between functions in different source files. See *10.2 Cross-Module Optimization*, p.188 for details. Cross-module optimization is disabled by default.

5.4.24 Use the 'new' Compiler Frontend (-Xcnew)

-Xcnew
Compile using a compiler frontend derived from one produced by the Edison Design Groupd. By default, invoking **-Xcnew** also invokes **-Xdialect-c99**. Supported only with the **:rtp** execution environment.

5.4.25 Use Absolute Addressing for Code (-Xcode-absolute...)

-Xcode-absolute-far
-X58=6
Use 32-bit absolute addressing for code.

See *14.2 Addressing Mode — Functions, Variables, Strings*, p.238.

-Xcode-absolute-near
-X58=5
Use 32-bit absolute addressing for code. (Same as **-Xcode-absolute-far**. Maintained for compatibility.)

See *14.2 Addressing Mode — Functions, Variables, Strings*, p.238.

5.4.26 Mark Sections as COMDAT for Linker Collapse (-Xcomdat)

-Xcomdat
-X120

C++ only. Mark all generated sections as COMDAT. The linker automatically collapses identical COMDAT sections to a single section in memory. This is the default.

By default, the compiler automatically generates a section for each instantiation of each member function or static class variable in a template in each module where the member function or variable is used. Given **-Xcomdat**, the compiler marks all implicit template instantiations as COMDAT and the linker collapses identical instances.

-Xcomdat-off

Generate all template instantiations and inline functions required as static entities in the resulting object file. If a template is used in more than one module, **-Xcomdat-off** results in multiple instances of static member function variables or static class variables, instead of a single instance as is likely intended; to avoid this, enable **-Ximplicit-templates-off**.

See *5.4.65 Control Template Instantiation (-Ximplicit-templates...)*, p.82 and *Templates*, p.223 for details.

If a section is present in both COMDAT and non-COMDAT forms, the linker will treat symbols in the COMDAT section as weak. See *weak Pragma*, p.134 for details on weak symbols.

5.4.27 Maintain Project-wide COMDAT List (-Xcomdat-info-file)

-Xcomdat-info-file=*filename*

C++ only. When **-Xcomdat** is enabled, generate and maintain (in *filename*) a list of COMDAT entries across modules. The list is automatically updated and checked for consistency with each build. This option speeds up builds and reduces object-file size in projects that make extensive use of templates. Since COMDAT sections are ultimately collapsed by the linker, this option has no effect on the final executable file.

5.4.28 Optimize Static and Global Variable Access Conservatively (-Xconservative-static-live)

-Xconservative-static-live
-X139
Make optimizations of static and global variable accessing less aggressive; for example, do not delete assignments to such variables in infinite loops from which there is no apparent return.

5.4.29 Locate Constants With "text" or "data" (-Xconst-in-text, -Xconst-in-data)

-Xconst-in-text=*mask*
-X74=*mask*

-Xconst-in-data
-X74=0
Locate data in the **CONST** (mask bit 0x1), and **STRING** (mask bit 0x4) section classes according to the given mask bit: if 1, locate in a "text" section (the default), else if 0, locate in a "data" section.

mask may be given in hex, and mask bits may be OR-ed to select more than one, e.g., **-Xconst-in-text=0x5**. Undefined mask bits are ignored.

The default value of this option is given in *Moving initialized Data From "text" to "data"*, p.245.

-Xconst-in-data and **-Xstrings-in-text** are historical shortcuts for locating all "constants" (**CONST**, and **STRING** classes, not just "const" or string data) in "data" sections (*mask*=0) or "text" sections (*mask*=0xff) respectively.

The exact name of the "text" and "data" sections depends on the target. See the discussion in *14. Locating Code and Data, Addressing, Access* for exact section names and examples, as well as *Moving initialized Data From "text" to "data"*, p.245.

When **STRING** is in a text section, identical string constants will be stored only once. This is the default in version 3.6 and later.

5.4.30 Dump Symbol Information for Macros or Assertions (-Xcpp-dump-symbols)

-Xcpp-dump-symbols=*mask*
-X158=*mask*
Dump symbol information for macros, assertions, or both. To show macros, set bit 0 (the LSB) of *mask* to 1. To show assertions, set bit 1 to 1. To show line

numbers, set bit 2 to 0. The default *mask* is 7 (show macros and assertions, no line numbers).

5.4.31 Suppress Preprocessor Spacing (-Xcpp-no-space)

-Xcpp-no-space
-X117
C only. Do not insert spaces around macro names and arguments during preprocessing.

5.4.32 Use Absolute Addressing for Code (-Xdata-absolute...)

-Xdata-absolute-far
-X59=6
Use 32-bit absolute addressing for data.

See *14.2 Addressing Mode — Functions, Variables, Strings*, p.238.

-Xdata-absolute-near
-X59=5
Use 16-bit absolute addressing for data.

See *14.2 Addressing Mode — Functions, Variables, Strings*, p.238.

5.4.33 Generate Position-independent Data (PID) (-Xdata-relative...)

-Xdata-relative-far
-X59=2
Generate position-independent data (PID) references to all global or static variables (except strings and **const** variables if the **-Xconst-in-text=0** option is used). Use 32-bit offsets from register **r9**.

Synonym: **-Xfar-data-relative.**

.

-Xdata-relative-near
-X59=1
"Near" addressing is not supported; for convenience, same as **-Xdata-relative-far.**

NOTE: If option **-Xconst-in-text=0xf (equivalent to older option -Xstrings-in-text)**, strings and **const** variables will be placed in "text" sections and addressed as code rather than as position-independent data. See *Moving initialized Data From "text" to "data"*, p.245 for details.

5.4.34 Align .debug Sections (-Xdebug-align=n)

-Xdebug-align[=*n*]
Align **.debug** sections on specified boundaries. *n* is a power of 2; e.g., **-Xdebug-align=3** aligns **.debug** sections on 8-byte boundaries. If *n* is omitted, alignment defaults to 4-byte boundaries.

Without this option, **.debug** sections are aligned on byte boundaries.

5.4.35 Select DWARF Format (-Xdebug-dwarf...)

-Xdebug-dwarf1
-X153=1
Generate DWARF 1.1 debug information.

-Xdebug-dwarf2
-X153=2
Generate DWARF 2 debug information. This is the default.

-Xdebug-dwarf3
-X153=3
Generate DWARF 3 debug information.

-Xdebug-dwarf2-extensions-off
Suppress vendor-specific extensions in DWARF 2 and DWARF 3 debug information.

5.4.36 Generate Debug Information for Inlined Functions (-Xdebug-inline-on)

-Xdebug-inline-on
Generate debugging information for all inlined functions. Works with DWARF 2 and DWARF 3 only. Can result in very large executables. This option is disabled by default.

5.4.37 Emit Debug Information for Unused Local Variables (-Xdebug-local-all)

-Xdebug-local-all
Emit debugging information for all local variables, even variables that are never used. This option is disabled by default.

5.4.38 Generate Local CIE for Each Unit (-Xdebug-local-cie)

-Xdebug-local-cie
Generate a local Common Information Entry (CIE) for each unit. This option, which eliminates the dependency on the debug library **libg.a**, is applicable only with DWARF 2 or DWARF 3 debug information.

5.4.39 Disable debugging information Extensions (-Xdebug-mode=mask)

-Xdebug-mode=*mask*
-X99=*mask*
Disable extensions to debugging information per bits in *mask*. May be necessary for other vendors' assemblers or for debuggers which cannot process the extensions.

mask may be given in hex, and mask bits may be OR-ed to select more than one, e.g., **-Xdebug-mode=0x6**. Undefined mask bits are ignored.

`0x2`
Information regarding executable code in a header file (DWARF1, ELF).

`0x4`
Use of **.d1line** assembler directive (DWARF1, ELF).

`0x10`
Line number information for **asm** statements (DWARF1, DWARF2, DWARF3).

`0x40`
Use of **.d1_line_start** and **.d1_line_end** assembler directives (DWARF1).

`0x100`
Column information (DWARF 2 and DWARF 3, C++).

5.4.40 Disable Debug Information Optimization (-Xdebug-struct-...)

-Xdebug-struct-all
-X116=1
Force generation of type information for **typedef**, **struct**, and **union**, and **class** types, even when such types are not referenced in a file.

-Xdebug-struct-compact
-X116=0
Do not output types which are not used in debug information. This is the default, and it generates more compact but still complete version of debug information.

5.4.41 Specify C Dialect (-Xdialect-...)

-Xdialect-c89
-X230=0
Follow the C89 standard for C. See Table B-1 for details.

-Xdialect-c99
-X230=1
Follow the C99 standard for C. See Table B-1 for details.

Only a subset of the C99 standard is supported.

-Xdialect-k-and-r
-X7=0
Follow the "C standard" as defined by the original K&R C reference manual, but with all the new ANSI C features added. Where K&R and ANSI differ, **-Xdialect-k-and-r** follows K&R. See Table B-2 for details.

Synonyms: **-Xk-and-r**, **-Xt**.

-Xdialect-ansi
-X7=1
Follow the ANSI C standard with some additions. See Table B-2 for details. This is the default.

Synonyms: **-Xansi**, **-Xa**.

-Xdialect-strict-ansi
-X7=2
Strictly follow the ANSI C *and* C++ standards. See Table B-2 for details. For C++, see *5.4.128 Compile C/C++ in Pedantic Mode (-Xstrict-ansi)*, p.107.

Synonym: **-Xstrict-ansi**, **-Xc**.

-Xdialect-pcc
-X7=3
Follow the C standard as defined by the UNIX System V.3 C compiler. See Table B-1 for details.

Synonym: **-Xpcc**.

5.4.42 Disable Digraphs (-Xdigraphs-...)

-Xdigraphs-on
-X202=0
C++ only. Enable 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**. This is the default.

-Xdigraphs-off
-X202
Disable digraphs.

Synonym: **-Xno-digraphs**.

5.4.43 Allow Dollar Signs in Identifiers (-Xdollar-in-ident)

-Xdollar-in-ident
-X67
Allow dollar sign characters, "**$**", in identifiers.

5.4.44 Control Use of Type "double" (-Xdouble...)

-Xdouble-avoid
-X96=3
C only. Force all double constants to single precision and generation of only single precision instructions.

-Xdouble-error
-X96=1
Generate an error if any double precision operation is used. It will also force all double constants to single precision and generation of only single precision instructions.

-Xdouble-warning
-X96=2

Generate a warning if any double precision operation is used. It will also force all double constants to single precision and generation of only single precision instructions.

5.4.45 Generate Initializers for Static Variables (-Xdynamic-init)

-Xdynamic-init=2
-X121=2
Extends the **-Xdynamic-init=1** option to generate code in the initialization section for all initializers, not just addresses.

5.4.46 Compile in Little-endian Mode (-Xendian-little)

-Xendian-little
-X94
Compile in little-endian mode. This option is generated automatically by the driver when "**L**" is used as part of the **-t** option, e.g., **-tARMLN**. *It should not be given by the user and doing so may lead to undefined behavior.* It is documented for information only.

5.4.47 Specify enum Type (-Xenum-is-...)

-Xenum-is-best
-X8=2
Use the smallest *signed or unsigned* integer type permitted by the range of values for an enumeration, that is, the first of **signed char**, **unsigned char**, **short**, **unsigned short**, **int**, **unsigned int**, **long**, or **unsigned long** sufficient to represent the values of the enumeration constants. (**long long** is not available for enumerated types.) Thus, an enumeration with values from 1 through 128 will have base type **unsigned char** and require one byte. (Using the **packed** keyword on an enumerated type yields the same result as **-Xenum-is-best**.)

-Xenum-is-int
-X8
This is the default. For C modules, the **enum** type is always equivalent to **int**. For C++, each **enum** type is equivalent to **int** if the range will fit, or **unsigned int** if it will not; if the range will not fit into either, a warning is issued and **unsigned int** is used.

-Xenum-is-short
-X8=3
Each enum type is always equivalent to **signed short** if the range will fit, or **unsigned short** if it will not. If the range will not fit into either, a warning is issued and **unsigned short** is used.

-Xenum-is-small
-X8=0
Use the smallest *signed* integer type permitted by the range of values for an enumeration, that is, the first of **signed char**, **short**, **int**, or **long** sufficient to represent the values of the enumeration constants. Thus, an enumeration with values from 1 through 128 will have base type **short** and require two bytes.

-Xenum-is-unsigned
-X8=4
Use the smallest *unsigned* integer type permitted by the range of values for an enumeration, that is, the first of **unsigned char**, **unsigned short**, **unsigned int**, or **unsigned long** sufficient to represent the values of the enumeration constants. Thus, an enumeration with values from 1 through 128 will have base type **unsigned char** and require one byte.

NOTE: If modules compiled with different **-Xenum-is-**... options are mixed in a program, compatibility problems may result.

When an enumerated type occurs within a packed structure, the default behavior is to use the smallest possible integer type for the enumeration constants (**-Xenum-is-best**). To override this behavior, specify **-Xenum-is-short** or **-Xenum-is-unsigned**.

5.4.48 Enable Exceptions (-Xexceptions-...)

-Xexceptions-off
-X200=0
C++ only. Disable exceptions. Compiling a program with any of the keywords **try**, **catch**, or **throw** will cause a compilation error. (But **throw( )** is still allowed in function declarations to indicate that **new** or **delete** will not throw exceptions.) Compiling with this option will reduce stack space and increase execution speed when classes with destructors are used.

Synonym: **-Xno-exception**.

-Xexceptions
-X200
C++ only. Enable exceptions. This is the default.

For mixed C/C++ programs, see also 5.4.59 Generate .frame_info for C functions (-Xframe-info), p.80.

Synonym: **-Xexception**.

5.4.49 Control Inlining Expansion (-Xexplicit-inline-factor)

-Xexplicit-inline-factor
-Xexplicit-inline-factor=*n*
-X136=*n*

Limits the inlining in a function (explicit and implicit) to an expansion of *n* times (measured in nodes where, roughly, each operator or operand counts as one node).

Given a function **f**, the compiler first inlines all functions explicitly declared inline which **f** calls, as well as any other small functions which can be inlined based on the other inlining optimization controls. It then divides the new size of the function (number of nodes) by the size with no inlining. If the result is **<=** *n*, it looks for new inlining opportunities in the resulting code and repeats the cycle. Once an expansion of *n* times is exceeded, inlining stops.

If **-Xexplicit-inline-factor** is specified with no value, *n* defaults to 3. If **-Xexplicit-inline-factor** is not specified, the default value is 0 (which means no limit) for C and 3 for C++.

See also 5.4.73 Allow Inlining of Recursive Function Calls (-Xinline-explicit-force), p.85.

5.4.50 Force Precision of Real Arguments (-Xextend-args)

-Xextend-args
-X77

Make all floating point arguments use the precision given by whichever of **-Xfp-min-prec-double**, **-Xfp-min-prec-long-double**, or **-Xfp-min-prec-float** is in force (all are settings of **-X3**), even if prototypes are used. (If none of the **-X3** options are also given, the default is **-Xfp-min-prec-double** as that is equivalent to **-X3=0**).

NOTE: If this option is used, libraries containing functions with floating point parameters must be recompiled. For safety, recompile all libraries to avoid missing any such functions.

5.4.51 Specify Degree of Conformance to the IEEE754 Standard (-Xfp-fast, -Xfp-normal, -Xfp-pedantic)

-Xfp-fast
-X82=2
Favor floating-point performance over conformance to the IEEE754 floating-point standard.

-Xfp-normal
-X82=0
Use normal (relaxed) conformance to the IEEE754 floating-point standard. This is the default.

-Xfp-pedantic
-X82=1
Use strict conformance to the IEEE754 floating-point standard. This option is equivalent to using **-Xieee754-pendatic**. (See *5.4.64 Enable Strict implementation of IEEE754 Floating Point Standard (-Xieee754-pedantic)*, p.81.)

The **-Xfp-fast** option allows floating-point division by a constant to be optimized into a multiply by the reciprocal of the constant. This optimization is inhibited for **-Xpf-normal** and **-Xfp-pedantic** unless the constant is a power of two.

5.4.52 Optimize Using Profile Data (-Xfeedback=file)

-Xfeedback
-Xfeedback=file
(no numeric equivalent)
Use profiling information generated by the **-Xblock-count** (see *5.4.11 Insert Profiling Code (-Xblock-count)*, p.60) option to optimize for faster code. *file* 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 *15.8.2 File I/O*, p.265 for file I/O in an embedded environment.)
- Recompile, now with the **-XO** and **-Xfeedback** options to produce high-level speed optimized code. Use **-Xfeedback-frequent** and **-Xfeedback-seldom** described below to control how the feedback data affects optimization.

5.4.53 Set Optimization Parameters Used With Profile Data (-Xfeedback-frequent, -Xfeedback-seldom)

-Xfeedback-frequent
-X68=n
-Xfeedback-seldom
-X69=n

Change the parameters used to control optimization of basic blocks when using profile data, for example, the amount of inlining, loop unrolling, and reorganization to reduce branches actually taken, all to increase speed (sometimes at the expense of space).

When using **-Xprof-feedback** (*5.4.110 Optimize Using RTA Profile Data (-Xprof-feedback)*, p.101) and **-Xfeedback (***5.4.52 Optimize Using Profile Data (-Xfeedback=file)*, p.76**)**, the compiler divides the basic blocks into three categories: code executed “frequently”, “sometimes”, and “seldom”. More of the above optimizations are done for “frequent” code, while less or none is done for code executed “seldom”.

The higher the thresholds, the more often code must be executed to get into the “frequent” category.

The defaults are **-Xfeedback-seldom=10** and **-Xfeedback-frequent=50** and are used as follows: each execution of a basic block recorded in the profile counts as one “tick”. The low-mark and high-mark values are normalized on a basis of 1,000 ticks, which means that the options have units of a tenth of a percent. That is, the default values mean that, if exactly 1,000 ticks are recorded, blocks executed fewer than 10 times (up to 1%) are marked “seldom”, those executed from 10 to 50 times (1% to 5%) are marked “sometimes”, and those executed 50 or more times (5% of more) are marked “frequent”. Example:

```
-Xfeedback-frequent=30
```

means that blocks accounting for 3% or more of all ticks will go into the “frequent” category, and the compiler will do more inlining of functions called within these blocks, more loop unrolling, etc., to decrease their execution time.

Synonyms: **-Xhi-mark** for **-Xfeedback-frequent**, **-Xlo-mark** for **-Xfeedback-seldom**.

5.4.54 Use Old for Scope Rules (-Xfor-init-scope-...)

-Xfor-init-scope-for
-X217=0
Use "new" scope rules for variables declared in the initialization part of a **for** statement. With this option, the scope of a variable declared in the initialization part extends to the end of the **for** statement.

-Xfor-init-scope-outer
-X217
C++ only. Use "old" scoping rules for variables declared in the initialization part of a **for** statement. With this option, the scope extends to the end of the scope enclosing the **for** statement.

Synonym: **-Xold-scoping**.

5.4.55 Generate Warnings on Undeclared Functions (-Xforce-declarations, -Xforce-prototypes)

-Xforce-declarations
-X9
Generate warnings if a function is used without a previous declaration.

-Xforce-prototypes
-X9=3
Generate warnings if a function is used without a previous prototype declaration.

These options are useful to make C a more strongly typed language. This option is ignored when compiling C++ modules.

5.4.56 Suppress Assembler and Linker Parameters (-Xforeign-as-ld)

-Xforeign-as-ld
(no numeric equivalent)
Cause the driver to call an assembler and linker without any implicit parameters.

This allows third-party assemblers and linkers to be used with the Wind River compiler. The **-W** *xfile* option may be used to specify a foreign assembler or linker (*5.3.30 Substitute Program or File for Default (-W xfile)*, p.44), the **-W a** option to pass parameters to the assembler (*5.3.25 Pass Arguments to the Assembler (-W a,arguments, -W :as:,arguments)*, p.42), and the **-W l** option to pass

parameters to the linker (*5.3.27 Pass Arguments to Linker (-W l,arguments, -W :ld:,arguments)*, p.43).

5.4.57 Convert Double and Long Double (-Xfp-long-double-off, -Xfp-float-only)

-Xfp-float-only
-X70=2

Force **double** and **long double** to be the same as **float**.

Synonym: **-Xno-double**.

-Xfp-long-double-off
-X70

Force **long double** to be the same as **double** on machines where they differ.

Synonym: **-Xno-long-double**.

NOTE: If this option is used, libraries containing functions with floating point parameters must be recompiled. For safety, recompile all libraries to avoid missing any such functions. Also, operation of library routines designed to process a suppressed type is undefined.

5.4.58 Specify Minimum Floating Point Precision (-Xfp-min-prec...)

-Xfp-min-prec-double
-X3=0

Use **double** as the minimum precision in expressions and for floating point arguments. Lesser precisions are used in expressions if the **-Xdialect-ansi** option is used. If prototypes are used, use the declared precision for arguments, unless the **-Xextend-args** option is used.

Synonym: **-Xuse-double**.

-Xfp-min-prec-float
-X3=1

Use **float** as the minimum precision in expressions and for floating point arguments.

Synonym: **-Xuse-float**.

-Xfp-min-prec-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 **-Xdialect-ansi** option is used.

If prototypes are used, use the declared precision for arguments, unless the **-Xextend-args** option is also given.

Synonym: **-Xuse-long-double**.

NOTE: If this option is used, libraries containing functions with floating point parameters must be recompiled. For safety, recompile all libraries to avoid missing any such functions. Also, operation of library routines designed to process a suppressed type is undefined.

5.4.59 Generate .frame_info for C functions (-Xframe-info)

-Xframe-info

Force the compiler to generate **.frame_info** sections for C funtions. Use this option when compiling mixed C/C++ programs in which C++ exceptions may propagate back through C functions. For more information, see *23.8 .frame_info sections*, p.355.

5.4.60 Include Filename Path in Debug Information (-Xfull-pathname)

-Xfull-pathname
-X125

Include the path prefix in filenames in debug information (specifically, in the **.file** assembler directive). Without this option, only the filename is included.

5.4.61 Control GNU Option Translator (-Xgcc-options-...)

-Xgcc-options-on

Enable automatic translation of GNU compiler (GCC) options. This is the default.

-Xgcc-options-off

Disable automatic translation of GCC options.

-Xgcc-options-verbose

Display all translations. Valid only if translation is enabled (**-Xgcc-options-on**).

When **-Xgcc-options-on** is enabled, GCC option flags from the command line or makefile are parsed and, if possible, translated to equivalent Wind River Compiler options. Translations are determined by the tables in the file **gcc_parser.conf**.

5.4.62 Treat All Global Variables as Volatile (-Xglobals-volatile)

See *5.4.91 Treat All Variables As Volatile (-Xmemory-is-volatile, -X...-volatile)*, p.93.

5.4.63 Do Not Pass #ident Strings (-Xident-off)

-Xident-on
-X63=0
Pass **#ident** strings to the assembler. This is the default.

-Xident-off
-X63
Do not pass **#ident** strings to the assembler.

Synonym: **-Xno-ident**.

5.4.64 Enable Strict implementation of IEEE754 Floating Point Standard (-Xieee754-pedantic)

-Xieee754-pedantic
-X82=1
Enable strict implementation of the IEEE754 floating point standard at some cost in performance. Specifically,

- Do not optimize a divide by a constant to a multiply of its reciprocal.
- Do not use floating multiply-add instructions on architectures where more bits are kept in intermediate results than is defined by the standard.
- Do not optimize *x-x* to zero so that possible NaN values are preserved.
- Do less equal and greater equal comparisons with behavior for NaN values as defined by the standard.

This option is equivalent to **-Xfp-pedantic**. (See *5.4.51 Specify Degree of Conformance to the IEEE754 Standard (-Xfp-fast, -Xfp-normal, -Xfp-pedantic)*, p.76.)

5.4.65 Control Template Instantiation (-Ximplicit-templates...)

-Ximplicit-templates
-X207=0
Instantiate each template in each module where it is used or referenced. This is the default.

-Ximplicit-templates-off
-X207=1
Instantiate templates only where explicit instantiation syntax is used.

Synonym: **-Xno-implicit-template**.

For further discussion, see *5.4.26 Mark Sections as COMDAT for Linker Collapse (-Xcomdat)*, p.66 and *Templates*, p.223.

C++ only.

5.4.66 Treat #include As #import (-Ximport)

-Ximport
-X75
Treat all **#include** directives as if they are **#import** directives. This means that any include file is included only once.

5.4.67 Ignore Missing Include Files (-Xincfile-missing-ignore)

-Xincfile-missing-ignore
-X172
This option, which suppresses error reporting, is effective only when used with **-Xmake-dependency** (*5.4.87 Show Make Rules (-Xmake-dependency)*, p.91). It causes preprocessing to continue even when a required header is not found. If **-Xincfile-missing-ignore** is used with **-Xmake-dependency=2** or **-Xmake-dependency=6**, the preprocessor issues a warning (but not an error) when a required system file (**#include <***filename***>**) is not found.

5.4.68 Initialize Local Variables (-Xinit-locals=mask)

-Xinit-locals=*mask*
-X87=*mask*

Initialize all local variables to zero or the value specified with **-Xinit-value** at every function entry. *mask* is a bit mask specifying the kind of variables to be initialized.

mask may be given in hex, e.g., **-Xinit-locals=0x9**. Mask bits may be OR-ed to select more than one. Undefined mask bits are ignored.

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.

5.4.69 Control Generation of Initialization and Finalization Sections (-Xinit-section)

This option controls generation of sections for run-time initialization and finalization invocation, including constructor and destructor functions and global class objects in C++. For more information, see *15.4.8 Run-time Initialization and Termination*, p.260.

-Xinit-section=0
-X91=0

Suppress generation of initialization and finalization sections. This option is not recommended and may result in incorrect run-time behavior.

-Xinit-section
-Xinit-section=1
-X91
-X91=1

Create **.ctors** and **.dtors** sections containing pointers to initialization and finalization functions, sorted by priority. This is the default.

Initialization and finalization functions are designated with attribute specifiers. See *constructor, constructor(n) Attribute*, p.141 and *destructor, destructor(n) Attribute*, p.142.

-Xinit-section=2
-X91=2
Create **.init$***nn* and **.fini$***nn* code sections containing calls to initialization and finalization functions, sorted by priority. Provides compatibility with previous versions of the compiler, including recognition of old-style function prefix designations for initialization and finalization functions.

Synonym: **-Xuse-.init**.

5.4.70 Control Default Priority for Initialization and Finalization Sections (-Xinit-section-default-pri)

-Xinit-section-default-pri=*n*
-X175=*n*
Assign the default priority for constructor and destructor functions and for C++ global class objects. The specified priority *n* applies to functions referenced in **.ctors**, **.dtors**, **.init**, and **.fini** sections. Functions with lower priority numbers execute first.

5.4.71 Define Initial Value for -Xinit-locals (-Xinit-value=n)

-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 can be used to initialize variables to some invalid or recognizable value that might produce a memory access error.

The value *n* is 32-bits, right-justified, zero-filled and may be specified as a decimal or hexadecimal number (**0***x*...).

5.4.72 Inline Functions with Fewer Than n Nodes (-Xinline=n)

-Xinline=*n*
-X19=*n*
Set the limit on the number of nodes for automatic inlining. Because the compiler collects functions until **-Xparse-size** KBytes of memory is used, the inlined function does not need to be defined before the function using it. See *__inline__ and inline Keywords*, **p.135** and *Inlining (0x4)*, p.191 for a discussion of inlining.

See *5.4.142 Control Loop Unrolling (-Xunroll=n, -Xunroll-size=n)*, p.112 for a definition of node count. (Assembly files saved with **-S** show the number of nodes for each function.) For purposes of automatic inlining, nodes that do not correspond to an operator or operand are not counted. Hence setting **-Xinline** to 0 inlines no functions automatically, and setting **-Xinline** to 1 inlines only "dummy" functions containing no code.

Defaults: **-Xinline** is 10 by default. **-XO** sets **-Xinline** to 40 by default.

NOTE: Inlining occurs only if optimization is selected by using the **-XO** or **-O** option.

5.4.73 Allow Inlining of Recursive Function Calls (-Xinline-explicit-force)

-Xinline-explicit-force
-Xinline-explicit-force=*n*
-X163
-X163=*n*

Inline recursive function calls up to *n* times. The default is 50. If this option is not used, the compiler inlines a function at most once.

If this option is combined with **-Xinline=0**, the compiler inlines only functions declared within a C++ class or with **inline**, **__inline__** , or **#pragma inline**.

This option is overridden by **-Xexplicit-inline-factor**. (See *5.4.49 Control Inlining Expansion (-Xexplicit-inline-factor)*, p.75.) By default, **-Xexplicit-inline-factor=3** is in effect for C++ programs; C++ programmers who want to use **-Xinline-explicit-force** should therefore specify **-Xexplicit-inline-factor=0**.

5.4.74 Enable Interworking (-Xinterwork)

-Xinterwork
-X40

Allow compiling mixed ARM and Thumb code with modules containing routines that can be called by routines for the other processor state.

5.4.75 Enable Intrinsic Functions (-Xintrinsic-mask)

-Xintrinsic-mask=*n*
-X154=*n*

Enable specified intrinsic functions. See *6.6 Intrinsic Functions*, p.144 for details.

5.4.76 Set longjmp Buffer Size (-Xjmpbuf-size=n)

-Xjmpbuf-size=*n*
-X201=*n*

C++ only. Set the size in bytes of the buffer allocated for **setjmp** and **longjmp** when using exceptions. The default size as determined by the compiler should usually be sufficient.

5.4.77 Create and Keep Assembly or Object File (-Xkeep-assembly-file, -Xkeep-object-file)

-Xkeep-assembly-file
(no numeric equivalent)

Always create and keep a **.s** file without the need for a separate compilation with the **-S** option. This option can be used with the **-c** option to create both assembly and object files at once.

-Xkeep-object-file
(no numeric equivalent)

Always create and keep a **.o** file without the need for a separate compilation with the **-c** option. This is needed only when a single file is compiled, assembled, and linked in one step, because in this case the driver deletes intermediate assembly and object files automatically.

5.4.78 Enable Extended Keywords (-Xkeywords=mask)

-Xkeywords=*mask*
-X78=*mask*

Recognize new keywords according to *mask*, a bit mask specifying which keywords to add.

mask may be given in hex, e.g., **-Xkeywords=0x9**. Mask bits may be OR-ed to select more than one. Undefined mask bits are ignored.

0x01 **extended** (C only)
0x02 **pascal** (C only)
0x04 **inline** (this keyword *always* available in C++)
0x08 **packed**
0x10 **interrupt** (C only)

See *6. Additions to ANSI C and C++* for more information on these keywords.

5.4.79 Disable Individual Optimizations (-Xkill-opt=mask, -Xkill-reorder=mask)

NOTE: These options are deprecated and should be used only on the advice Customer Support.

-Xkill-opt=*mask*
-X27=*mask*
Disable individual target-independent optimizations.

-Xkill-reorder=*mask*
-X28=*mask*
Disable individual target-dependent optimizations in the **reorder** program.

mask is a bit mask with one bit for each optimization type. *mask* may be given in hex, e.g., **-Xkill-opt=0x12**. Multiple optimizations can be disabled by OR-ing their mask bits. Undefined mask bits are ignored.

Both target-independent and target-dependent optimizations are described in *10. Optimization*. The name of each optimization is followed by its mask bit in parentheses, e.g. Tail recursion (0x2).

For *mask* bit values for **-Xkill-opt**, see *10.3 Target-Independent Optimizations*, p.190, and for **-Xkill-reorder**, *10.4 Target-Dependent Optimizations*, p.202. *mask* bit values are given in parentheses after the name of each optimization.

Either the **-O** or **-XO** option must be given to enable optimization before either of these **-Xkill-...** options can be used. To compile with almost no optimization, do not specify **-O** or **-XO.**

Two minor optimizations required by the code generation algorithms cannot be disabled: local strength reduction (e.g., multiply by power of 2 becomes shift or add) and simple branch optimization (e.g., branches to branches).

5.4.80 Wait For License (-Xlicense-wait)

-Xlicense-wait
-X138
If a license is not available, request that the compiler wait and retry once a minute, rather than returning with an error.

5.4.81 Generate Warnings On Suspicious/Non-portable Code (-Xlint=mask)

-Xlint[*=mask*]
-X84[*=mask*]
Generate warnings when suspicious and non-portable C code is encountered. For C++ modules, see note below. The two usual cases are:

-Xlint enables all warnings (equivalent to **-Xlint=1**).

-Xlint=0xffffffff disables all present and future warnings (equivalent to **-Xlint=0** or the default of not using the option at all).

Individual warnings can be *disabled* by OR-ing the following values. In effect, **-Xlint=1** is assumed, enabling all warnings, and then individual warnings are disabled. *mask* may be given in hex, e.g., **-Xlint=0x1a**. Undefined bits are ignored.

`0x02`
Variable used before being set.

`0x04`
Label not used.

`0x08`
Condition always true/false, for example, **i==i**.

`0x10`
Variable/function not used.

`0x20`
Missing return expression.

`0x40`
Variable set but not used.

`0x80`
Statement not reached.

`0x100`
Conversion problems.

`0x200`

In non-ANSI mode, warn when the compiler selects an unsigned integral type for an expression which would be signed under ANSI mode. For example:

```
"a.c", line 3: warning (1671):
non-portable behavior: type of
`>' operator is unsigned only
in non-ANSI mode
```

`0x400`
Possibly assignment (=) should be comparison (==).

`0x1000`
Missing function declaration (equivalent to **-Xforce-declarations**).

`0x2000`
Possible redundant expression. (Examples: **x=x**, **x&x**, **x|x**, **x/x**.)

11. The Lint Facility gives an example of a program which generates most of the **-Xlint warnings**.

See also the **__lint** macro in *6.1 Preprocessor Predefined Macros*, p.117 to avoid use of non-ANSI extensions in header files.

NOTE: For C++, **-Xlint** is equivalent to **-Xsyntax-warning-on**. (See *5.4.137 Disable Certain Syntax Warnings (-Xsyntax-warning-...)*, p.110.)

5.4.82 Allocate Static and Global Variables to Local Data Area (-Xlocal-data-area=n)

-Xlocal-data-area=*n*
-X115=*n*

Allocate the static and global variables which are defined in a module and referenced as least once in a contiguous block of memory, called the local data area (LDA), and make fast, efficient references to those variables via a temporary base register selected by the compiler.

n specifies the maximum of the LDA, and defaults to 64 bytes. (If *n* is greater than the default, references to variables in the LDA will be less efficient.)

The optimization does not apply to unreferenced variables. **-Xlocal-data-area** should be used with caution in multithreaded environments. To restrict the optimization to static variables, use **-Xlocal-data-area-static-only**; VxWorks developers are strongly advised to use this option.

See *14.4 Local Data Area (-Xlocal-data-area)*, p.247 for additional information.

Synonym: **-Xlocal-struct**.

NOTE: If at least one variable in the LDA has an initial value, the LDA is in the **.data** section; otherwise it is in the **.bss** section. Because **-Xlocal-data-area** is nonzero by default, uninitialized static and global variables which are referenced at least once are not stored in a **.bss** section. To store such variables in **.bss**, use **-Xlocal-data-area=0**.

5.4.83 Restrict Local Data Area Optimization to Static Variables (-Xlocal-data-area-static-only)

-Xlocal-data-area-static-only
-X166

Apply the local data area optimization only to static variables; do not optimize global variables. See *14.4 Local Data Area (-Xlocal-data-area)*, p.247 for information about this optimization.

5.4.84 Do Not Assign Locals to Registers (-Xlocals-on-stack)

-Xlocals-on-stack
-X5

By default, the compiler attempts to assign all local variables to registers. If **-Xlocals-on-stack** is given, only variables declared with the **register** keyword are assigned to registers.

5.4.85 Expand Macros in Pragmas (-Xmacro-in-pragma)

-Xmacro-in-pragma
-X157

Expand preprocessor macros in **#pragma** directives.

5.4.86 Warn On Undefined Macro In #if Statement (-Xmacro-undefined-warn)

-Xmacro-undefined-warn
-X171

Generate a warning when an undefined macro name occurs in a **#if** preprocessor directive.

5.4.87 Show Make Rules (-Xmake-dependency)

-Xmake-dependency
-Xmake-dependency=*mask*
-X156, -X156=*mask*

Generate a list of include files required to build each object file. Example:

```
main.o: main.c stdio.h
    command list
```

This output means that **main.c** and **stdio.h** are required to build the target **main.o**. A list of make commands follows the dependency.

mask, which defaults to 1, is a bit mask—always interpreted as hexadecimal—of which the four least significant bits are meaningful: the fourth (least significant) bit, if set to 1, means that all required files are shown; this is the default. The third bit means that only files enclosed in double quotation marks (**#include "***filename***"**) are shown. (If both the third and the fourth bits are set, the fourth overrides the third.) The second bit means that compilation continues after the dependency list is generated (if this bit is 0, no output is emitted other than the list of dependencies) and that the dependency list is sent to a file (instead of the standard output). The first bit creates a "phony target" for each dependency other than the main file; this is a work-around for errors caused by missing header files and is provided for GNU compatibility. The **-o** option can be used to specify the output file, the target name, or both. Hence:

-Xmake-dependency=1
Same as **-Xmake-dependency**. Show all required include files. If **-o** is used, the target is the name specified with **-o**. Results go to the standard output unless -**Xmake-dependency-savefile=***filename* is specified. No further output is emitted.

-Xmake-dependency=2
Same as **-Xmake-dependency=1**, but show only files enclosed in double quotation marks (**#include "***filename***"**).

-Xmake-dependency=4
Same as **-Xmake-dependency=1**, but write the dependency list to a file and then continue with normal compilation. The output file can be specified with either **-o** or **-Xmake-dependency-savefile=***filename* (which overrides **-o**); otherwise it is called *filename***.d,** where *filename* is the name of the main source file, and is created in the directory where the compiler was invoked. If **-o** is used without **-Xmake-dependency-savefile**, the output file is the basename specified by **-o** with **.d** appended.

-Xmake-dependency=8
Same as **-Xmake-dependency=1**, but output a phony target for each dependency other than the main file.

The bits can be OR-ed to combine options. Example:

-Xmake-dependency=6
Show only files enclosed in double quotation marks (**-Xmake-dependency=2**); write output to a file, then continue with normal compilation (**-Xmake-dependency=4**).

-Xmake-dependency=a
Show only files in double quotation marks (**-Xmake-dependency=2**) and output phony targets (**-Xmake-dependency=8**).

-Xmake-dependency=c
Output phony targets (**-Xmake-dependency=8**); write output to a file, then continue with normal compilation (**-Xmake-dependency=4**).

-Xmake-dependency=e
Show only files enclosed in double quotation mark**s (-Xmake-dependency=2**); output phony targets (**-Xmake-dependency=8**); write output to a file, then continue with normal compilation (**-Xmake-dependency=4**).

Ordinarily, the preprocessor returns an error and stops when a required file is not found. To continue preprocessing when files are missing, use **-Xmake-dependency** with **-Xincfile-missing-ignore** (*5.4.67 Ignore Missing Include Files (-Xincfile-missing-ignore), p.82*).

5.4.88 Specify Dependency Name or Output File (-Xmake-dependency-...)

This option is valid only when used with -**Xmake-dependency.**

-Xmake-dependency-target=*string*
Change the target name in the rule emitted by -**Xmake-dependency** to *string* **(**instead of using the name of the object file). To specify multiple target names, repeat the -**Xmake-dependency-target** option on the command line.

-Xmake-dependency-savefile=*filename*
Specify the output file for **-Xmake-dependency**.

5.4.89 Set Template Instantiation Recursion Limit (-Xmax-inst-level=n)

-Xmax-inst-level[=*n*]
-X216[=*n*]
C++ only. Set the maximum level for recursive instantiation of templates. Without this option, an error is emitted when a default level of 50 is reached. With this option, but without a value *n*, the limit is 100.

5.4.90 Set Maximum Structure Member Alignment (-Xmember-max-align=n)

-Xmember-max-align=*n*
-X88=*n*
Set the maximum byte boundary to which structure members will be aligned. If the natural alignment of a member is less than *n*, the natural alignment is used for it. See *pack Pragma*, p.129 and the *__packed__ and packed Keywords*, p.137 for details. See also *5.4.133 Set Minimum Structure Member Alignment (-Xstruct-min-align=n)*, p.109.

The default value of *n* is dependent on the processor as described in *8. Internal Data Representation*.

Synonym: **-Xstruct-max-align**.

5.4.91 Treat All Variables As Volatile (-Xmemory-is-volatile, -X...-volatile)

-Xmemory-is-volatile
-X4
-X4=7
Treat all variables as volatile.

-Xglobals-volatile
-X4=1
Treat all global variables as volatile.

-Xstatics-volatile
-X4=2
Treat all static variables as volatile.

-Xpointers-volatile
-X4=4
Treat all pointer accesses as volatile.

These options tell the compiler not to perform optimizations that can cause device drivers or other systems to fail. By default, the compiler 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 the compiler, unaware of the change, continues to use the old value stored in a register. While these situations can now be handled with the **volatile** keyword, the **-X4 options** allow compilation of older programs.

To combine these options, use the sum of their values with a single occurrence of the option flag. For example, use **-X4=3** to treat all global and static variables as volatile. **-X4=7**, equivalent to **-X4** or **-Xmemory-is-volatile**, combines all of the options.

5.4.92 Warn On Type and Argument Mismatch (-Xmismatch-warning)

-Xmismatch-warning
-X2
-Xmismatch-warning=2
-X2=2

Generate a warning only (instead of a fatal error) when either pointers of different types, or pointers and integers, are mixed in expressions. Example:

```
long i1, i2 = &i1;
```

is invalid in ANSI C but is allowed in some non-ANSI dialects. This option is set implicitly by **-Xdialect-pcc (-X7=3)**.

If the option **-Xmismatch-warning=2** is given, the compiler also generates a warning instead of an error when identifiers are redeclared and when a function call has the wrong number of arguments.

This option is ignored when compiling C++ modules.

NOTE: -Xmismatch-warning and **-Xmismatch-warning=2** override the **-e** option. If either form of **-Xmismatch-warning** is used, mismatched types will only produce a warning, even if **-e** is used to increase the severity level of the diagnostic. See *5.3.8 Change Diagnostic Severity Level (-e)*, p.36.

5.4.93 Specify Section Name (-Xname-...)

Use the following options to specify the name of a default section.

-Xname-code=*name*

Set the section name for code.

-Xname-const=*name*
Set the section name for initialized constants.

-Xname-data=*name*
Set the section name for initialized **data**.

-Xname-eh=*name*
C++ only.
Set the section name for all exception-handling tables.

-Xname-rtti=*name*
C++ only.
Set the section name for all RTTI tables.

-Xname-sconst=*name*
Set the section name for initialized small **const**.

-Xname-sdata=*name*
Set the section name for initialized small **data**.

-Xname-string=*name*
Set the section name for strings.

-Xname-uconst=*name*
Set the section name for uninitialized constants.

-Xname-udata=*name*
Set the section name for uninitialized **data**.

-Xname-usconst=*name*
Set the section name for uninitialized small **const**.

-Xname-usdata=*name*
Set the section name for uninitialized small **data**.

-Xname-vtbl=*name*
C++ only.
Set the section name for all virtual-function tables.

Section names can also be specified using the **section** pragma. For example, setting **-Xname-code=.code** has the same effect as:

```
#pragma section CODE ".code"
```

For more information, see *section Pragma*, p.133.

5.4.94 Disable C++ Keywords namespace and Using (-Xnamespace-...)

-Xnamespace-on
-X219=0
Recognize the **namespace** and **using** keywords or constructs.

-Xnamespace-off
-X219
C++ only. Do not recognize the **namespace** and **using** keywords or constructs.

5.4.95 Enable Extra Optimizations (-XO)

-XO
-X26
Enable all standard optimizations plus the following:

-O
(*5.3.17 Optimize Code (-O)*, p.40)

-Xinline=40
(10 with **-O**; *5.4.72 Inline Functions with Fewer Than n Nodes (-Xinline=n)*, p.84)

-Xopt-count=2
(1 with **-O**; *5.4.97 Execute the Compiler's Optimizing Stage n Times (-Xopt-count=n)*, p.97)

-Xparse-size=6000
(3000 with **-O**; *5.4.99 Specify Optimization Buffer Size (-Xparse-size)*, p.97)

-Xrestart
(**off with -O**; *5.4.112 Restart Optimization From Scratch (-Xrestart)*, p.103)

-Xtest-at-both
(**-Xtest-at-bottom** with **-O**; *5.4.139 Specify Loop Test Location (-Xtest-at-...)*, p.111)

5.4.96 Use Old Inline Assembly Casting(-Xold-inline-asm-casting)

-Xold-inline-asm-casting
-X137
This option affects small arguments to **asm** macros (arguments with size less than **int**).

By default, the compiler does not extend such arguments to **int**. Prior to version 4.2, the compiler did extend such arguments to **int**. Use this option to

force the old behavior for compatibility with existing **asm** macros which depend on it.

5.4.97 Execute the Compiler's Optimizing Stage n Times (-Xopt-count=n)

-Xopt-count=*n*
-X25=*n*
Execute the compiler'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.**

5.4.98 Disable Most Optimizations With -g (-Xoptimized-debug-...)

-Xoptimized-debug-on
-X89=0
Do not disable optimizations when using **-g**. This is the default.

-Xoptimized-debug-off
-X89
When using the **-g** option to generate debug information, disable most optimizations and force line numbers in debug information to be in increasing order — assists with debuggers that cannot handle optimized code. See also *5.4.39 Disable debugging information Extensions (-Xdebug-mode=mask)*, p.70, and *5.4.40 Disable Debug Information Optimization (-Xdebug-struct-...)*, p.71.

Synonym: **-Xno-optimized-debug**.

5.4.99 Specify Optimization Buffer Size (-Xparse-size)

-Xparse-size=*n*
-X20=*n*
Delay code generation of functions until *n* KBytes of main memory is used for internal tables. By delaying generation, the compiler can perform interprocedural optimizations such as inlining and register tracking.

The default is 3000 KB (6000 KB if option **-XO** is used). The highest useful value for a module depends on many factors; it is not practical to calculate it (see the discussion of "limitations related to memory size" in *C. Compiler Limits* for some of the factors).

For very large and complex modules, experiment with larger values, e.g. **-Xparse-size=8000**, to see if code size or execution time is reduced.

NOTE: That using a value larger than available physical memory will cause excessive swapping and slow compilation.

5.4.100 Output Source as Comments (-Xpass-source)

-Xpass-source
-X11
Output the source as comments in the generated assembly language code.

5.4.101 Use Precompiled Headers (-Xpch-...)

C++ only. These options are disabled by default. At most one of **-Xpch-automatic**, **-Xpch-create**, and **-Xpch-use** can be enabled; if more than one is specified, all but the first are ignored. For more information, see *13.7 Precompiled Headers*, p.229.

-Xpch-automatic
Generate and use precompiled headers.

-Xpch-create=*filename*
Generate a precompiled header (PCH) file with specified name.

-Xpch-diagnostics
Generate an explanatory message for each PCH file that the compiler locates but is unable to use.

-Xpch-directory=*directory*
Look for PCH file in specified directory.

-Xpch-messages
Generate a message each time a PCH file is created or used.

-Xpch-use=*filename*
Use specified PCH file.

5.4.102 Generate Position-Independent Code for Shared Libraries (-Xpic)

-Xpic
-X62
For VxWorks RTP application development. Allows a single copy of a shared library, loaded in a single memory location, to be called by different programs. RTP shared-library code must be compiled with this option.

5.4.103 Treat All Pointer Accesses As Volatile (-Xpointers-volatile)

See *5.4.91 Treat All Variables As Volatile (-Xmemory-is-volatile, -X...-volatile)*, p.93.

5.4.104 Control Interpretation of Multiple Section Pragmas (-Xpragma-section-...)

These options control the compiler's behavior when multiple **#pragma section** directives are used with different parameters for the same section class. The default is -**Xpragma-section-first.**

For more information, see *section and use_section Pragmas*, p.233.

-Xpragma-section-first
If this option is in effect when a variable or function is defined, the compiler uses the *earliest* currently-valid **section** pragma that specifies a non-default location for the variable or function.

-Xpragma-section-last
If this option is in effect when a variable or function is defined, the compiler uses the *last* currently-valid **section** pragma that specifies a non-default location for the variable or function.

5.4.105 Preprocess Assembly Files (-Xpreprocess-assembly)

-Xpreprocess-assembly
Invoke C preprocessor on assembly files before running the assembler.

5.4.106 Suppress Line Numbers in Preprocessor Output (-Xpreprocessor-lineno-off)

-Xpreprocessor-lineno-off
-X165

Suppress line-number information in the preprocessor output. Use this with the **-E** option (send preprocessor output to standard output) when line-number information is not needed.

5.4.107 Use Old Preprocessor (-Xpreprocessor-old)

-Xpreprocessor-old
-X155

Use the preprocessor from release 4.3. When **-Xpreprocessor-old** is specified, **vararg** macros are not supported and the following options are not available: **-Xmake-dependency**, **-Xmake-dependency-...**, **-Xmacro-in-pragma**, and **-Xcpp-dump-symbols**.

This option is valid only when compiling C modules or when compiling C++ modules with the **-Xc++-old** option.

5.4.108 Generate Profiling Code for the RTA Run-Time Analysis Tool Suite (-Xprof-...)

-Xprof-all
-X123=3

Collect count and time data.

-Xprof-all-fast
-X123=6

Collect count and time data for each function, but not for pairs of functions, so no hierarchical profile will be available.

-Xprof-count
-X123=2

Collect count data only, incrementing a counter for line of code executed (actually, for each basic block).

-Xprof-coverage
-X123=8

Like **-Xprof-count**, except just set the counter to one for each basic block executed instead of counting the number of executions.

-Xprof-time
-X123=1
Collect time data only.

-Xprof-time-fast
-X123=4
Collect time data for each function, but not for pairs of functions, so no hierarchical profile will be available.

These options cause the compiler to generate profiling code for the RTA. To be profiled, a function must be instrumented. The compiler inserts instrumentation code based on the following options. Every module to be profiled must be compiled with one of these options.

NOTE: In addition to an **-Xprof-***type* option, you must use the **-g** option to generate debug information.

Besides interactively analyzing the profile information generated by these options using the RTA, you may feed the collected data back to the compiler to improve optimization based on the actual execution of the target program. See *5.4.110 Optimize Using RTA Profile Data (-Xprof-feedback)*, p.101.

Do not use these options with the older pair of profiling options **-Xblock-count** (*5.4.11 Insert Profiling Code (-Xblock-count)*, p.60) and **-Xfeedback** (*5.4.52 Optimize Using Profile Data (-Xfeedback=file)*, p.76).

A function, its parent, and its children must all be compiled with the same **-Xprof-***type* option or the results are undefined.

5.4.109 Select Target Executable for Use by -Xprof-feedback (-Xprof-exec)

-Xprof-exec=*pathname*
(no numeric equivalent)
pathname must be the full pathname of a target executable for which profile data is present in the RTA database directory specified with **-Xprof-feedback**. See *5.4.110 Optimize Using RTA Profile Data (-Xprof-feedback)*, p.101 for details.

5.4.110 Optimize Using RTA Profile Data (-Xprof-feedback)

-Xprof-feedback=*pathname*
(no numeric equivalent)
pathname must specify an RTA database directory (not a file). Use the profiling information in that database (the latest "snapshot") to optimize for faster code.

See the *5.4.53 Set Optimization Parameters Used With Profile Data (-Xfeedback-frequent, -Xfeedback-seldom)*, p.77, to control how the profile data affects optimization.

The snapshot selected depends on **-Xprof-snapshot** (*5.4.111 Select Snapshot for Use by -Xprof-feedback (-Xprof-snapshot)*, p.103) and **-Xprof-exec** (*5.4.109 Select Target Executable for Use by -Xprof-feedback (-Xprof-exec)*, p.101) as follows:

-Xprof-exec	-Xprof- snapshot	Snapshot Selected
No	No	Use latest snapshot in the database.
No	Yes	Use snapshot named by **-Xprof-snapshot.** If a snapshot with the given name is present for more than one executable, use the latest.
Yes	No	Use latest snapshot for the executable specified by **-Xprof-exec**.
Yes	Yes	Use snapshot named by **-Xprof-snapshot**. Report an error if no snapshot with the given name is present for the executable specified by **-Xprof-exec**.

NOTE: This option is used in conjunction with the **-Xprof-...** options (*5.4.108 Generate Profiling Code for the RTA Run-Time Analysis Tool Suite (-Xprof-...)*, **p.100**). Do not use this option with the older pair of profiling options **-Xblock-count** (*5.4.11 Insert Profiling Code (-Xblock-count)*, p.60) and **-Xfeedback** (*5.4.52 Optimize Using Profile Data (-Xfeedback=file)*, p.76).

Also, the selected snapshot must include basic block count data, that is, the executed code must have been compiled with **-Xprof-all**, **-Xprof-all**, or **-Xprof-count**. The options **-Xprof-time**, **-Xprof-time-fast**, and **-Xprof-coverage** do not produce the data required for feedback-driven optimization.

5.4.111 Select Snapshot for Use by -Xprof-feedback (-Xprof-snapshot)

-Xprof-snapshot=*string*
(no numeric equivalent)

string must name a snapshot in the RTA database directory specified with **-Xprof-feedback**. See **-Xprof-feedback** (*5.4.110 Optimize Using RTA Profile Data (-Xprof-feedback)*, p.101) for details.

5.4.112 Restart Optimization From Scratch (-Xrestart)

-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 when the information needed is not 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 later in the compilation.

The compiler 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.**

5.4.113 Generate Code for the Run-Time Error Checker (-Xrtc=mask)

-Xrtc=*mask*
-X64=*mask*

With no *mask*, this option directs the compiler to insert checking code for all checks made by the Run-Time Error Checker. Use the *mask* to select specific checks rather than all.

5.4.114 Enable Run-time Type Information (-Xrtti, -Xrtti-off)

-Xrtti
-X205=1

Enable run-time type information. This is the default.

There are two approaches to generating run-time type information for a class:

- Compile all modules with **-Xrtti** and also with **-Xcomdat** (*5.4.26 Mark Sections as COMDAT for Linker Collapse (-Xcomdat)*, p.66): the run-time type information will be emitted for every module but will be marked COMDAT and collapsed to a single instance by the linker. This is the preferred method.
- For a class declaring one or more virtual functions, compile only the module defining the *key function* for the class with **-Xrtti**. Key functions are described in *Virtual Function Table Generation—Key Functions*, p.171.

-Xrtti-off
-X205=0

C++ only. Disable run-time type information. Using this option will save space because the compiler does not need to create type tables.

Synonym: **-Xno-rtti**.

5.4.115 Pad Sections for Optimized Loading (-Xsection-pad)

-Xsection-pad
-X152

Allow the linker to pad loadable sections for optimized loading.

5.4.116 Generate Each Function in a Separate CODE Section Class (-Xsection-split)

-Xsection-split
-X129
-Xsection-split-off
-X129=0

Generate a separate **CODE** section class for each function in the module. The default is **-Xsection-split-off**; a single module generates only one **CODE** section class containing the code for all functions for that module.

By default, with **-Xsection-split** enabled, the multiple **CODE** section classes will all still be named **.text** (absent the use of **.section** pragmas). While linking, a specific **.text** section for a given function may be singled out using the linker command language syntax:

object-filespec(*input-section-name*[*symbol*] ,...)

(where the "[" and "]" characters are required and do not mean "optional" in this case).

Example: if object file **test.o** contains functions **f1** and **f2**, then the **.text** section for **f1** may be specified by:

```
test.o(.text[f1])
```

NOTE: This option is especially useful in combination with **-Xremove-unused-sections** to reduce code size. See *Remove Unused Sections (-Xremove-unused-sections)*, p.373.

5.4.117 Disable Generation of Priority Section Names (-Xsect-pri-...)

-Xsect-pri-on
-X122=0
Enable section names of the form "...**$***n*". See *23.6 Sorted Sections*, p.354 for use of this form. This is the default.

-Xsect-pri-off
-X122
Disable generation of section names of the form "...$*n*" for use by third-party assemblers or linkers unable to process this form of name.

5.4.118 Control Listing of -X Options in Assembly Output (-Xshow-configuration=n)

-Xshow-configuration=0
Compiler-generated assembly listings (saved with the **-S** option) do not show **-X** options. This is the default.

-Xshow-configuration=1
Assembly listings contain **-X** options, but only user-configurable options are shown; internal compiler flags are suppressed.

5.4.119 Print Instantiations (-Xshow-inst)

-Xshow-inst
-X212
C++ only. Print to **stderr** a list of all template instantiations made during compilation. See also *5.4.65 Control Template Instantiation (-Ximplicit-templates...)*, p.82 and *Templates*, p.223.

5.4.120 Show Target (-Xshow-target)

-Xshow-target

dcc C and **dplus** C++ driver option. Display the target processor "**-t** option" on standard output, but do not compile any file.

5.4.121 Optimize for Size Rather Than Speed (-Xsize-opt)

-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. For character arrays, **-Xstring-align=value** will override **-Xsize-opt**. See the description of array alignment in *8.3 Arrays*, **p.166.**

5.4.122 Select Software Floating Point Emulation (-Xsoft-float)

-Xsoft-float
-X56

The implementation is a very fast call-based method. This option is controlled by DFP, which also selects which library to use, and should not usually be set explicitly by the user.

5.4.123 Enable Stack Checking (-Xstack-probe)

-Xstack-probe
-X10

Enable stack checking (probing). For users of the Run-Time Error Checker, this option is equivalent to **-Xrtc=4**.

NOTE: -Xstack-probe cannot be used with "interrupt" functions, that is, with a function named in an **interrupt** pragma or declared using the **interrupt** or **__interrupt__** keywords

5.4.124 Diagnose Static Initialization Using Address (-Xstatic-addr-...)

-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). This option is on by default.

5.4.125 Treat All Static Variables as Volatile (-Xstatics-volatile)

See *5.4.91 Treat All Variables As Volatile (-Xmemory-is-volatile, -X...-volatile)*, p.93.

5.4.126 Buffer stderr (-Xstderr-fully-buffered)

-Xstderr-fully-buffered
-X173
Buffer **stderr** using 10KB buffer. Use this option to reduce network traffic; **stderr** is unbuffered by default.

5.4.127 Terminate Compilation on Warning (-Xstop-on-warning)

-Xstop-on-warning
-X85
Terminate compilation on any warning. Without this option, only errors terminate compilation. (For both errors and warnings, compilation terminates after a small number of errors are output.)

5.4.128 Compile C/C++ in Pedantic Mode (-Xstrict-ansi)

-Xstrict-ansi
Compile in "pedantic" mode. This option is equivalent to **-Xdialect-strict-ansi**. For C, see *5.4.41 Specify C Dialect (-Xdialect-...)*, p.71. For C++, **-Xstrict-ansi** generates diagnostic messages when nonstandard features are used and

disables features that conflict with ANSI/ISO C++, including **-Xusing-std-on** and **-Xdollar-in-ident**.

Disabled by default.

5.4.129 Ignore Sign When Promoting Bit-fields (-Xstrict-bitfield-promotions)

-Xstrict-bitfield-promotions

Conform to the ANSI standard when promoting bit-fields. When a bit-field occurs in an expression where an **int** is expected, the compiler promotes the bit-field to a larger integral type. Unless this option is enabled, such promotions preserve sign as well as value. If **-Xstrict-bitfield-promotions** is specified, however, an object of an integral type all of whose values are representable by an **int** (that is, an object smaller than 4 bytes) is promoted to an **int**, even if the original type is unsigned.

-Xstrict-ansi or **-Xdialect-strict-ansi** implicitly enables **-Xstrict-bitfield-promotions** by default, but can be overridden with **-Xstrict-bitfield-promotions=0**.

See also *5.4.10 Specify Sign of Plain Bit-field (-Xbit-fields-signed, -Xbit-fields-unsigned)*, p.60.

5.4.130 Align Strings on n-byte Boundaries (-Xstring-align=n)

-Xstring-align=*n*
-X18=*n*

Align each string on an address boundary divisible by *n*. The default value is 4. See also *5.4.8 Specify Minimum Array Alignment (-Xarray-align-min)*, p.59.

5.4.131 Warn on Large Structure (-Xstruct-arg-warning=n)

-Xstruct-arg-warning=*n*
-X92=*n*

C only. Emit a warning if the size of a structure argument is larger than or equal to n bytes.

5.4.132 Control Optimization of Structure Member Assignments (-Xstruct-assign-split-...)

-Xstruct-assign-split-diff=*n*
-X147=*n*
-Xstruct-assign-split-max=*n*
-X146=*n*

These options control optimization of assignments of local **struct** variables. The compiler uses a number of techniques to optimize structure members (it uses registers, etc.). A structure can be assigned as a one or more blocks (depending on a number of factors) or member-by-member. However, block structure assignment disables member optimization, so options are available to control the type of structures that will assigned as a block.

By default, the assignment is member-by-member if the structure has 6 or fewer members and if the increase in assignments (over block assignments) is 3 or fewer. Otherwise, the structure is assigned as a block.

Use **-Xstruct-assign-split-max** to set the maximum number of members in a struct that may be assigned member-by-member.

Use **-Xstruct-assign-split-diff** to set the maximum number of additional assignments allowed. If member-to-member assignment involves a higher number of additional assignments than the number set by **-Xstruct-assign-split-diff**, a block assignment is performed.

5.4.133 Set Minimum Structure Member Alignment (-Xstruct-min-align=n)

-Xstruct-min-align=*n*
-X76=*n*

Force structures to begin on at least an *n* byte boundary. If any member in a structure has a greater alignment, the structure will be aligned on a boundary divisible by the size in bytes of the largest member.

See *pack Pragma*, p.129 and *__packed__ and packed Keywords*, p.137 for details. See also *5.4.90 Set Maximum Structure Member Alignment (-Xmember-max-align=n)*, p.93.

The default value of *n* is dependent on the processor as described in *8. Internal Data Representation*.

5.4.134 Suppress Warnings (-Xsuppress-warnings)

-Xsuppress-warnings
-X14

Suppress compiler warnings. Same as the **-w** option.

5.4.135 Swap '\n' and '\r' in Constants (-Xswap-cr-nl)

-Xswap-cr-nl
-X13
C only. Swap '**n**' and '**r**' in character and string constants. Used on systems where carriage return and line feed are reversed.

5.4.136 Set Threshold for a Switch Statement Table (-Xswitch-table...)

-Xswitch-table=*n*
-X143=*n*
Implement a **switch** statement using compares if there are fewer than *n* **case** labels in the **switch**, use a jump table if there are *n* or greater. This option is on by default with a value of 7.

-Xswitch-table-off
Do not use a jump table to implement a **switch** statement under any conditions.

5.4.137 Disable Certain Syntax Warnings (-Xsyntax-warning-...)

-Xsyntax-warning-on
-X215=0
Enable certain syntax warnings, for example, warning on a comma after the last enumerator. This is the default.

-Xsyntax-warning-off
-X215
C++ only. Disable these warnings.

5.4.138 Select Target Processor (-Xtarget)

-Xtarget
-X39=*n*This option is for internal use should usually not be set by the user. See *4. Selecting a Target and Its Components*.

5.4.139 Specify Loop Test Location (-Xtest-at-...)

-Xtest-at-both
-X6=2
Force the compiler 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 more space. Even if **-Xtest-at-both** is not set, other optimizations may cause the compiler 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.

5.4.140 Truncate All Identifiers After m Characters (-Xtruncate)

-Xtruncate=*m*
-X22=*m*
Truncate all identifiers after *m* characters. If *m* is zero, no truncation is done. This is the default.

5.4.141 Append Underscore to Identifier (-Xunderscore-...)

-Xunderscore-leading
-X71=1
Prefix every externally visible identifier with an underscore in the symbol table.

Synonym: **-Xleading-underscore**.

-Xunderscore-trailing
-X71=2
Suffix every externally visible identifier with an underscore in the symbol table.

Synonym: **-Xtrailing-underscore**.

-Xunderscore-surround
-X71=3
Prefix and suffix every externally visible identifier with an underscore in the symbol table.

Synonym: **-Xsurround-underscore**.

NOTE: The **-Xunderscore...** options are provided for use in linking code generated by the compiler with third-party libraries or with other tools requiring generated underscores.

The default value of this option is 0 (no extra underscore).

Because Wind River libraries are compiled with the default setting, setting this option to anything but the default will require recompiling every library used.

5.4.142 Control Loop Unrolling (-Xunroll=n, -Xunroll-size=n)

-Xunroll=*n*
-X15=*n*

Unroll small loops *n* times. Set to 2 by default. *n* must be a power of two. See *Loop Unrolling (0x8000)*, p.197.

NOTE: Some sufficiently small loops may be unrolled more than *n* times if total code size and speed is better.

-Xunroll-size=*n*
-X16=*n*

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. (There is also a small number of additional nodes for each function.) *n* is set to 20 by default. Assembly files saved with **-S** show the number of nodes for each function.

NOTE: Unrolling is done only if option **-O** or **-XO** is given to enable optimization

5.4.143 Runtime Declarations in Standard Namespace (-Xusing-std-...)

-Xusing-std-on

C++ only. Automatically search for runtime library declarations in the **std** namespace (as if "**using namespace std;**" had been specified in the source

code), not in global scope. This is the default behavior, but it is disabled by **-Xstrict-ansi**; use **-Xusing-std-on** on the command line to override **-Xstrict-ansi**.

This option allows you to use the newer C++ libraries, which are in the **std** namespace, without adding **using namespace std;** to legacy code.

-Xusing-std-off
Search for runtime library declarations in global scope unless an explicit **using namespace std;** is given.

5.4.144 Void Pointer Arithmetic (-Xvoid-ptr-arith-ok)

-Xvoid-ptr-arith-ok
-X167
Treat void pointers as **char *** for the purpose of arithmetic. For example:

```
some_void_ptr += 1;  /* adds 1 to some_void_ptr */
```

5.4.145 Define Type for wchar (-Xwchar=n)

-Xwchar=*n*
-X86=*n*
Define the type to which **wchar** will correspond. The desired *type* is given by specifying a value *n* equal to a value returned by the operator **sizeof(***type***, 2)**. See *sizeof Extension*, p.149. The default type is **long** integer (32 bits), that is, **-Xwchar=4**.

5.4.146 Control Use of wchar_t Keyword (-Xwchar_t-...)

-Xwchar_t-on
-X214=0
Enable the **wchar_t** keyword.

-Xwchar_t-off
-X214
C++ only. Disable the **wchar_t** keyword.

Synonym: **-Xno-wchar**.

5.5 Examples of Processing Source Files

The following examples show typical ways of compiling.

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

```
/* file1.c */
void outarg(char *);
int main(int argc, char **argv)
{
    while(--argc) outarg(*++argv);
    return 0;
}

/* file2.cpp */
#include <stdio.h>

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

    printf("arg #%d: %s\n",++count,arg);
}
```

5.5.1 Compile and Link

When compiling small programs such as this, the driver can be invoked to execute all four stages of compilation in one command. For example:

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

The driver preprocesses, compiles, and assembles the two files (one C and one C++), and links them together with the appropriate libraries to create a single executable file, by default called **a.out**. When more than one file is compiled to completion, object files are created and kept, in this case, **file1.o** and **file2.o**. When only one file is compiled, assembled, and linked, the intermediate assembly and object files are deleted automatically (see *5.4.77 Create and Keep Assembly or Object File (-Xkeep-assembly-file, -Xkeep-object-file)*, p.86 to change this).

If the target system supports 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 *15. Use in an Embedded Environment* for comments on executing programs in embedded environments.)

To execute the program on the host system using the WindISS simulator, compile the program with **windiss** specified on the command line—for example:

```
dplus -tARMES:windiss file1.c file2.cpp
```

Then run the program with WindISS:

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

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

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

To also enable optimization, use the **-O** option:

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

To convert the linked output to **S** records:

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

will produce file **srec.out** by default. See *29. D-DUMP File Dumper* for additional options and details.

5.5.2 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.c file2.cpp
```

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

Create the executable program as follows. Note that the driver is used to invoke the linker; this is convenient because defaults will be supplied as required based on the current target, for example, for libraries and **crt0.o**.

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

If **file2.cpp** is altered, **prog2** can be 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.

5.5.3 Assembly Output

It is frequently desirable to look at the generated assembly code. Two options are available for this purpose:

- The **-S** option stops compilation after generating the assembly and automatically names the file *basename***.s**, **file1.s** in this case:

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

- When using a command which generates an object file, the **-Xkeep-assembly-file** option will preserve the assembly file in addition to the object, naming it *basename***.s**.

The option **-Xpass-source** outputs the compiled 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 file2.cpp
```

5.5.4 Precompiled Headers

In C++ projects with many header files, you can often speed up compilation by using precompiled headers, enabled with the **-Xpch-...** options. See *13.7 Precompiled Headers*, p.229.

6 Additions to ANSI C and C++

6.1 Preprocessor Predefined Macros

The following preprocessor macros are predefined. The macros that do not start with two underscores ("__") are not defined if option **-Xdialect-strict-ansi** is given.

__BIG_ENDIAN__
Big-endian implementation.

__bool
The constant 1 if type **bool** is defined when compiling C++ code, otherwise undefined. Option **-Xbool-off** disables the **bool**, **true**, and **false** keywords. C++ only.

__CHAR_UNSIGNED__
Indicates that plain **char** characters are unsigned.

__cplusplus
The constant 199711 when compiling C++ code, otherwise undefined.

__DATE__
The current date in "*mm dd yyyy*" format; it cannot be undefined.

__DCC__
The constant 1.

__DCPLUSPLUS__
The constant 1 when compiling C++ code, otherwise undefined.

_DIAB_TOOL
Indicates the Wind River Compiler is being used.

__EABI__
ELF ABI object format.

__ETOA__
Indicates that full ANSI C++ is supported. Not defined when compiling C code or when an older version of the compiler is invoked.

__ETOA_IMPLICIT_USING_STD
Defined if **-Xusing-std-on** is enabled. Indicates that runtime library declarations are automatically searched for in the **std** namespace (not in global scope), regardless of whether **using namespace std;** is specified.

__ETOA_NAMESPACES
Defined if the runtime library uses namespaces.

__EXCEPTIONS
Exceptions are enabled. C++ only.

__FILE__
The current filename; it cannot be undefined.

__FUNCTION__
__FUNCTION__ is not really a preprocessor macro, but a special predefined identifier that returns the name of the current function (that is, the function in which the identifier occurs).

__LITTLE_ENDIAN__
Little-endian implementation.

__LITTLE_ENDIAN__
Little-endian implementation.

__LDBL__
The constant 1 if the type **long double** is different from **double**.

__LINE__
The current source line; it cannot be undefined.

__lint
This macro is not predefined; instead, define this when compiling to select pure-ANSI code in Wind River header files, avoiding use of any non-ANSI extensions.

__arm
Target flag used by various tools.

__THUMB__
Target flag used by various tools.

__nofp
No floating point support.

__PRETTY_FUNCTION__
__PRETTY_FUNCTION__ is not really a preprocessor macro, but a special predefined identifier that returns the name of the current function (that is, the function in which the identifier occurs). In C modules, **__PRETTY_FUNCTION__** always returns the same value as **__FUNCTION__**. For C++, **__PRETTY_FUNCTION__** may return additional information, such as the class in which a method is defined.

__RTTI
C++ only. Run-time type information is enabled.

__SIGNED_CHARS__
C++ only. Defined as 1 if plain **char** is signed. See *5.4.20 Specify Sign of Plain Char (-Xchar-signed, -Xchar-unsigned)*, p.63.

__softfp
Software floating point support.

__STDC__
The constant 0 if **-Xdialect-ansi** and the constant 1 if **-Xdialect-strict-ansi** is given. It cannot be undefined if **-Xdialect-strict-ansi** is set. For C++ modules it is defined as 0 in all other cases.

__STRICT_ANSI__
The constant 1 if **-Xdialect-strict-ansi or -Xstrict-ansi** is enabled.

__TIME__
The current time in "*hh*:*mm*:*ss*" format; it cannot be undefined.

__VERSION__
The version number of the compiler and tools, represented as a string.

__VERSION_NUMBER__
The version number of the compiler and tools, represented as an integer.

__wchar_t
The constant 1 if type **wchar_t** is defined when compiling C++ code, otherwise undefined. Option **-X-wchar-off** disables the **wchar_t** keyword.

6.2 Preprocessor Directives

The preprocessor recognizes the following additional directives.

#assert and #unassert Preprocessor Directives

The **#assert** and **#unassert** directives allow definition of preprocessor variables that do not conflict with names in the program namespace. These variables can be used to direct conditional compilation. The C and C++ preprocessors recognize slightly different syntax for **#assert** and **#unassert**.

Assertions can also be made on the command line through the **-A** option.

To display information about assertions at compile time, see *5.4.30 Dump Symbol Information for Macros or Assertions (-Xcpp-dump-symbols)*, p.67.

To make an assertion with a preprocessor directive, use the syntax:

#assert *name* **(***value***)**	C or C++
#assert *name*	C++ only

In the first form, *name* is given the value *value*. In the second form, *name* is defined but not given a value. Whitespace is allowed only where shown.

Examples:

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

To make an assertion on the command line, use:

-A *name***(***value***)**

Examples:

```
dcc -A "system(unix)" test.c        UNIX
dcc -A system\(unix\) test.c        UNIX
dcc -A system(unix) test.c          Windows
```

Assertions can be tested in an **#if** or **#elif** preprocessor directive with the syntax:

#if #*name* **(***value***)** C or C++

#if #*name* C only

A statement of the first form evaluates to true if an assertion of that name with that value has appeared and has not been removed. (A *name* can have more than one value at the same time.) A statement of the second form evaluates to true if an assertion of that name with *any* value has appeared.

Examples:

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

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

#unassert *name* C++ only

#unassert *name* **(***value***)** C++ only

#unassert #*name* **(***value***)** C only

The first form removes all definitions of *name*. The other forms remove only the specified definition.

Examples:

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

#error Preprocessor Directive

The **#error** preprocessor directive displays a string on standard error and halts compilation. Its syntax is:

#error *string*

Example:

```
#error "Feature not yet implemented."
```

See also *#info, #inform, and #informing Preprocessor Directives*, p.122 and *#warn and #warning Preprocessor Directives*, p.123.

#ident Preprocessor Directive (C only)

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

#import Preprocessor Directive

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 header files with protective **#ifdefs**, but using **#import** is much more efficient since the compiler does not have to open the file. Using the **-Ximport** command-line option will cause all **#include** directives to behave like **#import**.

#info, #inform, and #informing Preprocessor Directives

The **#info**, **#inform**, and **#informing** preprocessor directives display a string on standard error and continue compilation. Their syntax is:

```
#info string
#inform string
#informing string
```

Example:

```
#info "Feature not yet implemented."
```

See also *#error Preprocessor Directive*, p.121 and *#warn and #warning Preprocessor Directives*, p.123.

#warn and #warning Preprocessor Directives

The **#warn** and **#warning** preprocessor directives display a string on standard error and continue compilation. Their syntax is:

#warn *string*
#warning *string*

Example:

```
#warn "Feature not yet implemented."
```

See also *#error Preprocessor Directive*, p.121 and *#info, #inform, and #informing Preprocessor Directives*, p.122.

6.3 Pragmas

This section describes the pragmas supported by the compiler. A warning is issued for unrecognized pragmas.

Pragma directives are not preprocessed. Comments are allowed on pragmas.

In C++ modules, a pragma naming a function affects all functions with the same name, independently of the types and number of parameters—that is, independently of overloading.

align Pragma

#pragma align [**(** [[*max_member_alignment*], [*min_structure_alignment*] [, *byte-swap*]] **)**]

The **align** pragma, provided for portability, is a synonym for *pack Pragma*, p.129.

error Pragma

#pragma error *string*

Display *string* on standard error as an error and halt compilation. See also *info Pragma*, p.125 and *warning Pragma*, p.133.

global_register Pragma

#pragma global_register *identifier*=*register* , ...

This pragma forces a global or static variable to be allocated to a specific register. This can increase execution speed considerably when a global variable is used frequently, for example, the "program counter" variable in an interpreter.

identifier gives the name of a variable. *register* gives the name of the selected register in the target processor. See *9.6 Register Use*, p.180 for a list of valid register names.

The following rules apply:

- Only registers which are preserved across function calls may be assigned to global variables.
- When assigning several variables to registers, start by using the lowest preserved register available. Some targets cannot use lower preserved registers for automatic and register variables.
- Do not mix modules using global registers with modules not using them. Never call a function using global registers from a module compiled without them.
- **#pragma global_register** can be used to force the compiler to avoid specific registers in code generation by defining dummy variables as global registers in all modules.
- The pragma must appear before the first definition or declaration of the variable being assigned to a register.

NOTE: A convenient method of ensuring that all modules are compiled with the same global register assignments is to put all **#pragma global_register** directives in a header file, e.g. **globregs.h**, and then include that file with every compilation from the command line with the **-i** option, e.g. **-i=globregs.h**.

Examples:

```
#pragma global_register counter=register-name
char *counter;          /* allocated to the named register */

/* Force the compiler to avoid a named register. */
#pragma global_register __dummy=register-name
```

hdrstop Pragma

#pragma hdrstop

C++ only. Suppress generation of precompiled headers. Headers included after **#pragma hdrstop** are not saved in a parsed state. See *13.7 Precompiled Headers*, p.229 for more information.

ident Pragma

#**pragma ident** *string*

Insert a comment into the generated object file.

Example:

```
#pragma 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** section.

info Pragma

#pragma info *string*

Display *string* on standard error and continue compilation. See also *error Pragma*, p.123 and *warning Pragma*, p.133.

inline Pragma

#pragma inline *func* **,...**

Inline the given function whenever possible. The pragma must appear before the definition of the function. Unless cross-module optimization is enabled (**-Xcmo-**...), a function can be inlined only in the module in which it is defined.

In C++ modules, the **inline** function specifier is normally used instead. This specifier, however, also makes the function local to the file, without external linkage. Conversely, the **#pragma inline** directive provides a hint to inline the code directly to the code optimizer, without any effect on the linkage scope.

NOTE: The **inline** pragma has no effect unless optimization is selected (with the **-XO** or **-O** options).

Example:

```
#pragma inline swap

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

interrupt Pragma

#pragma interrupt *function* **,...**

Designate *function* as an interrupt function. Code is generated to save all general purpose scratch registers and to use a different return instruction.

Important interrupt Pragma Notes

- Floating point and other special registers, if present on the target, are not saved because interrupt functions usually do not modify them. If such registers must be saved in order to handle nested interrupts, use an **asm** macro to do so (see *7. Embedding Assembly Code*). To determine which registers are saved for a particular target, compile the program with the **-S** option and examine the resulting assembler file (it will have a **.s** extension by default).
- The compiler does not generate instructions to re-enable interrupts. If this is required to allow for nested interrupts, use an **asm** macro.
- See *5.4.123 Enable Stack Checking (-Xstack-probe), p.106* for when this option *cannot* be used with interrupt functions.
- This pragma must appear before the definition of the function. A convenient method is to put it with a prototype declaration for the function, perhaps in a header file.

  Example:

  ```
  #pragma interrupt trap

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

no_alias Pragma

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

Promise that the variable *var1* is not accessed in any manner (through pointers etc.) other than through the variable name; promise that the data at **var2* is only accessed through the pointer *var2*. This allows the compiler to better optimize references to such variables.

The pragma must appear after the definition of the variable and before its first use.

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**. See also *5.4.7 Assume No Aliasing of Pointer Arguments (-Xargs-not-aliased)*, p.59.

no_pch Pragma

#pragma no_pch

Suppress all generation of precompiled headers from the file where **#pragma no_pch** occurs. See *13.7 Precompiled Headers*, p.229, for more information.

no_return Pragma

#pragma no_return *function* **,...**

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

This pragma must appear before the first *use* of the function. A convenient method is to put it with a prototype declaration for the function, perhaps in a header file.

Example:

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

no_side_effects Pragma

#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 those named or 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.

This pragma must appear before the first *use* of the function. A convenient method is to put it with a prototype declaration for the function, for example, in a header file.

Contrast with *pure_function Pragma*, p.132, which also promises that a function does not use any global or static variables.

Example:

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

option Pragma

#pragma option *option* [*option* ...]

Where *option* is any of the **-g**, **-O**, or **-X** options (including the leading '**-**' character). This option makes it possible to set these options from within a source file.

These options must be at the beginning of the source file before any other source lines. The effect of other placement is undefined.

Note that some **-X** options are consumed by driver or compiler command-line processing before a source file is read. If an **-X** option does not appear to hve the intended effect, try it on the command line. If effective there, that option can not be used as a pragma.

pack Pragma

#pragma pack [(**[**[*max_member_alignment*]**,** [*min_structure_alignment*][**,** *byte-swap*]])]

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

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

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

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

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.

It is not possible to take the address of a byte-swapped member.

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

The form **#pragma pack** is equivalent to **#pragma pack(1,1,0)**. The form **#pragma pack( )** is equivalent to **#pragma pack(0,0,0)**.

The **align** pragma, provided for portability, is an exact synonym for **pack**.

An alternative method of specifying structure padding is by using *__packed__ and packed Keywords*, p.137.

Default values for *max_member_alignment* and *min_structure_alignment* can be set by using the **-Xmember-max-align** and the **-Xstruct-min-align** options. The order of precedence is values **-X** options lowest, then the **packed** pragma, and **__packed__** or **packed** keyword highest.

Restrictions and Additional Information

Note that if a structure is *not packed*, the compiler will insert extra padding to assure that no alignment exception occurs when accessing multi-byte members because the processor requires that multi-byte variables be aligned on 4-byte boundaries; see *5.4.6 Specify Minimum Alignment for Single Memory Access to Multi-byte Values (-Xalign-min=n)*, p.58.

When a structure is *packed*, because the processor requires that multi-byte values be aligned (**-Xalign-min > 1**), the following restrictions apply:

- Access to multi-byte members will require multiple instructions. (This is so even if a member is aligned as would be required within the structure because the structure may itself be placed in memory at a location such that the member would be unaligned, and this cannot be determined at compile time.)
- **volatile** members cannot be accessed atomically. The compiler will warn and generate multiple instructions to access the **volatile** member. Also, "compound" assignment operators to **volatile** members, such as **+=**, **|=**, etc., are not supported. For example, assuming **i** is a **volatile** member of packed structure **struct1**, then the statement:

  ```
  struct1.i += 3;
  ```

  must be recoded as:

  ```
  struct1.i = struct1.i + 3;
  ```

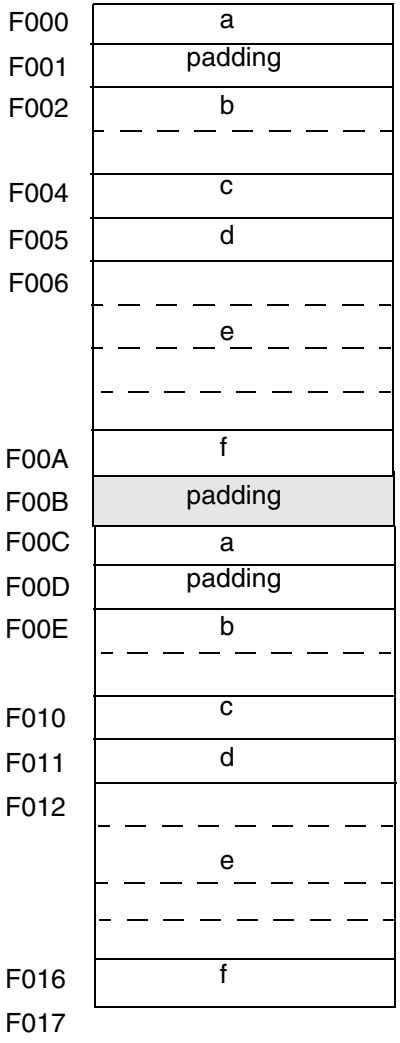
In addition, for packed structures, an **enum** member will use the smallest type sufficient to represent the range**,** see *5.4.47 Specify enum Type (-Xenum-is-...)*, p.73.

Examples

Later examples depend on earlier examples in some cases.

```
#pragma pack(2,2)
struct s0 {
    char a;                 1 byte at offset 0, 1 byte padding
    short b;                2 bytes at offset 2
    char c;                 1 byte at offset 4
    char d;                 1 byte at offset 5
    int e;                  4 bytes at offset 6
    char f;                 1 byte at offset 10
};                          total size 11, alignment 2
```

If two such structures are in a section beginning at offset 0xF000, the layout would be:




```
#pragma pack(1)
struct S1 {
    char c1
    long i1;
```

Same as **#pragma pack(1,1)**, no padding.

1 byte at offset 0
4 bytes at offset 1

```
    char d1;                    1 byte at offset 5
};                              total size 6, alignment 1

#pragma pack(8)                 Use "natural" packing for largest member.
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 packing on machines which cannot
struct S3 {                         access multi-byte values on odd-bytes.
    char  c3;                   1 byte at offset 0, 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 {                     Using pragma from prior example.
    char c4;                    1 byte at offset 0, 1 byte padding
};                              total size 2, alignment 2 since
                                    min_member_alignment is 2 above

#pragma pack(8)                 "Natural" packing since S3 is 8 bytes long.
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)                 Set to default packing.
```

pure_function Pragma

`#pragma pure_function` *function* **`,...`**

Promises that each function does not modify or use any global or static data. Helps the compiler generate better code, for example, in optimization of common sub-expressions containing identical function calls. Contrast with *no_side_effects*

Pragma, p.128, which only promises that a function does not modify global variables.

This pragma must appear before the first *use* of the function. A convenient method is to put it with a prototype declaration for the function, perhaps in a header file.

Example:

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


section Pragma

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

The **#pragma section** directive defines sections into which variables and code can be placed. It also defines how objects in sections are addressed and accessed.

This pragma must appear before the declaration (for functions, before the prototype if present) of all variables and all functions to which it is to apply.

The **section** pragma is discussed in detail in *14. Locating Code and Data, Addressing, Access*.

use_section Pragma

#pragma use_section *class_name variable* **,...**

Selects the section class into which a variable or function is placed. A section class is defined by **#pragma section**.

This pragma must appear before the declaration (for functions, before the prototype if present) of all variables and all functions to which it is to apply.

The **use_section** pragma is discussed in detail in *14. Locating Code and Data, Addressing, Access*.

warning Pragma

#pragma warning *string*

Display *string* on standard error as a warning and continue compilation. See also *error Pragma*, p.123, and *info Pragma*, p.125.

weak Pragma

#pragma weak *symbol*

Mark *symbol* as **weak**.

When a **#pragma weak** for a symbol is given in the module defining the symbol, it is a *weak definition*. When the **#pragma weak** is in a module using but not defining it, it is a *weak reference*.

Because this pragma is ultimately processed by the assembler, it may appear anywhere in the source file.

A weak symbol resembles a global symbol with two differences:

- When linking, a weak definition with the same name as a global or common symbol is not considered a duplicate definition; the weak symbol is ignored.
- If no module is present to define a symbol, unresolved weak references to the symbol have a value of zero and remain undefined in the symbol table after linking, and no error is reported.

Note while a symbol may be defined in more than one module as long as at most one of the definitions is global or common while the rest (or all) are weak, the linker resolves references to the first instance of the symbol it encounters. Consider the following scenario. Function **foo()** uses *x*, which is declared weak in library 1 and global in library 2. If library 1 is searched first, the weak version of *x* will be used. On the other hand, if library 2 is subsequently linked (because, for example, another function uses it), then the global version of *x* will replace the weak version.

#pragma weak is incompatible with local data area (LDA) allocation; using **#pragma weak** with **-Xlocal-data-area** or **-Xlocal-data-area-static-only** enabled will produce a warning and temporarily disable LDA. See *5.4.82 Allocate Static and Global Variables to Local Data Area (-Xlocal-data-area=n)*, p.89, and *14.4 Local Data Area (-Xlocal-data-area)*, p.247.

6.4 Keywords

The following additional keywords are recognized by the compiler.

__asm and asm Keywords

Used to embed assembly language (see *7. Embedding Assembly Code*) and use the information found in *Assigning Global Variables to Registers*, p.147.

__attribute__ Keyword

See *6.5 Attribute Specifiers*, p.139.

extended Keyword (C only)

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

Example:

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

__inline__ and inline Keywords

The **__inline__** and **inline** keywords provide a way to replace a function call with an inlined copy of the function body. The **__inline__** keyword is intended for use in C modules but is disabled in strict-ANSI mode. The **inline** keyword is normally used in C++ modules but can also be used in C if the option **-Xkeywords=0x4** is given (*5.4.78 Enable Extended Keywords (-Xkeywords=mask)*, p.86).

__inline__ and **inline** make the function local (**static**) to the file by default. Conversely, the **#pragma inline** directive provides a hint to inline the code directly to the code optimizer, without any effect on the linkage scope. Use **extern** to make an **inline** function public.

NOTE: Functions are not inlined, even with an explicit **#pragma inline**, or **__inline__** or **inline** keyword unless optimization is selected with the **-XO** or **-O** options.

Note that using **-O** will automatically inline functions of up to 10 nodes (including "empty" functions), and **-XO** will automatically inline functions of up to 40 nodes. See how these values are controlled in *5.4.72 Inline Functions with Fewer Than n Nodes (-Xinline=n)*, p.84**.** An explicit pragma or keyword can be used to force inlining of a function larger than the value set with implicitly or explicitly with **-Xinline**.

See *Inlining (0x4)*, p.191, for a complete discussion of all inlining methods.

Example:

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

    inc(&x);
```

The function call will be replaced with

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

__interrupt__ and interrupt Keywords (C only)

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 *interrupt Pragma*, p.126. The keyword **interrupt** can also be used; see *5.4.78 Enable Extended Keywords (-Xkeywords=mask)*, p.86.

NOTE: See why this cannot be used with interrupt functions, *5.4.123 Enable Stack Checking (-Xstack-probe), p.106*).

Example:

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

long long Keyword

The compiler supports 64-bit integers for all ARM microprocessors. 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** suffix. A suffix is required because constants are of type **int** by default.

Example:

```
long long mask_nibbles (long long x)
{
    return (x & 0xf0f0f0f0f0f0f0f0LL);
}
```

NOTE: Bit-fields are not permitted in variables of type **long long**.

__packed__ and packed Keywords

`__packed__` (([*max_member_alignment*], [*min_structure_alignment*] [, *byte-swap*]))

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 *pack Pragma*, p.129 for treatment of 0 values, defaults, and restrictions.

The *max_member_alignment* value specifies the maximum alignment of any member in the structure. If the natural alignment of a member is less than *max_member_alignment*, the natural alignment is used. See *8. Internal Data Representation* for more information about alignments and padding.

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

Default values for *max_member_alignment* and *min_structure_alignment* can be set by using the **-Xmember-max-align** and the **-Xstruct-min-align** options. The order of precedence is values **-X** options lowest, then the **packed** pragma, and **__packed__** or **packed** keyword highest.

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 *pack Pragma*, p.129 for defaults for missing parameters and for additional examples.

Examples:

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

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

__packed__ (4) struct s3 {      maximum alignment 4
    char c;
    int  i;                     starts at offset 4
};                              total size 8 bytes

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

For the C compiler only, constant expressions (in addition to simple constants) can be specified as arguments to the **__packed__** or **packed** keyword.

pascal Keyword (C only)

If the option **-Xkeywords=***x* is used with bit 1 set in *x* (e.g., **-Xkeywords=0x2**), the compiler 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 (for example, MC68000), the called function clears the argument stack space instead of the caller.

__typeof__ Keyword (C only)

__typeof__(*arg***)**, where *arg* is either an expression or a type, behaves like a defined type. Examples:

```
__typeof__(int *) x;
__typeof__(x) y;
```

The first statement declares a variable **x** whose type is the type of pointers to integers, while the second declares a variable **y** of the same type as **x**. Note that **typeof** (without underscores) is not supported.

6.5 Attribute Specifiers

Attribute specifiers, formed with the **__attribute__** keyword, assign extra-language properties to variables, functions, and types. They can specify packing, alignment, memory placement, and execution options. When you have a choice between an attribute specifier and an equivalent pragma, it is preferable to use the attribute specifier.

Attribute specifiers have the form **__attribute__((** *attribute-list* **))**, where *attribute-list* is a comma-delimited list of *attributes*. Supported attributes, some of which include parameters in parentheses, are described in the sections that follow.

An attribute specifier can appear in a variable or function declaration, function definition, or type definition; or following any variable within a list of variable declarations. Multiple attribute specifiers should be separated by whitespace.

When an attribute specifier modifies a function, it can appear before or after the return type. Examples:

```
__attribute__((pure)) int foo(int a, b);
int __attribute__((no_side_effects)) bar(int x);
```

When an attribute specifier modifies a **struct**, **union**, or **enum**, it can appear immediately before or after the keyword, or after the closing brace. Example:

```
struct b {
    char b;
    int a;
} __attribute__(( aligned(2) )) str1;
```

For non-structure fields, the specifier can be placed anywhere before or immediately following the identifier name:

```
__attribute__(( aligned(2) )) int foo;
int __attribute__(( aligned(4) )) bar;
int foobar __attribute__(( aligned(8) ));
```

Placement of a specifier determines how the attribute is applied. Example:

```
// align a and b on 4-byte boundaries
__attribute__(( aligned(4) )) char a='a', b='b';

// force alignment only for c
char __attribute__(( aligned(4) )) c='c', d ='d';

// force alignment only for f
char e='e', f __attribute__(( aligned(4) )) ='f';
```

If an attribute specifier modifies a **typedef**, it applies to all variables declared using the new type:

```
typedef __attribute__(( aligned(4) )) char AlignedChar;

// a and b are aligned on 4-byte boundaries
AlignedChar a='a', b='b';
```

To eliminate naming conflicts between attributes and preprocessor macros, any attribute name can be surrounded by double underscores. For example, **aligned** and **__aligned__** are synonyms; **__attribute__((aligned(2)))** is equivalent to **__attribute__((__aligned__(2)))**.

NOTE: The placement of attribute specifiers can be misleading. For example:

```
int last_func() {
...
} __attribute__((noreturn))       // modifies foo, not last_func

int foo() {
...
}
```

This example is confusing because *in type definitions*, the attribute specifier can follow the closing brace. But in function definitions, the attribute specifier must appear directly before or after the return type.

When an attribute takes a numeric parameter, the parameter can be a simple constant or a constant expression. Example:

```
__attribute__(( aligned(sizeof(double)) )) int x[32];
```

In this example, the constant expression **sizeof(double)** is used as a parameter to the **aligned** attribute.

absolute Attribute (C only)

__attribute__((absolute)) indicates that a **const** integer variable is an absolute symbol. Example:

```
const int foo __attribute__((absolute)) = 7;
```

This declaration means that **foo** appears in the symbol table and always represents the value 7; no memory is allocated to store **foo**.

aligned(n) Attribute

To specify byte alignment for a variable or data structure, use:

```
__attribute__ (( aligned(n) ))
```

where *n* is a power of two. Example:

```
// align structure on 8-byte boundary
__attribute__(( aligned(8) )) struct a {
    char b;
    int a;
} str1;
```

This is often combined with the *packed Attribute*, p.143. Example:

```
struct b {
    char b;
    int a;
} __attribute__(( aligned(2), packed )) str2;
```

You can force alignment for a specific element within a structure:

```
struct c {
    int k;
    __attribute__ (( aligned(8) )) char m;  // align m on 8 bytes
} str3;
```

But special alignment for members of a *packed* structure is ignored:

```
struct c {

    int k;
    __attribute__ (( aligned (8) )) char m;  // alignment ignored
} __attribute__((packed)) str4;
```

Nested alignment attributes are preserved within a **struct** or **union**.

constructor, constructor(n) Attribute

A *constructor*, or *initialization*, function is executed before the entry point of your application—that is, before **main( )**. To designate a function as a constructor with default priority, use:

```
__attribute__ ((constructor))
```

To designate a function as a constructor with a specified priority, use:

```
__attribute__ (( constructor(n) ))
```

where *n* is a number between 0 and 65535. Specifying a priority level allows you to control the order in which initialization functions execute; the lower the value of *n*, the earlier the function executes. For more information, see *15.4.8 Run-time Initialization and Termination*, p.260.

deprecated, deprecated(string) Attribute (C only)

Causes the compiler to issue a warning when the marked function, variable, or type is referenced.

```
__attribute__ ((deprecated))
__attribute__ ((deprecated(string) ))
```

The optional *string* is included with the warning message.

destructor, destructor(n) Attribute

A *destructor*, or *finalization*, function is executed after the entry point of your application or after **exit( )**. To designate a function as a destructor with default priority, use:

```
__attribute__ ((destructor))
```

To designate a function as a destructor with a specified priority, use:

```
__attribute__ (( destructor(n) ))
```

where *n* is a number between 0 and 65535. Specifying a priority level allows you to control the order in which finalization functions execute; the lower the value of *n*, the earlier the function executes. For more information, see *15.4.8 Run-time Initialization and Termination*, p.260.

noreturn, no_return Attribute

To indicate that a function will never return to the caller, use:

```
__attribute__ ((noreturn))
```

This allows the compiler to remove unnecessary code intended for returning execution to the caller on exit. The **no_return** attribute is equivalent to **no return**.

no_side_effects Attribute

This attribute is a less restrictive version of **pure** (see *pure, pure_function Attribute*, p.143). **__attribute__((no_side_effects))** indicates that a function does not modify any global data.

packed Attribute

This attribute specifies alignment for types and data structures. **__attribute__((packed))** tells the compiler to use the smallest space possible for the data to which it is applied. Example:

```
struct b {
     char b;
     int a ;
}  __attribute__ ((packed)) str1;
```

When used with **aligned**, the **packed** attribute takes precedence as discussed in *aligned(n) Attribute*, p.141.

pure, pure_function Attribute

This attribute indicates that a function does not modify or use any global or static data and that it accesses only data passed to it as parameters. Using **__attribute__((pure))** allows the compiler to perform optimizations such as global common subexpression elimination. The **pure_function** attribute is equivalent to **pure**. If this attribute is applied to a function that has side effects, run-time behavior may be indeterminate.

See also *no_side_effects Attribute*, p.143.

section(name) Attribute

To specify a linker section in which to place a function or variable, use:

```
__attribute__ (( section("name") ))
```

This creates a section called *name* and places the designated code in it. Example:

```
// place func1 in a section called foo
void func1(void) __attribute__(( section("foo") ));
```

For variables, the section is created as a read-write data segment. For functions, the section is created as a read-execute code segment. There are no options to change

the properties of the section. For greater control over sections, use **#pragma section** (see *14. Locating Code and Data, Addressing, Access*).

An attempt to mix types of information in a single section (for example, constant data in a section reserved for code or variables) produces an error (dcc1793). In this example, the compiler assumes from the first statement that the section **.mydata** is intended to be of the **DATA** section class, whereas the second statement assumes that **.mydata** will be a **CONST** section class:

```
__attribute__((section(".mydata"))) int var = 1;
__attribute__((section(".mydata"))) const int const_var = 2;
```

NOTE: In some cases, the compiler may not honor an attempt to use the **section** attribute to place initialized data into a section intended for uninitialized data, and vice-versa. For example, in the following code:

```
__attribute__((section(".bss"))) int x = 3;
```

x will be assignedto the **.data** section, not **.bss**.

See Table 14-1 on page 237 for a list of sections and section classes.

There is no cross-module verification that section names are used consistently. Incorrect usage, including typographical errors, cannot be detected until link time.

6.6 Intrinsic Functions

The compiler implements the following intrinsic functions to give access to specific ARM instructions. See the processor manufacturer's documentation for details on machine instructions.

Intrinsic functions can be selectively disabled with the **-Xintrinsic-mask=***n* (**-X154=***n*) option, where *n* is a bit mask that can be given in hex; mask bits can be **OR-ed** to select more than one. *n* defaults to 0xf.

Note:

- Functions taking **long long** arguments first sign-extend their arguments to 64 bits.

- These functions are not prototyped and return a 32-bit **int** (or **void**) by default, except that functions taking a **long long** argument return **long long** (or **void**). A prototype may be used to define a different return type.

Function	Mask	Description
alloca (*integral***)**	0x800000	Allocates temporary local stack space for an object of size integral. Returns a pointer to the start of the object. The allocated memory is released at return from the current function.
__alloca (*integral***)**		Same as **alloca( )**, but cannot be disabled.

Function	Mask	Assembly Code
__ffi (*int var***)**	0x10	`clz rd, (var)`
__mar (*long long a***)**	0x11	`mar acc0, (a Lo_32), (a Hi_32)`
__mia (*int a***, *int b*)**	0x11	`mia acc0, (a), (b)`
__miabb (*int a***, *int b*)**	0x11	`miabb acc0, (a), (b)`
__miabt (*int a***, *int b*)**	0x11	`miabt acc0, (a),( b)`
__miaph (*int a***, *int b*)**	0x11	`miaph acc0, (a), (b)`
__miatb (*int a***, *int b*)**	0x11	`miatb acc0, (a), (b)`
__miatt (*int a***, *int b*)**	0x11	`miatt acc0, (a), (b)`
__mra (*void***)**	0x11	`mra (a Lo_32), (a Hi_32),acc0` (Returns **long long**.)
__qadd (*int a***, *int b*)**	0x10	`qadd rd, (a), (b)`
__qdadd (*int a***, *int b*)**	0x10	`qdadd rd, (a), (b)`
__qdsub (*int a***, *int b*)**	0x10	`qdsub rd, (a), (b)`
__qsub (*int* a**, *int b*)**	0x10	`qsub rd, (a), (b)`
__smlabb (*int a***, *int b*, *int c*)**	0x10	`smlabb rd, (b), (c), (a)`

Function	Mask	Assembly Code
__smlabt (*int a***, *int b***, *int c***)**	0x10	`smlabt rd, (b), (c), (a)`
__smlalbb (*long long a***, *int b***, *int c***)**	0x10	`smlalbb (a Lo_32), (a Hi_32), (b), (c)`
__smlalbt (*long long a***, *int b***, *int c***)**	0x10	`smlalbt (a Lo_32), (a Hi_32), (b), (c)`
__smlaltb (*long long a***, *int b***, *int c***)**	0x10	`smlaltb (a Lo_32), (a Hi_32), (b), (c)`
__smlaltt (*long long a***, *int b***, *int c***)**	0x10	`smlaltt (a Lo_32), (a Hi_32), (b), (c)`
__smlatb (*int a***, *int b***, *int c***)**	0x10	`smlatb rd, (b), (c), (a)`
__smlatt (*int a***, *int b***, *int c***)**	0x10	`smlatt rd, (b), (c),( a)`
__smlawb (*int a***, *int b***, *int c***)**	0x10	`smlawb rd, (b), (c), (a)`
__smlawt (*int a***, *int b***, *int c***)**	0x10	`smlawt rd, (b), (c), (a)`
__smulbb (*int a***, *int b***)**	0x10	`smulbb rd, (a) (b)`
__smulbt (*int a***, *int b***)**	0x10	`smulbt rd, (a), (b)`
__smultb (*int a***, *int b***)**	0x10	`smultb rd, (a), (b)`
__smultt (*int a***, *int b***)**	0x10	`smultt rd, (a), (b)`
__smulwb (*int a***, *int b***)**	0x10	`smulwb rd, (a), (b)`
__smulwt (*int a***, *int b***)**	0x10	`smulwt rd, (a), (b)`

6.7 Other Additions

C++ Comments Permitted

C++ style comments beginning with **//** are allowed by default. To disable this feature, use **-Xdialect-strict-ansi**. Example:

```
int number1bits (int i)     // Count the number of 1 bits
{                           // in "i".
    int n = 0;

    while (i != 0) {
        i &= (i - 1);
        n ++;
    }
    return n;
}
```


Dynamic Memory Allocation with alloca

The **alloca(***size***)** and **__alloca(***size***)** functions are provided to dynamically allocate temporary stack space inside a function. Example:

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

p = alloca(1000);
```

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

See *6.6 Intrinsic Functions*, p.144 for additional details.

Binary Representation of Data

The compiler recognizes variables and constants that are given in binary format. For example, it will accept the following:

```
unsigned int x = 0b00001010;
```

Note that the compiler does not recognize the following format:

```
unsigned int x = 00001010b;
```

Use of binary representation in C may make your code non-portable.

Assigning Global Variables to Registers

You can assign a global variable to a preserved register by placing **asm("***register-name***")** or **__asm("***register-name***")** immediately after the variable name in the declaration. Example:

```
int some_global_var asm("r11");
```

This assigns the variable **some_global_var** to **r11**. Local variables cannot be assigned in this way.

__ERROR__ Function

The **__ERROR__( )** function produces a compile-time error or warning if it is seen by the code generator. This is useful for making compile-time checks beyond those possible with the preprocessor—e.g. ensuring that the sizes of two structures are the same, as shown in the example below. If the **__ERROR__( )** function is placed after an **if** statement that is not executed unless the assertion fails, the optimizer removes the **__ERROR__( )** function and no error is generated. (The optimizer must be enabled (at any level) for this technique to work.)

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 warning - compilation will continue

1 error - compilation will continue but will stop after the entire file has been processed

2 fatal error - compilation is aborted

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

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

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

When **__ERROR__( )** is used in C++ code, it must be declared like this:

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

sizeof Extension

The **sizeof** operator 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
char	C: 1 (char is **unsigned** by default) C++: 44
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	18
pointer to any type	19
array of any type	22
struct, union	C: 23 C++: same as **class**, 32
function	25
class	C++: 32
reference	C++: 33
enum	C++: 34

Examples:

```
i = sizeof(long ,2)        /* type of long: i = 6 */
j = sizeof(short,1)        /* alignment of short: j = 2 */
```

vararg Macros

The preprocessor supports several styles of **variadic** macro, including ANSI C draft, C99, and GNU. Use of **vararg** macros is illustrated below:

```
va_arg.c:
// C draft
#define debug(...)    fprintf(stderr, __VA_ARGS__)
#define showlist(...) puts(#__VA_ARGS__)
#define report(test, ...) ((test)?puts(#test):\
                     printf(__VA_ARGS__))
// C99
#define foo(string1, ...) printf(string1, ## __VA_ARGS__, ":end")
// GNU
#define bar(string2, args...) printf(string2, ## args, ":end")

debug("Flag");
debug("X = %d\n", x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
foo("start");
bar("begin");

> dcc -E va_arg.c
# 1 "va_arg.c" 0

fprintf(stderr, "Flag") ;
fprintf(stderr, "X = %d\n", x) ;
puts("The first, second, and third items.") ;
((x>y)?puts("x>y"):       printf( "x is %d but y is %d", x, y)) ;
printf("start", ":end") ;
printf("begin", ":end") ;
>
```

7 Embedding Assembly Code

7.1 Introduction

There are three approaches to embedding assembly code in source files: flexible **asm** macros, simple but less flexible **asm** strings, and *direct functions* for embedding machine code.

WARNING: When embedding assembly code with any method, you must use only scratch registers. See *9.6 Register Use*, p.180 to determine the scratch registers.

If optimization is enabled, even hand-inserted assembly language may be optimized. See *7.4 Reordering in asm Code*, p.160

NOTE: The compiler recognizes extended GNU inline syntax (e.g. register usage specification) but does not translate it. When extended syntax is encountered, the compiler issues an error message.

The **asm** and **__asm** keywords provide a way to embed assembly code within a compiled program. Either keyword may be used to introduce an assembly string or assembly macro as defined below, but **asm** is not defined in C modules if the **-Xdialect-strict-ansi** option is used. In the text below, whenever **asm** is used, **__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 to define an **asm** macro that inlines different assembly code sections, depending on the types of arguments given. The following two sections discuss both methods. *7.5 Direct Functions*, p.161 provide a third way to embed code by using integer values. The following table contrasts the three method.

Table 7-1 **Methods for Embedding Assembly Code**

Method	Implementation	Calling Conventions, Parameters
asm string	Expanded inline where encountered. Functions containing **asm** strings with labels may not be inlined more than once per function.	None — difficult to access source variables.
asm macro	Expanded inline where called. Functions containing **asm** macros may be inlined without restriction.	Parameters matched by type per storage mode lines. May return a value.
Direct function	Always inlined where called.	All normal calling conventions are followed. May return a value.

To confirm that embedded assembly code has been included as desired, compile with the **-S** option and examine the resulting **.s** file.

The examples in this chapter apply to both C and C++.

7.2 asm Macros

While **asm** strings (described in *7.3 asm String Statements*, p.158) can be useful for embedding simple assembly fragments, they are difficult to use with variables inside the assembly code. **asm** macros provide a more flexible way to embed assembly code in compiled programs.

asm Macro Syntax

An **asm** macro definition looks much like a function definition, including a return type and parameter list, and function body.

The syntax is:

```
asm [volatile] [return-type] macro-name ( [ parameter-list ] )
{
% storage-mode-list                (must start in column 1)
! register-list                    ("!" must be first non-whitespace)
      asm-code

}                                  (must start in column 1)
```

where:

- **volatile** prevents instructions from being interspersed or moved before or after the ones in the macro.
- *return-type* is as in a standard C function. *For a macro to return a value of the given type, the assembly code must put the return value in an appropriate register as determined by the calling conventions.* See *9.5 Returning Results*, p.179 for details.
- *macro-name* is a standard C identifier.
- *parameter-list* is as in a standard C function, using either old style C with just names followed by separate type declarations, or prototype-style with both a type and a name for each parameter. Parameters should not be modified because the compiler has no way to detect this and some optimizations will fail if a parameter is modified.
- *storage mode line* begins with a "**%**" which must start in column 1. The *storage-mode-list* is used mainly to describe parameters and is described below. A macro with no parameters and no labels does not require a storage mode line.
- *register-list* is an optional list of scratch registers, each specified as a double-quoted string, or the string "**call**" if the macro makes a call, separated

by commas. Specifying this list enables the compiler to generate more efficient code by invalidating only the named registers. Without a *register-list*, the compiler assumes that all scratch registers are used by the **asm** macro. See *Register-List Line*, p.156 for details.

- *asm-code* is the code to be generated by the macro.
- final right "}"closes the body; it must start in column 1.

The compiler treats an **asm** macro much like an ordinary function with unknown properties:

- 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.

However, because the **asm** macro is by definition inlined, it is not possible to take the address of an **asm** macro.

The compiler discards any invocation of an empty **asm** macro (one with no storage mode line and no assembler code). This may be useful for macros used for debugging purposes.

NOTE: An **asm** macro must be defined in the module where it is to be used before its use. Otherwise the compiler will treat it as an external function and, assuming no such function is defined elsewhere, the linker will issue an unresolved external error.

In C++, forward declarations of **asm** macros are not permitted. Hence, while static member functions can be **asm** macros, the **asm** keyword must occur in the function definition, not in the class declaration.

Storage Mode Line — Describing Parameters and Labels

The storage mode line is not required if a macro has no parameters and no labels.

For a macro with parameters, a storage mode line is required to describe the methods used to pass the parameters to the macro. Currently, for ARM targets, all parameters are passed in registers for convenience. A storage mode line is also required if the macro generates a label.

Every parameter name in the *parameter-list* must occur exactly once in a single storage mode line. The form of the *storage-mode-line* is:

%[reg | con | lab] *name*, ...; **[reg | con | lab]** *name*, . . . ; ...

where:

reg
Introduces a list of one or more parameters. Every parameter name in the *parameter-list* must occur exactly once in the single storage mode line.

Arguments to a macro are assigned to registers following the usual calling conventions. For example, four **int** arguments will use registers **r0**, **r1**, **r2**, and **r3**. Other scratch registers may be used freely in the macro. This limits the maximum number of parameters to the number allowed by the registers used for parameters (see *9.3 Argument Passing*, p.177 and *9.6 Register Use*, p.180).

NOTE: If the compiler has already moved an argument to a preserved register, the compiler will use it from there in the macro rather than moving it to the usual parameter register. Therefore, *always* use a *parameter* name rather than a *register* name when coding a macro.

NOTE: Because arguments may be in preserved registers as just noted, macros should avoid use of preserved registers, even if saved and restored.

con
The parameter is a constant.

lab
A new label is generated. **lab** is not actually a storage mode — the *name* 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, the compiler will generate a unique label to substitute for the uses of the *name* in the macro.

Names of **long long** parameters must be appended with **!H** or **!L**—e.g. **someParameter!H**. This replaces the parameter with a register holding the most (**!H**) or least (**!L**) significant 32 bits. The register is chosen based on the compilation's endian mode.

"No Matching asm Pattern Exists"

The compiler error message "no matching asm pattern exists" indicates that no suitable storage mode was found for some parameter, or that a label was used in the macro but no **lab** storage mode parameter was present. For example, it would be an error to pass a variable to a macro containing only a **con** storage mode parameter.

Register-List Line

An **asm** macro body may optionally contain a *register-list line*, consisting of the character "**!**" in column 1 and an optional *register-list*. The *register-list* if present , is a list of scratch registers, each specified as a double-quoted string, or the string "**call**", separated by commas. Specifying this list enables the compiler to generate more efficient code by invalidating only the named registers. Without a *register-list*, the compiler assumes that all scratch registers are used by the **asm** macro. Also, if a *register-list* is specified and the assembly macro makes a call, the "**call**" string must also be specified to cause the link register to be saved and restored.

The *register-list* line must begin with a "**!**" character, which must be the first non-whitespace character on a line. The specification can occur anywhere in the macro body, and any number of times, however it is recommended that a single line be used at the beginning of the macro for clarity.

Supported scratch registers are **r0**, **r1**, **r2** and **r3**. Also, if a *register-list* line is specified and the assembly macro makes a call, then "**call**" must also be specified to cause the link register to be saved and restore around the macro. See *9.6 Register Use*, p.180 for more information about registers.

If the "**!**" is present without any list, the compiler assumes that no scratch or link registers are used by the macro.

NOTE: If supplied, the *register-list* must be complete, that is, must name all scratch registers used by the macro and must include "**call**" if the macro makes a call. Otherwise, the compiler will assume that registers which may in fact be used by the macro contain the same value as before the macro.

Also, as noted below, any comment on the *register-list* line must be a C-style comment ("**/* ... */**") because this line is processed by the compiler, not the assembler.

Comments in asm Macros

Any comment on the non-assembly language lines—that is, the **asm** macro function-style header, the "{" or "}" lines, or a *storage-mode* or *register-list* line—must be a C-style comment ("**/* ... */**") because this line is processed by the compiler, not the assembler.

Comments on the assembly language line may be either C style or assembler style. If C style, they are discarded by the compiler and are not preserved in the

generated **.s** assembly-language file. If assembler style, they are visible in the **.s** file on every instance of the expanded macro.

Assembler-style comments in **asm** macros are read by the preprocessor when the source file is processed. For this reason, apostrophes and quotation marks in assembler-style comments may generate warning messages.

Examples of asm Macros

In this example, a macro loops until the value at the address given by its parameter is non-zero and then returns the value at that address (**int** values are returned in register **r0**).

```
asm  int  get_data (volatile unsigned int *address_p)
{
% reg address_p; lab loop;
! "r10", "r11"               /* scratch registers used */
loop:
    ldr         r10,[r11,#0]
    cmp         r10,#0
    beq         loop
}

extern volatile unsigned int device_in;  /* input port */

int  test (volatile unsigned int *device_in_p)
{
    int data;
    data = get_data (device_in_p);
    return get_data (& device_in);
}
```

The above code was compiled with:

```
dcc -tARMES:cross -S -Xpass-source asm_macro.c
```

Extracts from the generated assembly code for the two macro calls follow.

```
    stmfd       r13,{r10,r11}
    sub         r13,r13,#48
    mov         r11,r0
#     data1 = get_data(device_in_p);
.L3:
    ldr         r10,[r11,#0]
    cmp         r10,#0
    beq         .L3
    ldr         r12,+data1
    str         r0,[r12,#0]
```

```
#     return get_data(& device_in);
    ldr         r10,=device_in
.L4
    ldr         r10,[r11,#0]
    cmp         r10,#0
    beq         .L4
    add         r13,r13,#48
    ldmea       r13,{r10,r11}
    mov         pc,lr
```

NOTE:

- The uniquely generated loop labels.
- The macro argument is always forced to a register. Before the first expansion, the address of **device_in_p** was loaded into **r11**. For the second expansion, **device_in** is loaded into **r10**.

7.3 asm String Statements

NOTE: **asm** string statements are primarily useful for manipulating data in static variables and special registers, changing processor status, etc., and are subject to several restrictions: no assumption can be made about register usage, non-scratch registers must be preserved, values may not be returned, some optimizations are disabled, and more. **asm** macro functions described above are recommended instead. See *Notes and Restrictions*, p.159 below.

An **asm** string statement provides a simple way to embed instructions in the assembly code generated by the compiler. Its syntax is:

asm[**volatile**] (*"string"*[**!** *register-list*]);

where *string* is an ordinary string constant following the usual rules (adjacent strings are pasted together, a "\" at the end of the line is removed, and the next line is concatenated) and *register-list* is a list of scratch registers (see *Register-List Line*, p.156). The optional **volatile** keyword prevents instructions from being moved before or after the string statement.

An **asm** string statement can be used wherever a statement or an external declaration is allowed. *string* will be output as a line in the assembly code at the

point in a function at which the statement is encountered, and so must be a valid assembly language statement.

If several assembly language statements are to be generated, they may either be written as successive **asm** string statements, or by using "**\n**" within the string to end each embedded assembly language statement. The compiler will not insert any code between successive **asm** string statements.

If an **asm** string statement contains a label, and the function containing the **asm** string is inlined more than once in some other function, a duplicate label error will occur. Use an **asm** macro with a storage mode line containing a **lab** clause for this case. See *7.2 asm Macros*, p.153.

Notes and Restrictions

asm string statements are primarily useful for tasks like changing processor status (as in the example above) and manipulating data in static variables and special registers. When using **asm** string statements, consider the following notes and restrictions:

- No assumptions may be made regarding register values before and after an **asm** string statement. For example, do not assume that parameters passed in registers will still be there for an **asm** string statement.
- The compiler does not expect an **asm** string statement to "return" a value. Thus, using an **asm** string statement as the last line of a function to place a value in a return register does not ensure that the function will return that value.
- The compiler assumes that non-scratch registers are preserved by **asm** string statements. If used, these registers must be saved and restored by the **asm** string statements.
- The compiler assumes that scratch registers are changed by **asm** string statements and so need not be preserved.
- Some optimizations are turned off when an **asm** string statement is encountered.
- A function containing an **asm** string statement is never inlined.
- Because the string contained in quotation marks is passed to the assembler exactly as is (after any pasting of continued lines), it must be in the format required for an assembly language line. Specifically, an instruction line must begin with a space, a tab, or a label. Assembler directives may start in column one but only if the assembler **-Xlabel-colon** option is enabled (see *Set Label Definition Syntax (-Xlabel-colon...)*, p.285).

- When an **asm** string statement appears in global scope, the compiler adds it to the output assembly module *after* all of the function definitions. For this reason, global **asm** string statements should not use assembler directives—such as **.set** *symbol*—on which other **asm** statements (appearing in functions) depend.

Example 7-1 **Disable Interrupts**

The following sequence of **asm** string statements disables hardware interrupts. Note that a scratch register is used in the example.

```
asm(" mrs   r0,cpsr     ; read the cpsr");
asm(" orr   r0,r0,#0x80 ; set the interrupt disable bit");
asm(" msr   cpsr_c,r0   ; update the control bits in the cpsr");
```

7.4 Reordering in asm Code

If optimization is requested (options **-O** or **-XO**), after generating an assembly file, the driver will run the **reorder** optimization program. **reorder** runs peephole optimizations and schedules the assembly file before the assembler assembles it, and does not distinguish assembly code generated by the compiler from assembly code inserted by **asm** macros or **asm** strings. Thus, explicit assembly instructions written in a particular order by the user may still be reordered by **reorder**.

In general this may improve even hand-coded assembly language. If it is necessary to prevent this, write a **.set noreorder** directive in the **asm** string or **asm** macro at the point at which such re-ordering should be disabled, and a **.set reorder** directive where re-ordering can be re-enabled. Alternatively, define the string or macro as **volatile**.

7.5 Direct Functions

Direct functions, available in C modules only, provide a way to inline machine code in a function. In a direct function definition, the body of the function is a list of integer constant expressions which represent the machine code. The form is:

```
[return_type] function_name ( [ parameter_type parameter_name ,... ] ) =
{
    integer-constant-expression , ...,
    integer-constant-expression , ...,
    ...
};                                  /* ';' required */
```

Rules:

- A direct function is signaled by the presence of an "**=**" character between the parameter list and the body of the function.
- The expressions in the body are separated by commas and may be written one or more per line (with a comma after the final expression on a line if additional expression lines follow).
- The final "**}**" closing the function body must be followed by a "**;**".

A direct function is always inlined when called. When called, what would be the branch to the function is replaced by a **.long** assembler directive having as operands the value of each expression as a hex constant. Otherwise, normal calling conventions are followed (e.g., any parameters are set up in the usual manner).

Direct functions are supported primarily for compatibility reasons. **asm macros** provide a more flexible method to do nearly the same thing. See Table 7-1 which contrasts the differences.

8 Internal Data Representation

This chapter describes the alignments, sizes, and ranges of the C and C++ data types for ARM microprocessors.

8.1 Basic Data Types

By default, the type plain **char**—that is, **char** without the keyword **signed** or **unsigned**—is treated as unsigned.

The following table describes the basic C and C++ data types available in the compiler. All sizes and alignments are given in bytes. An alignment of 2, for example, means that data of this type must be allocated on an address divisible by 2.

Table 8-1 **C/C++ Data Types, Sizes, and Alignments**

Data Type	Bytes	Align	Notes
char	1	1	range (0, 255), or (-128, 127) with **-Xchar-signed** (Note 1)
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 (Note 2)	4	4	same as **int**
	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

Table 8-1 **C/C++ Data Types, Sizes, and Alignments** (cont'd)

Data Type	Bytes	Align	Notes
ptr-to-member-fn	12	4	C++: pointer to member function

Notes:

1. If the option **-Xchar-unsigned** is given, the plain char type is **unsigned**. If the option **-Xchar-signed** is given, the plain char type is **signed**.
2. If the option **-Xenum-is-int** is given, enumerations take four bytes. This is the default for C.

   If the option **-Xenum-is-small** is given, the smallest **signed** integer type permitted by the range of values for the enumeration is used, that is, the first of **signed char**, **short**, **int**, or **long** sufficient to represent the values of the enumeration constants. Thus, an enumeration with values from 1 through 128 will have base type **short** and require two bytes.

   If the option **-Xenum-is-best** is given, the smallest **signed** or **unsigned** integer type permitted by the range of values for an enumeration is used, that is, the first of **signed char**, **unsigned char**, **short**, **unsigned short**, **int**, **unsigned int**, **long**, or **unsigned long** sufficient to represent the values of the enumeration constants. Thus, an enumeration with values from 1 through 128 will have base type **unsigned char** and require one byte. This is the default for C++.
3. A reference is implemented as a pointer to the variable to which it is initialized.

8.2 Byte Ordering

Unless the "**L**" object format is used, all data is 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** in *pack Pragma*, p.129 and *__packed__ and packed Keywords*, p.137.

8.3 Arrays

Arrays, excluding character arrays, have the same alignment as their element type. The size of an array is equal to the size of the data type multiplied by the number of elements. Character arrays have a default alignment of 4. **-Xsize-opt sets the** alignment of character arrays to 1, and **-Xstring-align** overrides **-Xsize-opt**. **-Xarray-align-min**, which overrides **-Xstring-align**, specifies a minimum alignment for all arrays.

8.4 Bit-fields

Bit-fields can be of type **char**, **short**, **int**, **long**, or **enum**. Plain bit-fields are unsigned by default. By using the **-Xbit-fields-signed** option (C only) or by using the **signed** keyword, bit-fields become signed. The following rules apply to bit-fields:

- Allocation is from most significant bit to least.
- 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.
- Bit-fields are allocated as closely as possible to the previous **struct** member without crossing a type boundary.
- A zero-length bit-field pads the structure to the next boundary specified by its type.
- Bit-fields may not be type **long long**.
- The compiler accesses a bit-field by loads and stores appropriate to the bit-field's type. For example, an **int** bit-field is accessed using a **word** load or store (or an equivalent set of smaller load/stores in the unaligned case), even if the bit-field spans only one byte. To ensure that a bit-field is accessed using byte (or half-word) load/stores, make the bit-field **char** or **short**, or use the **-Xcompress-bitfields** option.
- When a bit-field is promoted to a larger integral type, the comiler preserves sign as well as value unless **-Xstrict-bitfield-promotions**, **-Xdialect-strict-ansi**, or **-Xstrict-ansi** is enabled.

8.5 Classes, Structures, and Unions

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 *pack Pragma*, p.129 and *__packed__ and packed Keywords*, p.137 for more information.

8.6 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 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** is 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

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:

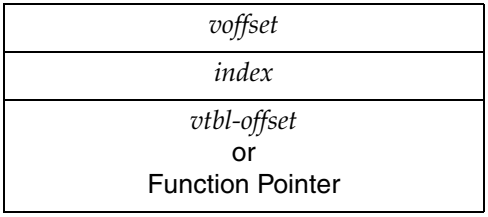
Base part of **D**
Body of **E**: . . .
V1
V2
magic for **V3**
V3

- 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:



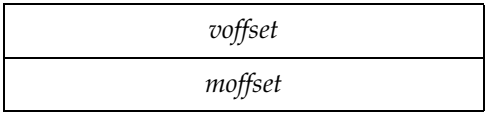
The *voffset* field is an integer that is used when the virtual function table is located in a virtual base class. In this case it contains the offset to the virtual base class pointer + 1. Otherwise it has a value of 0.

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:



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.

Virtual Function Table Generation—Key Functions

The virtual function table for a class will be generated only in the module which *defines* (not declares) its *key* virtual function (and does not inline it). The *key* virtual function is the virtual function declared lexically first in the class (or the only virtual function in the class if there is only one).

Consider, for example:

```
class C {
    public:
        virtual void f1(...);
        virtual void f2(...);
}
```

Because **f1** is the first virtual function declared in the class, it is the key virtual function.

Then, the virtual function table will be emitted for the module which provides the non-inlined definition of **f1**.

8.7 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 "none" above.

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	In a register or on the stack (Note 2).

Specifier	Linkage	Allocation
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 cannot 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. Allocation of static variables is as per .
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 on 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.

9 Calling Conventions

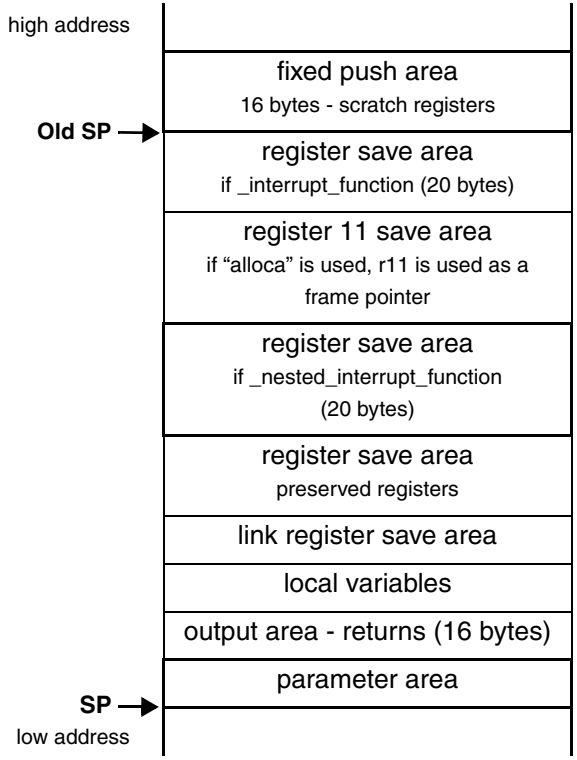
9.1 Introduction

This chapter describes the interface between a function caller and the called function. Stack layout, argument passing, returning results, and register use are all described in detail.

9.2 Stack Layout

The following figure shows the stack frame after completion of the prolog in the

then "current" function (**SP** = stack pointer, **r13**).



Notes:

- To facilitate certain optimizations, the stack pointer is aligned on an 8-byte boundary by inserting a 4-byte alignment gap if necessary.
- Leaf routines that do not use any stack space do not create a stack area at all.

9.3 Argument Passing

Values of size one and two bytes are extended to four bytes. Following the expansion, the first sixteen bytes of arguments are passed in registers **r0**, **r1**, **r2**, and **r3** (with any eight-byte values in either **r0-r1** or **r2-r3**). Additional arguments are passed on the stack. Arguments with an alignment of 8 are aligned on 8, all other arguments are aligned on 4. For a function with a variable argument list, sixteen bytes are reserved on the stack for a copy of the arguments passed in the registers. The stack is always aligned on 8.

The stack layout is shown above. Examples of argument passing are shown below.

Table 9-1 **Examples of ARM Argument Passing**

Example Function Call	r0	r1	r2	r3	Stack
`f(char a, char b, char c, char d, char e)`	a	b	c	d	e
`f(char a, int b, double c, double d)`	a	b	— c —		d
`f(char a, double b, char c)`	a	-	— b —		c
`f(double a, char c);`	— a —		c		

9.4 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 *13.5 C++ Name Mangling*, p.225.
- An argument of **class**, **struct**, or **union** type may, depending on the target architecture and the size of the actual parameter, be passed as a pointer to the object. (But this does not happen if the function is declared with **extern "c"**.) For this reason, when a C++ function with **class**, **struct**, or **union** parameters is called from a C module, it should always be assumed that the C++ compiler

expects a pointer argument. For example, suppose the following function is defined in a C++ module:

```
int ff(struct S s);
```

To call this function from a C module, use code like this:

```
struct S xyz;
int i = ffmangledname(&xyz);
```

where *ffmangledname* is the mangled form of **ff**. To find the mangled name of a C++ function, see *13.5 C++ Name Mangling*, p.225 and *29. D-DUMP File Dumper*.

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.

9.5 Returning Results

Characters and shorts are extended to 32-bits and returned in register **r0**. Integers, pointers, and **float** values are returned in register **r0**. **Double** and **long long** values are returned in **r0**/**r1**. Structures, unions, and classes with total size and alignment of 1 are returned in **r0**. All other types are returned in the memory area pointed to by a hidden address argument passed in register **r0**.

Class, Struct, and Union Return Types

With the exceptions mentioned above, a function with a return type of **class**, **struct**, or **union** is called with a hidden argument of type *pointer to function return type*. The called function copies the return argument to the object pointed at by the hidden argument; the ordinary arguments are “bumped” one place to the right.

9.6 Register Use

The following describes how registers are used by the compiler.

r0 - r3
Scratch registers; not preserved by functions. Hold variables whenever possible. Also used to pass parameters. **r0 - r1** used to return results.

r4 - r11
Preserved registers; saved when used by functions.

r12
Scratch register; not preserved by functions.

r13
Stack pointer.

r14
Link register; contains the return address.

r15
Program counter.

10 Optimization

Optimizations have two purposes: to improve execution speed and to reduce the size of the compiled program.

Most optimizations are activated by the **-O** option (*5.3.17 Optimize Code (-O)*, p.40). A few, such as inlining, are activated by the **-XO** option (*5.4.95 Enable Extra Optimizations (-XO)*, p.96). See also the discussion of optimization and debugging under the **-g** option (*5.3.9 Generate Symbolic Debugger Information (-g)*, p.37).

10.1 Optimization Hints

The compilers attempt to produce code as compact and efficient as possible. However, some information about characteristics of the program only the user has. This section describes various ways the user can enable the compiler to 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, several 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 *5.4.72 Inline Functions with Fewer Than n Nodes (-Xinline=n)*, p.84, for more details and *5.4.142 Control Loop Unrolling (-Xunroll=n, -Xunroll-size=n)*, p.112**,** for a definition of internal nodes. Other options that control inlining include **-Xexplicit-inline-factor** (*5.4.49 Control Inlining Expansion (-Xexplicit-inline-factor)*, p.75) and **-Xinline-explicit-force** (*5.4.73 Allow Inlining of Recursive Function Calls (-Xinline-explicit-force)*, p.85).

- **-Xunroll-size=***n*

  Controls the maximum size of a loop body to be unrolled. See also *5.4.142 Control Loop Unrolling (-Xunroll=n, -Xunroll-size=n)*, p.112, for more details.

There is also a 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 interprocedural optimizations (optimizations across functions) the compiler keeps internal structures of every function in main memory. This can slow compilation 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 *5.4.99 Specify Optimization Buffer Size (-Xparse-size)*, p.97.)

With all the different optimization options, it is sometimes difficult to decide which options will produce the best result. The **-Xblock-count** and **-Xfeedback** options (*5.4.11 Insert Profiling Code (-Xblock-count)*, p.60, *5.4.52 Optimize Using Profile Data (-Xfeedback=file)*, p.76), which produce and use profiling information, provide

powerful mechanisms to help with this. With profiling information available, the compiler can make most optimization decisions by itself.

The following guidelines summarize which optimizations to use in varying situations. The options used are found in *5. Invoking the Compiler*.

- 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, use the profiling features referred to above.
- 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 the compiler can use to retain functions. This allows the compiler to perform more interprocedural optimizations. Use the following option to increase the available memory to 8,000 KByte:

```
-Xparse-size=8000
```

- 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:

```
DFLAGS="-XO -Xparse-size=8000 -Xinline=50"          (UNIX)
export DFLAGS
set DFLAGS=-XO -Xparse-size=8000 -Xinline=50        (Windows)
```

- If possible, disable exceptions and run-time type information (**-Xexceptions-off**, **-Xrtti-off**). This can reduce code size significantly.

What to Do With Programs

The following list describes coding techniques which will help the compiler produce optimized code.

- Use local variables. The compiler can keep these variables in registers for longer periods than global and static variables, since it can trace all possible uses of local variables.
- 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.
- In a structure, put larger members first. This minimizes padding between members, saving space, and ensures optimal alignment, saving both space and time. For example, change:

```
struct \_pack {
    char    flag;
    int     number;
    char    version;
    int     op;
}
```

to

```
struct good_pack {
    int     number;
    int     op;
    char    flag;
    char    version;
}
```

- For target architectures which include a cache, declare variables which are frequently used together, near each other to reduce cache misses. For example, change:

```
struct bad {
    int             type;
    ...
    struct  bad     *next;
};
```

to

```
struct good {
    int             type;
    struct  good    *next;
    ...
};
```

Then both **type** and **next** will likely be in the cache together in constructs such as:

```
while (p->type != 0) {
    p = p->next;
}
```

- Use the **const** keyword to help the optimizer find common sub-expressions. For example, ***p** can be kept in a register in the following:

```
void func(const int *p) {
    f1(*p);
    f2(*p);
}
```

- Use the **static** keyword on functions and module-level variables that are not used by any other file. Optimization can be much more effective if it is known that no other module is using a function or variable. Example:

```
static int si;

void func(int *p) {
    int i;
    int j;

    i = si;
    *p = 0;
    j = si;
    ...
}
```

The compiler knows that ***p = 0** does not modify variable **si** and so can order the assignments optimally.

- Use the **volatile** keyword only when necessary because it disables many optimizations.
- Avoid taking the address of variables. When the address of a variable is taken, the compiler usually assumes that the variable is modified whenever a function is called or a value is stored through a pointer. Also, such variables cannot be assigned to registers. Use function return values instead of passing addresses.

  Example: change

```
int func (int var) {
    far_away1(&var);
    far_away2(var);
    return var;
}
```

to

```
int func (int var) {
    var = new_far_away1(var);
    far_away2(var);
    return var;
}
```

- Use the **#pragma inline** directive and the **inline** keyword for small, frequently used functions. **inline** eliminates call overhead for small functions and increases scheduling opportunities.
- Use the **#pragma no_alias** directive to inform the compiler about aliases in time critical loops. Example:

```
void add(double d[100][100], double s1[100], double s2[100])
#pragma no_alias *d, *s1, *s2
{
    int i;
    int j;

    for (i = 0; i < 100; i++) {
    for (j = 0; j < 100; j ++) {
    d[i][j] += s1[i] * s2[i];
    }
    }
}
```

  Because it is known that there is no overlap between **d** and each of **s1** and **s2**, the expression **s1[i]*s2[i]** can be moved outside of the innermost loop.

- Use **#pragma no_side_effects** and **#pragma no_return** on appropriate functions. Example:

```
comm.h:
    #pragma no_side_effects busy_wait(1)
    #pragma no_return comm_err

file.c:
    #include "comm.h"
         a = *p;
        busy_wait(&sem);
        if (error) {
        ...
        comm_err("fatal error");
    }
    b = *p;
```

  Because **busy_wait** is known to have no side effects and **comm_err** is known not to return, the compiler can assign ***p** to a register.

- Use **asm** macros rather than separate assembly functions because it eliminates call overhead. See *7. Embedding Assembly Code*.

- Avoid **setjmp( )** and **longjmp( )**. When the compiler finds **setjmp( )** in a function, a number of optimizations are turned off. For example, when the **-Xdialect-pcc** 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** on 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.

- If possible, eliminate C++ exception-handling code (**try**, **catch**, or **throw**). This allows you to compile with exceptions disabled (**-Xexceptions-off**), which reduces stack space and increases execution speed.

10.2 Cross-Module Optimization

Cross-module optimization, controlled with the **-Xcmo-...** options (see *5.4.23 Enable Cross-module Optimization (-Xcmo-...)*, p.64), allows the compiler to optimize calls between functions in different source files. This feature can improve execution efficiency but requires the developer to track intermodule dependencies with care.

Currently, function inlining is the only implemented cross-module optimization.

The compiler implements cross-module optimization by constructing a database of information about functions and variables. To use cross-module optimization, compile your project twice—first with **-Xcmo-gen** to create a database, then with **-Xcmo-use** to optimize using information from the database. You must specify a name and location for the database file. Examples:

```
dcc -Xcmo-gen=C:\projects\MyProject\MyProject.db main.c        (Windows)
dcc -Xcmo-use=C:\projects\MyProject\MyProject.db main.c
dcc -Xcmo-gen=/projects/MyProject/MyProject.db main.c          (UNIX)
dcc -Xcmo-use=/projects/MyProject/MyProject.db main.c
```

The **-Xcmo-gen** compiler pass is used only for building the database. All object files created by this pass should be regenerated during the next build.

NOTE: Do not use the **-Xcmo-...** options to compile a project that contains two or more source files (in different directories) with the same base name.

If there are functions that you do *not* want to have inlined across modules, you can specify them by adding **-Xcmo-exclude-inline** to the command line with **-Xcmo-use**. For example:

```
dcc -Xcmo-use=...\MyProject.db -Xcmo-exclude-inline=f1,f2 main.c
```

tells the compiler not to inline **f1** or **f2** across modules. Names of C++ functions must be given in mangled form (see *13.5 C++ Name Mangling*, p.225); to find the mangled form of a function name, use the **ddump** utility (see *29. D-DUMP File Dumper*).

-Xcmo-verbose, combined with **-Xcmo-use** or **-Xcmo-gen**, outputs a list of inlined (or inlinable) functions.

Before using cross-module optimization, please read the following additional notes.

Database Location and Use

The database name should be specified with a full directory path. Otherwise, the compiler uses the current working directory, which could result in fragmented databases residing in multiple locations.

It is preferable to use a non-network directory for the database. Never share a database among compiler installations, even when building from the same source files.

Use With Other Optimizations and Build Options

The **-Xcmo-...** switches are affected by other build options. In general, you should turn compiler optimizations *off* when building with **-Xcmo-gen** and *on* when building with **-Xcmo-use**. More specifically:

- To save time, disable optimizations and skip the linking step when building with **-Xcmo-gen**. (Executable output from the **-Xcmo-gen** compilation is ultimately discarded.)
- **-Xcmo-use** is ignored unless other optimizations are enabled (**-O** or **-XO**).
- Optimization-related compiler switches, including **-Xinline**, apply to cross-module optimization as well. If **-Xinline** is set to a very low value, cross-module optimization is unlikely to be useful. (**-Xinline** has no effect on the construction of the database itself.)
- If **-Xinline** is set to a high value, cross-module optimization can result in large executables and long compilation time. You may want to compile specific source files with cross-module optimization disabled.

Database Maintenance

Every time you compile with **-Xcmo-use**, the compiler updates the existing database by adding to the list of functions that are candidates for inlining—*but it does not perform dependency analysis*. Hence the database can easily become unsynchronized after repeated incremental builds. (This occurs, for example, when a source file containing a called function has changed, but the source file containing the calling function is unchanged.) It is important to track dependencies and recompile periodically with **-Xcmo-gen**. When in doubt, *manually delete* the database file before recompiling.

After moving or copying files, always delete the database file and regenerate it with **-Xcmo-gen**.

Special Name Mangling

To enable cross-module optimization, the compiler assigns a unique mangled name to each function and static variable. Mangled function names begin with **__STF** followed by a line number, function name, mangled filename, and other information. Mangled variable names begin with **__STV** followed by a line number, variable name, mangled filename, and other information. The demangling utility does not demangle these names.

10.3 Target-Independent Optimizations

The following optimizations are performed by the compiler on all targets.

The numbers in parentheses after the name of each optimization are mask bits for the **-Xkill-opt** option. Optimizations can be selectively disabled by specifying **-Xkill-opt=***mask*, where *mask* can be given in hex (e.g. **-Xkill-opt=0x12**). Multiple optimizations can be disabled by OR-ing their bits; undefined mask bits are ignored. **-Xkill-opt=0xffffffff** has the same effect as not using the **-O** option at all.

NOTE: Regardless of which options are specified, there is no way (short of disabling optimizations completely) to guarantee that the compiler will or will not perform a specific optimization on a given piece of code.

-Xkill-opt is deprecated and should be used only on the advice of Customer Support.

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 functions with fewer than the number of nodes set by **-Xinline** with the actual code from the same functions to avoid call-overhead and generate more opportunities for further optimizations. See *5.4.142 Control Loop Unrolling (-Xunroll=n, -Xunroll-size=n)*, p.112, for the definition of *node*; assembly files saved with **-S** show the number of nodes for each function.

To be inlined, the called function must be in the same file as the calling function.

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 or the **inline** keyword if enabled by **-Xkeywords=4**. Functions inlined by the use of keywords are local (**static**) by default, but can be made public with **extern**. See *__inline__ and inline Keywords*, **p.135**.
2. Use the **#pragma inline** *function-name* directive. The **#pragma** directive can be used in C++ code to avoid the local **static** linkage forced by the **__inline__** or **inline** keywords. See *inline Pragma*, p.125.
3. Use option **-XO** to automatically inline functions of up to the number of nodes set by **-Xinline** (see *5.4.72 Inline Functions with Fewer Than n Nodes (-Xinline=n)*, p.84). Option **-XO** sets this value to 40 nodes by default.

In addition to **-Xinline**, the options -**Xexplicit-inline-factor**, **-Xinline-explicit-force**, and **-Xcmo-...** also control inlining of functions.

NOTE: Code must be optimized by use of the **-XO** or **-O** option for inlining to occur.

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:

```
extern int x;

int check(int *x)
{
    if ( *x > 569) {
        return(999);
    } else {
        return(100);
    }
}
```

```
int foo(int y)
{
    int i, j;         // can be placed in registers

    i = x * y;
    j = check(&i);
    if (j > i) {
        i = check(&j);
    } else {
        i = 365;
    }
    return j*i;
}
```

Structure Members to Registers (0x10)

This optimization places members of local structures and unions in registers whenever it is possible. It also optimizes assignments to structure and union members. Example:

```
int fpp(int);
int bar(int, int);
struct x{
    int a;
    int b;
};
void goo();

foo()
{
    struct x X;

    X.a = fpp(3);
    X.b = fpp(5);

    if (bar(X.a, X.b)) {
        goo();
    }
}
```

If the optimization is enabled, the compiler attempts place **X.a** and **X.b** in registers rather than allocating memory for **X**.

Assignment Optimization (0x80)

Multiple increments of the same variable are merged:

```
p++;                    ->
p[0] = 0;                       p[1] = 0;
p++;                            p[2] = 1;
p[1] = 1;                       p += 2;
```

Pre- and post-increment/decrement addressing modes are used *when available on the target processor*:

```
p++;                    ->
p[0] = 0;                       *++p = 0;
p++;
p[1] = 1;                       *++p = 1;
```

Increments are moved from the end of a loop to the beginning in order to use incrementing addressing modes *when available on the target processor*:

```
while(*s++) ;           ->      s--; while(*++s) ;
```

Tail Call Optimization (0x100)

In the following case, the call to **printf** is converted to a branch to **printf** and the stack frame is undone before the branch.

```
int _myfunc(char *fmt, int val)
{
    return printf(fmt,val);
}
```

This optimization is performed even if no **-O** or **-XO** switch is used.

NOTE: In earlier releases (prior to version 4.3), the 0x100 mask was used to disable simple branch optimization.

Common Tail Optimization (0x200)

Different paths with equal tails are rewritten. This optimization is most effective when many **case** statements end the same way:

```
void bar(), foo(), gfoo(), hfoo();
```

```
lucky()
{
    switch (a) {
    case 1:
        foo(); bar();
        break;
    case 2:
        gfoo(); bar();
        break;
    case 3:
        hfoo(); bar();
        break;
    case 4:
        foo(); bar();
        break;
    default:
        bar();
        break;
    }
}
```


The call to **bar( )** is removed from the individual **case** statements and executed separately at the end of the **switch** statement.

This optimization cannot be disabled unless **reorder** is disabled. To disable **reorder**, use **-W1** with no argument (see *5.3.30 Substitute Program or File for Default (-W xfile)*, p.44).

Variable Live Range 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 code that falls through to conditional branches where the outcome can be computed are rewritten. This typically occurs after a loop with multiple exits.

```
extern int x;
extern int bar(int x);

int foo(int a, int b)
{
    int i, y, z = 0;

    x = bar(a);
    if (x > 44)
    {
        y = a + b;
        if (x < 22) {    // always false when evaluated
            z = a * 365; // never executed
        }
    }
    return (x + y + z);
}
```

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 +=var[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++;
}                           }
```

Also, empty loops are removed.

Loop Unrolling (0x8000)

Small loops are unrolled to reduce the loop overhead and increase opportunities for rescheduling. Option **-Xunroll** option sets the number of times the loop should be unrolled. Option **-Xunroll-size** defines the maximum size of loops allowed to be unrolled (see *5.4.142 Control Loop Unrolling (-Xunroll=n, -Xunroll-size=n)*, p.112 for both options).

Note: some sufficiently small loops may be unrolled more than *n* times if total code size and speed is better. 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.

```
int bar(int);

int foo()
{
    int x, a, b, y;

    x = 365 * (a + b);
    y = bar(x);
    return y;
}
```

No memory is allocated for **a** or **b**. The operation **a + b** is not performed.

Unused assignment deletion (0x40000)

Assignments to variables that are not used are removed.

```
int foo(int x, int y)
{
    int a, b;

    a = x + 365;    // removed
    b = x - y;
    return b;
}
```

This optimization cannot be disabled unless **reorder** is disabled. To disable **reorder**, use **-W1** with no argument (see *5.3.30 Substitute Program or File for Default (-W xfile)*, p.44).

Minor Transformations to Simplify Code Generation (0x80000)

Some minor transformations are performed to ease recognition in the code generator:

```
if (a) return 1;        ->      return a ? 1 : 0;
return 0;
```

Register Coloring (0x200000)

This optimization locates variables that can share a register.

```
extern int a[100], b[100];

foo()
{
    int i, a, j, b;

    for (i = 0; i < 10; i++) {
        a += bar(i) + i;
    }

    for (j = 0; j < 80; j-=6) {
        b += bar(i) - i;
    }
}
```

a and **j** use the same register.

Interprocedural Optimizations (0x400000)

Registers are allocated across functions. Inlining and argument address optimizations are performed.

```
static int foo(int a, int b)
{
    return ((a > b)? a: b);
}

bar(int i, int j)
{
    printf("larger value = %d\n", foo(i,j));
}
```

The **foo** function is inlined into **bar**.

Remove Entry and Exit Code (0x800000)

The prolog and epilog code at the beginning and end of a function which sets up the stack-frame is not generated whenever possible.

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).

NOTE: When this optimization is disabled, the compiler may still use registers to store variables. To control register use, use **#pragma global_register** (*global_register Pragma*, p.124).

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.

```
int sum;
int c[10];
int bar(int);
foo(int a, int b)
{
    int i;

    for(i = 0; i < 10; i++) {
        sum += a * b;
        c[i] = bar(i);
    }
}
```

The operation **a*b** is performed outside of the loop statement.

Live-Variable Analysis (0x40000000)

Live variable analysis is done for global and static variables. This means that global and static variables can be allocated into registers and any stores into them can be postponed until the last store in a live range.

Local Data Area Optimization (0x80000000)

This optimization creates a Local Data Area (LDA) into which variables may be placed for fast, efficient base-offset addressing. See *14.4 Local Data Area (-Xlocal-data-area)*, p.247 for details.

This optimization can be disabled by setting **-Xlocal-data-area=0** or restricted to static variables by setting **-Xlocal-data-area-static-only**.

Feedback Optimization

By utilizing 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 (***expr***) return;**
- Branch prediction is performed.

The **-Xblock-count** and **-Xfeedback** options are available to collect and use profiling data. See *15.12 Profiling in An Embedded Environment*, p.272.

10.4 Target-Dependent Optimizations

The following target-dependent optimizations are specific to the ARM family and are done by the **reorder** program.

The numbers in parentheses after the name of each optimization are mask bits for the **-Xkill-reorder o**ption. Optimizations can be selectively disabled by specifying **-Xkill-reorder=***mask*, where *mask* can be given in hex (e.g. **-Xkill-reorder=0x9**). Multiple optimizations can be disabled by OR-ing their bits; undefined mask bits are ignored.

NOTE: Regardless of which options are specified, there is no way (short of disabling optimizations completely) to guarantee that the compiler will or will not perform a specific optimization on a given piece of code.

-Xkill-reorder is deprecated and should be used only on the advice of Customer Support.

NOTE: The **reorder** program, which does target-dependent optimization, parses the assembler output of the compiler. Because this output is assumed to be correct, **reorder** may abort on assembly code errors, including errors in hand-written **asm** macros and strings. If an error in **reorder** appears to be persistent, confirm that any handwritten assembly code is correct, perhaps by removing it temporarily, before reporting the difficulty to Customer Support.

General Peephole Optimization (0x8)

Peephole optimization makes final improvements within basic blocks, especially to remove inefficiencies caused by interactions among other optimizations which would be uneconomical to detect otherwise. Examples:

- A branch to a single instruction followed by another branch is rewritten by inlining the instruction at the current address.
- Certain instructions which do not change any register are removed.
- Elimination of redundant load and stores.
- Register coalescing to eliminate moves.

Make Conditional (0x9)

Allows for the use of conditional instructions eliminating the necessity of forward branches.

Simple Scheduling Optimization (0x1000)

Attempt to optimize load instructions.

10.5 Example of Optimizations

The following C program demonstrates several of the optimizations available in the compiler and how they interact with each other.

The numbers in parentheses are used to identify the optimizations in the generated code for the example, shown following the table.

The target processor is the ARM. The optimizations shown are:

(1) use scratch registers for variables

(2) unused assignment deletion

(3) complex branch optimization

(4) peephole optimization

(5) loop strength reduction

(6) loop count-down optimization

(7) global common subexpression elimination

(8) inlining of functions

(9) constant and variable propagation

(10) make conditional

(11) basic reordering optimization

bubble.c implements sorting of 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 following line,

```
dcc -tARMEN -S -Xpass-source -XO bubble.c
```

the file **bubble.s** is generated as shown below (option **-Xpass-source** conveniently causes the source to be included intermixed as comments with the generated assembly code in **bubble.s**).

Only the **bubble( )** function is shown; code will also be present for the **swap( )** function in **bubble.s** because it is not **static** and may therefore be called from another module. Comments have been added below to explain the optimizations performed.

Table 10-1 **Illustration of Optimizations for ARM**

C Code	Generated Assembly Code	Explanation
	`.align 1` `.export bubble`	
	`bubble:`	Start of function **bubble**.

	`stmfd r13!,{r7,r8,`
	`r9,r10,r11,`
	`lr}`	

{		
int change = 1; **int i;**		The assignment **change = 1** is eliminated (2) since it is used only in the first **while** test, which is known to be true and removed (3).
int test_count = 0;	`ldr r7,=0`	**r7 test_count = 0;**
	`.L4:`	Top of **while (change)** loop.
while (change) {		**change** was just initialized to 1 and cannot initially be 0, so the loop test can be made only at the bottom.
change = 0; **count--;**	`ldr r2,=0` `sub r1,r1,#1`	**r2 change = 0;** **r1 count--;**
for (i = 0; **i < count;** **i++) {**	`cmp r1,#0` `ble .L16`	Before entering loop, if **r1 count** is <= 0, branch to the **return** because just set **change** to 0 so further passes through the outer loop would leave **change** unchanged.
	`cmp r1,#0` `ble .L16`	Before entering loop, if **r1 count** is <= 0, branch to the **return** because just set **change** to 0 so further passes through the outer loop would leave **change** unchanged.

Table 10-1 **Illustration of Optimizations for ARM** (cont'd)

C Code	Generated Assembly Code			Explanation
		mov	r11,r0p	Loop strength reduction (5) has replaced all references to **base[i]** with a created pointer, **$$2**, initialized to **r0 base** and placed in **r11**. Since no more references are made to **count**, loop count-down optimization (6) will decrement **i** from **count** to 0 instead of incrementing **i** and comparing it against **count**.
		mov	r8,r1	**r8 i** is set to **r1 count** per the above.
	.L8:			Top label of **for** loop.
test_count++;	.L7:			
if (base[i] > base[i+1] {		ldr ldr	r9,[r11,#0] r10,[r11,#4]	Load **base[i]** equivalent **$$2[0]** to **r9 ($$4)**, and load **base[i+1]** equivalent **$$2[1]** to **r10 ($$3)** (7).
swap2 ...		cmp strgt strgt	r9,r10 r10,[r11,#0] r9,[r11,#4]	If **r9** > **r10**, the function **swap2** is inlined (8) by utilizing make conditional (10) which allows for conditional instructions without branching.
}				End of **if**.
}		subs add	r8,r8,#1 r11,r11,#4 .L8	Decrement **r8 i** (6). Increment **base** pointer **$$2++** (6). Bottom test of **for**: test **r8 i** against 0 (4). If **i** is not zero, branch to the top of the **for** loop.
}		cmp bne	r2,#0 .L4	Bottom test of **while (change)**. If **change** is not zero, branch to the top of the loop.
	.L16:			
return test_count;		mov	r0,r7	Move **test_count** to return register.
}		ldmfd mov	r13!,{r7,r8, r9,r10,r11, pc} pc,lr	
	# Allocations for bubble			Variable allocations are commented for debugging.

Table 10-1 **Illustration of Optimizations for ARM** (cont'd)

C Code	Generated Assembly Code			Explanation
	#	r0	base	Arguments are kept in their original registers (2).
	#	r1	count	
	#	r2	change	Variables are put in scratch registers.
	#	r8	i	
	#	r7	test_count	
	#	r11	$$2	Loop strength reduction (6) variable for **base** pointer.
	#	r10	$$3	Global common subexpression elimination (7) for **base[i+1]** and **base[i]**.
		r9	$$4	
	#	not allocated	tmp	Variables deleted by Variable propagation (9).
	#	not allocated	ip	

11 The Lint Facility

11.1 Introduction

The lint facility is a powerful tool to find common C programming mistakes at compile time. (For C++, see **-Xsyntax-warning-on** on *5.4.137 Disable Certain Syntax Warnings (-Xsyntax-warning-...)*, p.110.) 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 after a small number of warnings, use the **-Xstop-on-warning** option to 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 *5.4.81 Generate Warnings On Suspicious/Non-portable Code (-Xlint=mask), p.88* for details.
- **-Xlint** can be used 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.** There are three types of errors marked by different comment forms:

- Comments containing the form "(0x*XX*)" are on lines with suspicious constructs detected by **-Xlint**; the hex value is the **-Xlint** bit mask which disables the test.
- Comments of the form */* warning: ... */* and */* error: ... */* are used on lines for which the compiler reports a warning or error with or without **-Xlint**.
- Two lines are a result of option **-Xforce-prototypes** as noted.

Actual warnings from the compiler follow the code. Note that warnings are not necessarily in line number order because the compiler detects the errors during different internal passes.

11.2 Examples

Example 11-1 **Program for -Xlint Demonstration**

```
1:    void f1(int);
2:    void f2();
3:                                  /* (-Xlint mask bit disables)     */
4:    static int f4(int i)          /* function never used           (0x10) */
5:    {
6:     if (i == 0)
7:            return;               /* missing return expression      (0x20) */
8:        return i+4;
9:    }
10:
11:    static int f5(int i);        /* error: function not found               */
12:
13:    static int i1;               /* variable never used             (0x10) */
14:
15:    int m(char j, int z1)        /* parameter never used            (0x10) */
16:    {
17:        int i, int4;
18:        char c1;
19:        unsigned u = 1;          /* variable set but not used       (0x40) */
20:        int z2;                  /* variable never used             (0x10) */
21:
22:        c1 = int4;               /* narrowing type conversion      (0x100) */
23:
24:        if (j) {
25:            u = 4294967295;
```

```
26:             i = 0;
27:         } else {
28:             u = 4294967296;      /* warning: constant out of range          */
29:         }
30:         f1(i);                   /* variable might be used
31:                                     before being set                (0x02) */
32:         switch(i) {
33:                 j = 2;           /* statement not reached           (0x80) */
34:                 break;
35:
36:             case 0:              /* -X force prototype, not lint, warns:    */
37:                 f2(i);           /*    function has no prototype            */
38:                 f3(i);           /*    function not declared                */
39:                 f5(i);
40:                 break;
```

Example 11-2 **-Xlint example output**

```
"lint.c", line 7: warning (dcc:1521): missing return expression
"lint.c", line 22: warning (dcc:1643): narrowing or signed-to-unsigned type
                                       conversion found: int to unsigned char
"lint.c", line 28: warning (dcc:1243): constant out of range
"lint.c", line 37: warning (dcc:1500): function f2 has no prototype
"lint.c", line 38: warning (dcc:1500): function f3 has no prototype
"lint.c", line 42: warning (dcc:1583): overflow in constant expression
"lint.c", line 48: warning (dcc:1643): narrowing or signed-to-unsigned type

   conversion found: short to unsigned char
"lint.c", line 48: warning (dcc:1244): constant out of range (=)
"lint.c", line 47: warning (dcc:1251): label deflaut not used
"lint.c", line 15: warning (dcc:1516): parameter z1 is never used
"lint.c", line 20: warning (dcc:1518): variable z2 is never used
"lint.c", line 33: warning (dcc:1522): statement not reached
"lint.c", line 50: warning (dcc:1522): statement not reached
"lint.c", line 62: warning (dcc:1521): missing return expression
"lint.c", line 19: warning (dcc:1604): Useless assignment to variable u.
                                       Assigned value not used.
"lint.c", line 22: warning (dcc:1604): Useless assignment to variable c1.
                                       Assigned value not used.
"lint.c", line 43: warning (dcc:1604): Useless assignment to variable j.
                                       Assigned value not used.

--------------------------------------------------------------------------------

"lint.c", line 22: warning (dcc:1608): variable int4 might be used before set
"lint.c", line 30: warning (dcc:1608): variable i might be used before set
"lint.c", line 54: warning (dcc:1606): condition is always true/false
"lint.c", line 58: warning (dcc:1606): condition is always true/false
"lint.c", line 4: warning (dcc:1517):  function f4 is never used
"lint.c", line 11: error (dcc:1378):   function f5 is not found
"lint.c", line 13: warning (dcc:1518): variable i1 is never used
```

12 Converting Existing Code

12.1 Introduction

Compiling code originally developed for a different system or toolkit is usually straightforward, especially given the extensive compatibility options supported by the tools. This chapter gives pointers on working around the most common differences among systems and compilers.

12.2 Compilation Issues

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

Look for Missing Standard Header Files

Different systems have different standard header files and the declarations within the header files may be different. Use the **-i** *file1=file2* option to change the name of a missing header file (see *5.3.13 Modify Header File Processing (-i file1=file2)*, p.39 for details).

Older C Code

Look for Code Using Loose Typing Control

Some older C code is written for compilers that do 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

C code written for older UNIX compilers, such as PCC (Portable C Compiler), may not be compatible with the C standard. Use the **-Xdialect-pcc** option to enable some older language constructs. See *B. Compatibility Modes: ANSI, PCC, and K&R C* for more information.

Older Versions of the Compiler

C++ Coding Conventions

When exceptions and run-time type information are enabled (**-Xrtti** and **-Xexceptions**), the current compiler supports the C++ standard. Source code written for earlier versions of the Wind River (Diab) C++ compiler may require modification before it can be compiled with version 5.0 or later. We strongly recommend bringing all source code into compliance with the ANSI standard, but if time does not permit this, you can use the **-Xc++-old** option to invoke the older compiler.

C++ Libraries

Older (pre-5.0) versions of the compiler require different C++ libraries:

Default library	Old library
libd.a	**libdold.a**
libstl.a	**libios.a**, **libcomplex.a**
libstlstd.a	**libios.a**, **libcomplex.a**
libstlabr.a	(none)

See *32.2.1 Libraries Supplied*, p.444 for more information.

When **-Xc++-old** is specified, the **dplus** driver automatically selects the appropriate standard C++ library—that is, it invokes **-ldold** instead of **-ld** to link **libdold.a** instead of **libd.a**. However, to link the older **iostream** and complex libraries, you must use the **-l** option (see *Specify Library or File to Process (-lname, -l:filename)*, p.365) explicitly. If you use the **dcc** driver or invoke **dld** directly, all the old libraries must be specified explicitly. Examples:

```
dplus -Xc++-old hello.cpp
dplus -Xc++-old -lios -lcomplex hello.cpp
dcc -Xc++-old -ldold -lios -lcomplex hello.cpp
dld -YP,search-path -l:windiss/crt0.o hello.o
    -o hello -ldold -lios -lc version-path/conf/default.dld
```

In the first two examples, **-ldold** is invoked automatically because of **-Xc++-old**. In the second two examples, all the older C++ libraries must be specified explicitly.

NOTE: The **-Xc++-old** option cannot be used selectively within a project. If this option is used, all files must be compiled and linked with **-Xc++-old** to make the output binary-compatible. Selective use of **-Xc++-old** should produce linking errors; if it does not, the resulting executable is still likely to be unstable.

VxWorks developers should not use **-Xc++-old**.

To select the old compiler and libraries by default (eliminating the need for **-Xc++-old**), create a **user.conf** file in which **DCXXOLD** is set to **YES** and **ULFLAGS2** invokes the old libraries. For example:

```
# Select old compiler
DCXXOLD=YES
# Add these as default C++ libraries
ULFLAGS2="-ldold -liosold"
```

For more information, see *A. Configuration Files* and *2.3 Environment Variables*, p.15.

Startup and Termination Code

If you are compiling legacy projects that used old-style **.init$***nn* and **.fini$***nn* code sections to invoke initialization and finalization functions, or if your code designates initialization and finalization functions with old-style **_STI__***nn*_ and **_STD__***nn*_ prefixes, you may get compiler or linker errors. The **-Xinit-section=2 option** (see *5.4.69 Control Generation of Initialization and Finalization Sections (-Xinit-section)*, p.83**)** allows you to continue using old-style startup and termination. The recommended practice, however, is to adopt the new method of creating startup and termination code—that is, using attributes to designate initialization and finalization functions, and **.ctors** and **.dtors** sections to invoke them at run-time. See *15.4.8 Run-time Initialization and Termination*, p.260 for more information.

12.3 Execution Issues

The following list includes 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 constructs. See *11. The Lint Facility*.

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. Note: option **-Xmemory-is-volatile** disables some optimizations which may produce slower code.
- Use of uninitialized variables 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. The compiler 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 *__packed__ and packed Keywords*, p.137 on methods for accessing byte-swapped data.
- Floating point format.
- Sign of plain **chars** (those declared without either the **signed** or **unsigned** keyword). By default plain **char** is un**signed**. To force a convention opposite to the default, see *5.4.20 Specify Sign of Plain Char (-Xchar-signed, -Xchar-unsigned)*, p.63.
- Sign of plain **int** bit-fields. bit-fields of type **int** are unsigned by default. Use the option **-Xbit-fields-signed** (C only) to be compatible with systems that treat plain **int** bit-fields as signed.

12.4 GNU Command-Line Options

By default, GCC option flags from the command line or makefile are parsed and, if possible, translated to equivalent Wind River options. Translations are determined by the tables in the file **gcc_parser.conf**. Use **-Xgcc-options-off** to disable this feature. **-Xgcc-options-verbose** outputs a list of translated options.

13

C++ Features and Compatibility

This chapter describes compiler's implementation of the ANSI C++ standard. For more information, see the references cited in *Additional Documentation*, p.8.

13.1 Header Files

The C++ compiler supports all ANSI-specified header files. Generally C++ uses the same header files as C (see *33. Header Files*), but the C++ standard imposes additional requirements on standard C header files and the declarations need to be adjusted to work in both environments. See *13.3 Migration From C to C++*, p.221 below.

13.2 C++ Standard Libraries

The Wind River Compiler includes two versions of the standard C++ library. The complete version provides full support for exceptions. The abridged version does not provide exception-handling functions, the **type_info** class for RTTI support, or complete STL functionality.

The abridged version produces smaller, faster executables than the complete version, but the difference in size and speed varies from project to project. In general, the more an application uses the Standard Template Library, the greater the benefit from switching to the abridged version.

To use the standard library, include one of the following linker options in your project makefile:

Option	Library
-lstl	Link to the complete standard library.
-lstlstd	Same as **-lstl**.
-lstlabr	Link to the abridged standard library.

Projects that use any part of the standard library (including **iostreams**) must specify one of these linker options. For more information about library modules, see *32. Library Structure, Rebuilding*.

NOTE: VxWorks developers should not specify any of the **-lstl...** options listed above. To select a C++ library for VxWorks projects, see the documentation that accompanied your VxWorks development tools.

To use the abridged library, you must also specify the **-Xc++-abr** compiler option. For example:

```
dplus -Xc++-abr file1.cpp
```

-Xc++-abr automatically disables exception-handling (**-Xexceptions=off**).

For projects that use the *complete* C++ library, exception-handling must be enabled (**-Xexceptions**, the default). For projects that use the abridged version, exception-handling may be enabled as long as no exception propagates through the library.

While the compiler supports the **wchar_t** type, in most environments the libraries do not support locales, wide- or multibyte-character functions, or the **long double** type. (Some VxWorks files may include stubs for unsupported wide-character

functions.) For user-mode (RTP) VxWorks projects, the libraries support wide-character functions.

Nonstandard Functions

The C++ libraries include definitions for certain traditional but nonstandard Standard Template Library and **iostream** functions. You can omit these definitions by editing the file *version_path*/**include/cpp/yvals.h**.

To omit the Standard Template Library extensions, change the definition of **_HAS_TRADITIONAL_STL** to:

```
#define _HAS_TRADITIONAL_STL        0
```

To omit the iostream extensions, change the definition of **_HAS_TRADITIONAL_IOSTREAMS** to:

```
#define _HAS_TRADITIONAL_IOSTREAMS  0
```

To see which functions are nonstandard, look for the **_HAS_TRADITIONAL_STL** and **_HAS_TRADITIONAL_IOSTREAMS** macros in the library header files.

13.3 Migration From C to C++

When C functions are converted to C++ or called from a C++ program, minor differences between the languages must be observed 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 header files for code that will be used in both C and C++ modules.

To call a C function from a C++ program, declare the prototype with **extern "C"** (to avoid name mangling) and declare the arguments in C++-compatible format. The **extern "C"** specification may apply to the single declaration that follows or to all declarations in a block. For example:

```
extern "C" int f (char c);

extern "C"
{
#include  "my_c_lib.h"
}
```

For information about calling C++ functions from C modules, see *9.4 C++ Argument Passing*, p.177.

A few general differences between C and C++ are listed below. For more information, see *Additional Documentation*, p.8.

- 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**.
- An **enum** always has the size of an **int** in C, but can have another size in C++.
- The name scope of a **struct** or **typedef** differs slightly between C and C++.
- There are additional keywords in C++ (such as **catch**, **class**, **delete**, **friend**, **inline, new**, **operator**, **private**, **protected**, **public**, **template**, **throw**, **try**, **this**, and **virtual**) that could make it necessary to modify C programs in which these keywords occur as declared identifiers.
- In C, a global **const** has external linkage by default. In C++, **static** or **extern** must be used explicitly.

13.4 Implementation-Specific C++ Features

This subsection describes features of C++ that may behave differently in other implementations of the language.

Construction and 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 constructor and destructor operations are carried out by code in the initialization and finalization sections as described under *15.4 Startup and Termination Code*, p.254.

Templates

Function and class templates are implemented according to the standard.

Template Instantiation

There are two ways to control instantiation of templates. By default, templates are instantiated *implicitly*—that is, they are instantiated by the compiler whenever a template is used. For greater control of template instantiation, the **-Ximplicit-templates-off** option tells the compiler to instantiate templates only where explicitly called for in source code—for example:

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

The compiler options summarized below control multiple instantiation of templates.

Options Related to Template Instantiation in C++

-Ximplicit-templates (*5.4.65 Control Template Instantiation (-Ximplicit-templates...)*, p.82)
Instantiate each template wherever used. This is the default.

-Ximplicit-templates-off (*5.4.65 Control Template Instantiation (-Ximplicit-templates...)*, p.82)
Instantiate templates only when explicitly instantiated in code.

-Xcomdat (*5.4.26 Mark Sections as COMDAT for Linker Collapse (-Xcomdat)*, p.66)
When templates are instantiated implicitly, mark each generated code or data section as “comdat”. The linker collapses identical instances so marked into a single instance in memory. This is the default.

-Xcomdat-off (*5.4.26 Mark Sections as COMDAT for Linker Collapse (-Xcomdat)*, p.66)
Generate template instantiations and inline functions as static entities in the resulting object file. Can result in multiple instances of static member-function or class variables. This requires that **-Ximplicit-templates-off** be enabled.

-Xcomdat-info-file (*5.4.27 Maintain Project-wide COMDAT List (-Xcomdat-info-file), p.66*)
Maintain a list of COMDAT entries across modules. Speeds up builds and reduces object-file size, but has no effect on final executables.

-Xexpl-instantiations (*Write Explicit Instantiations File (-Xexpl-instantiations)*, p.371)

This *linker* option writes a file of all instantiations to **stdout**. Can be used with **-Xcomdat-off** to generate a complete list of template instantiations; source code can then be edited to explicitly instantiate templates where needed and then recompiled with **-Ximplicit-templates-off**.

This option is deprecated.

Using Export With Templates

There are two constraints on the use of the **export** keyword:

- An exported template must be declared exported in any translation unit in which it is instantiated (not just in the translation unit in which it is defined). In practice, this means that an exported template should be declared with **export** in a header file.
- A translation unit containing the definition of an exported template must be compiled before any translation unit which instantiates that template.

Exceptions

Exception handling provides a mechanism for responding to software-generated errors and other exceptional events. It is implemented according to the standard.

NOTE: See *15. Use in an Embedded Environment* for a notes on implementing exceptions in a multitasking environment.

The generation of exception-handling code can be disabled using the **-Xexceptions=0** compiler option. When this option is enabled, the compiler also flags the keywords **try**, **catch**, and **throw** as errors.

Array New and Delete

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

Namespaces

Namespaces are implemented according to the standard. The compiler option **-Xnamespaces-off** disables namespaces; **-Xnamespaces-on** (the default) enables them.

Undefined Virtual Functions

The C++ standard requires that each virtual function, unless it is declared with the pure-specifier (**=0**), be defined somewhere in the program; this rule applies even if the function is never called. However, no diagnostic is required for programs that violate the rule. Programs with undefined non-pure virtual functions compile and run correctly in some cases, but in others generate “undefined symbol” linker errors.

13.5 C++ Name Mangling

NOTE: To interpret a mangled name, see *Demangling utility*, p.228.

The compiler encodes every function name in a C++ program with information about the types of its arguments and (if appropriate) its class or namespace. This process, called *name mangling*, resolves scope conflicts, enables overloading,

standardizes non-alphanumeric operator names, and helps the linker detect errors. Some variable names are also mangled.

When C code is linked with C++ code, the C functions must be declared with the **extern "C"** linkage specification, which tells the C++ compiler not to mangle their names. (The **main** function, however, is never mangled.) See *13.3 Migration From C to C++*, p.221 for examples.

The scheme used for mangling follows the suggestions in *The Annotated C++ Reference Manual* (by Ellis and Stroustrup), which should be consulted for details. In a mangled name, two underscore characters separate the original name from the other encoded information. For this reason, the user should avoid double underscores in class or function names.

A function name is encoded with the types of its arguments. A member function also has the class name or namespace encoded with it. The names of classes and other user-defined types are encoded as the length of the name in decimal followed by the name itself; nested class names contain the names of all classes in the hierarchy using the **Q** modifier (see the table below), and template class names include the arguments of the template. When necessary, local class names and other identifiers are encoded as the name itself followed by **__L** followed by an arbitrary number. Simple type indicators are single characters.

A global function has a double underscore appended to its name, followed by the indicator **F** and the types of its arguments. For example, **void myFunc(int, float)** would be mangled as **myFunc__Fif**.

A member function has the encoded class name or namespace inserted before the **F** indicator—for example, **myFunc__7MyClassFif**. An **S** preceding the **F** indicates a static member function.

Static data members and variables that are members of namespaces are also mangled. Their mangled form consists of a double underscore appended to the variable name, followed by the encoded class name or namespace—for example, **myNumber__7MyClass**.

Functions that instantiate or specialize templates have a template signature. Template parameters are encoded as **Z***n***Z**, where *n* is the parameter's position (starting with 1); if a parameter's depth is greater than 1, it is encoded as **Z***n*_*m***Z**, where *m* is parameter depth. The return type is also included in the mangled name. An **__S** after a template name indicates that the template is specialized; an **__S** after the argument list indicates that the instance is specialized. The **__S** indicator is similarly placed in the encoded names of parent classes of functions and static data members generated from templates.

For constructors, destructors, operator class members, and certain other constructs, a special string beginning with two underscores is prefixed to the class name. For example, **__ct** indicates a constructor and **__pl** indicates the **+** operator. See *The Annotated C++ Reference Manual* for details.

Argument types are encoded as follows:

Type Encodings for Name Mangling in C++

A*n*_
Array (followed by the simple type name), where *n* is the array size.

b
bool

d
double

c
char

e
Ellipses parameter (...)

F*type-list*
Function with parameters of types specified by the *type-list*.

f
float

i
int

L
long long

l
long

M*Type1Type2*
Pointer to member in *Type1* of *Type2. Type1* is always of the form *n name*.

M*mn*
Repeat *m* arguments with the same type as argument number *n*. *m* is limited to a single digit.

nName
User-defined type, with *n* giving the length of *Name* and *Name* giving the type name.

P*type*
Pointer to *type*.

Q*m_n1name1 n2name2...*
Nested class name or namespace: *m* user-defined type names after Q*m*.

R*type*
Reference to *type*.

r
long double

s
short

T *n*
Same type as argument number *n*.

v
void

w
wchar_t

The following modifiers are inserted before the type indicator. If more than one modifier is used, they appear in alphabetical order.

Modifiers for Type Encodings

C
const type

S
signed type

U
unsigned type

V
volatile type

Demangling utility

To interpret a mangled name, enter

```
ddump -F
```

and then interactively enter mangled names one per line. **ddump** displays the demangled meaning of the name after each entry. If the entry is not a valid mangled name, there will be no output.

Table 13-1 **Examples of ddump -F**

Entry to ddump	Interpreted result
`myfunc__Fv`	`myfunc(void )`
`mymain__FiPPc`	`mymain (int , char **)`

13.6 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**. It will still be necessary to account for allocations and deallocations not performed through contructors and destructors of automatic objects.

13.7 Precompiled Headers

In projects with many header files, a large part of the compilation time is spent opening and parsing included headers. (To see how many header files are opened during compilation, use the **-H** option.) You can speed up compilation by using precompiled headers, enabled with the **-Xpch-...** options. The easiest option to use is **-Xpch-automatic**. For example:

```
dplus -Xpch-automatic file1.cpp
```

compiles **file1.cpp** using precompiled headers. This means that a set of header files is saved in a preparsed state and reused each time **file1.cpp** is compiled. The first

time you compile a project with **-Xpch-automatic** you will probably not notice an improvement in speed, but subsequent compilations should be faster.

Within a header file, use **#pragma no_pch** to suppress all generation of precompiled headers from that file. To selectively suppress generation of precompiled headers, use **#pragma hdrstop**; headers included after **#pragma hdrstop** are not saved in a parsed state.

Precompiled headers are supported by the C++ compiler only.

PCH Files

Parsed headers are saved in PCH (precompiled header) files. The compiler processes PCH files only if one of the following options is enabled: **-Xpch-automatic**, **-Xpch-create=***filename*, or **-Xpch-use=***filename*. If more than one of these options is given, only the first is considered.

When **-Xpch-automatic** is enabled, the compiler looks for a PCH file in the current working directory (unless you use **-Xpch-directory=***directory* to specify a different location) and, if possible, uses the preparsed headers in that file. Otherwise a PCH file is generated with the default name *sourcefile***.pch**, where *sourcefile* is the name of the primary source-code file. When the source file is recompiled, or when another file is compiled in the same directory, *sourcefile***.pch** is checked for suitability and used if possible.

Before using a PCH file, the compiler always verifies that it was created in the correct directory using the same compiler version, command-line options, and header-file versions as the current compilation; this information is stored in each PCH file. If more than one PCH file is applicable to a compilation, the compiler uses the largest file available.

If you want to specify a name for the generated PCH file, use **-Xpch-create=***filename* instead of **-Xpch-automatic**:

```
dplus -Xpch-create=myPCH file1.cpp
```

Later, you can reuse **myPCH**—when compiling the same file or a different file—by specifying **-Xpch-use=***filename*:

```
dplus -Xpch-use=myPCH file2.cpp
```

The *filename* specified with **-Xpch-create** or **-Xpch-use** can include a full directory path, or the option can be combined with **-Xpch-directory**:

```
dplus -Xpch-use=myPCH -Xpch-directory=/source/headers somefile.cpp
```

Limitations and Trade-offs

A generated PCH file includes a snapshot of all the code preceding the *header stop point*—that is, **#pragma hdrstop** or the first token in the primary source file that does not belong to a preprocessor directive. If the header stop point appears within an **#if** block, the PCH file stops at the outermost enclosing **#if**.

A PCH file is *not* generated if the header stop point appears within:

- An **#if** block or **#define** started within a header file.
- A declaration started within a header file.
- A linkage specification's declaration list.
- An unclosed scope, such as a class declaration, established by a header file. (In other words, the header stop point must appear at file scope.)

Further, a PCH file is *not* generated if the header stop point is preceded by:

- A reference to the predefined macro **__DATE__** or **__TIME__**.
- The **#line** preprocessing directive.

A PCH file is generated only if the code preceding the header stop point has produced no errors and has introduced a sufficient number of declarations to justify the overhead associated with precompiled headers. Finally, a PCH file is generated only if sufficient memory is available.

Efficient use of precompiled headers requires experimentation and, in most cases, minor changes to source code. PCH files can become bulky; included files must be organized so that headers are preparsed to as few shared PCH files as possible.

Diagnostics

The **-Xpch-messages** option generates a message each time a PCH file is created or used. The **-Xpch-diagnostics** option generates an explanatory message for each PCH file that the compiler locates but is unable to use.

14

Locating Code and Data, Addressing, Access

14.1 Controlling Access to Code and Data

By default, the compiler generates architecture-specific code for locating and accessing code and data in memory which will be suitable for many cases. In addition, a number of options are available for exercising fine control over the process, for locating code and data at specific locations in memory, and for generating position-independent code. All are described in detail in this chapter.

section and use_section Pragmas

Code and data are generated in *sections* in an object file, combined by the linker into an executable file, and ultimately located in target memory at specific locations. Default sections are predefined and have certain attributes. To change the name of a default section, use the **-Xname-...** option (see *5.4.93 Specify Section Name*

(-Xname-...), p.94). The **section** and **use_section** pragmas may be used to change the default attributes, to define new sections, and to control the assignment of code and variables to particular sections and, along with the linker command file, their locations.

#pragma *section class_name* [*istring*] [*ustring*] [*addr-mode*] [*acc-mode*] [**address**=*x*]
#pragma use_section *class_name* [*variable* | *function*] , ...

class_name
Required. Symbolic name for a predefined or user-defined section class to hold objects of a particular *class*, e.g., code, initialized variables, or uninitialized variables.

istring
Name of the actual section to contain initialized data. *For variables, this means those declared with an initializer* (e.g., **int x=1;**). Use empty quotes if this section is not needed but the *ustring* is.

ustring
Name of actual section to contain uninitialized data. For variables, this means those declared with no initializer (e.g., **int x;**). This name may be omitted if not needed (the default value is used).

addr-mode
Form of addressing mode for access to variables or functions in the section. See *14.2 Addressing Mode — Functions, Variables, Strings*, p.238 for details.

acc-mode
Accessibility to the section. See *14.3 Access Mode — Read, Write, Execute*, p.240 for details.

#pragma section defines a *section class* and, optionally, one or two sections in the class. A section class controls the addressing and accessibility of variables and code placed in an instance of the class.

For C++, **#pragma section** declarations apply to all global and namepace scope variables, class static member variables, global and namespace scope functions, and class member functions that follow the pragma.

#pragma use_section selects a section class for specific variables or functions after the section class has been defined by **#pragma section**.

Notes for #pragma section and #pragma use_section

The C++ compiler has the following limitations for **#pragma section** and **#pragma use_section**:

- Templates are not affected by **#pragma section** or **#pragma use_section**. However, you can alter the placement of all the data or code in a file (including templates) by using the command-line options **-Xname-data** (and related options, such as **-Xname-sdata** or **-Xname-const**) or **-Xname-code**. See *5.4.93 Specify Section Name (-Xname-...)*, p.94 for more information on these options.
- **#pragma section STRING** cannot be used to alter the placement of strings. Instead, use the command-line option **-Xname-string**.
- **#pragma use_section** must be followed by at least one declaration or definition of an entity for it to apply to that entity, as in:

```
#pragma section MYCODE ".mycode"
void my_func()
{
}
```

- A section *class_name* (e.g., **DATA**) is the symbolic name of a section class and it is used only in writing **#pragma section** and **#pragma use_section** directives.

  At any given point in the source, there may be up to two physical sections associated with a section class—an initialized section and an uninitialized section as named by the *istring* and *ustring* attributes to **#pragma section** respectively (e.g., "**.data**"). It is these physical sections which will appear in the object file and which may be manipulated during linking.

- *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 assembler **.section** directive to switch to the desired section for initialized data. An empty string or no string at all indicates that the default value should be used. Note that a section to contain code is "initialized" with the code. 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 the desired 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. See *23.4 COMMON Sections*, p.352 regarding allocation of common variables for full details; generally however, **COMM** sections are gathered together by the linker an placed at the end of the **.bss** output section. Examples:

```
".bss", ".data", "COMM"
```

- *Predefined section classes*: Except when a user-defined section class has been specified, all variables and functions are categorized by default into one of several predefined section classes depending on how they are defined. Each predefined section class is defined by default values for all of its attributes. Table 14-1 gives the names and attributes of all predefined section classes.
- By using the **#pragma use_section** directive, any variable and function can be individually assigned to any of the predefined section classes, or to a user-defined section class.
- If a **section** pragma for some class is given with no values for one or more of the attributes, those attributes are always restored to their default values as given in Table 14-1. This is true even for a user-defined *class_name* (the table shows the default attributes in this case as well).
- Multiple **#pragma section** directives with different attributes can be given for the same *class_name*. Variables and functions use the earliest non-default directive that is valid at the point of definition. (This behavior can be changed with the **-Xpragma-section-last** directive; see *5.4.104 Control Interpretation of Multiple Section Pragmas (-Xpragma-section-...)*, p.99.)
- Pragmas are not seen across modules unless a common header file is included.
- The compiler associates each function with a storage space at the point in a module where it is first declared or defined. Subsequent attempts within the same module to assign a function to a storage space are ignored.
- For functions that are declared multiple times, the first section binding applies, unless the **-Xpragma-section-last** option has been specified. For example:

```
void my_func();     /* binds to "text" */

#pragma section CODE ".mycode"

void my_func()     /* does not override previous binding unless
                    -Xpragma_section-last has been used */
```

  In this example, to force **my_func** to go into **.mycode**, you need to do one of the following:

  - Move the **#pragma section** before the initial declaration of **my_func**.
  - Specify **-Xpragma-section-last** on the command line.
  - Use **#pragma use_section**:

```
void my_func();

#pragma section CODE ".mycode"
#pragma use_section CODE my_func

void my func()
{
}
```

Section Classes and Their Default Attributes

Table 14-1 below gives the predefined section classes and their default attributes, and also the defaults for a user-defined section class.

Table 14-1 **Section Classes and Their Default Attributes**

section *class_name*	**Description and Example**	**Default** *istring*	*ustring*	*addr-mode*	*acc-mode*

CODE	code generated in functions and global **asm** statements:
	`int cube(int n)`
	`{ return n*n*n; }`
		.text	n/a	**standard**	**RX**

DATA	static and global variables: `static int a[10];`	**.data**	**COMM**	**far-absolute**	**RW**
CONST	**const** variables: `const int a[10] = {1, ...};`	**.rodata**	**.rodata**	**far-absolute**	**R**
STRING	string constants: `"hello\n"`		n/a	**far-absolute**	**R**
user-defined	`#pragma section USER ...`	**.data**	**COMM**	**far-absolute**	**RW**

Notes for Table 14-1:

- *14.4 Local Data Area (-Xlocal-data-area)*, p.247Local data area optimization: global and static scalar variables may be placed in a *local data area* if **-Xlocal-data-area**, which has a default value of 64 bytes, is non-zero and optimization is in effect (either **-O** or **-XO** is present). The local data area will be placed in the **.data** section for the module if any such variable in it has an initial value, or in the **.bss** section for the module if none do. When uninitialized

variables are placed in the **.data** section in this way, it overrides the default **COMM** (common) section name as given above. See *14.4 Local Data Area (-Xlocal-data-area)*, p.247 for further details and restrictions.

- The section names shown in the table assume the default value for option **-Xconst-in-text**. See *Moving initialized Data From "text" to "data"*, p.245 if **-Xconst-in-text** is set to a non-default value.
- Dynamically initialized C++ **const** variables are treated like uninitialized non-**const** variables. For example:

```
int f();
const int x = f();
```

By default, **x** is placed in the **.bss** section.

14.2 Addressing Mode — Functions, Variables, Strings

The *addr-mode* for a section is the addressing mode to be used when referencing a variable, function, or string in the section. It is one of the hex numbers given in the "Code" column of the following table. For the relative addressing modes constructed from a base register and offset, the table also shows the base register and the number of bits in the offset. Notes follow the table. See *Implementation*, p.244 for examples of code generated for each addressing mode.

Notes:

Table 14-2 ***addr-mode* Definitions**

addr-mode				
Name	**Code**	**Description**	**Bits**	**Base Register**
standard	0x01	See Notes below.		
far-absolute	0x11	absolute	32	not applicable
far-data	0x21	data relative	32	**r9**

- The "code" hexadecimal number is used for command-line options described below.

- The *addr-mode* **standard** for the **CODE** section class means that a processor specific method is being used, usually defined to minimize access time.
  - Branches are PC-relative using 24-bit offsets (setting the addressing mode to **far-absolute** would generate 32-bit offsets).
  - Function pointers are always absolute.

  The *addr-mode* **standard** for data sections is equivalent to **far-absolute**.
- Position-independent Data (PID) can be achieved by using the data relative addressing modes.

Default *addr-mode* values for the predefined section classes are shown in Table 14-1.

The following options change the default *addr-mode*:

-Xaddr-data=*mode*

-Xaddr-string=*mode*

-Xaddr-code=*mode*

-Xaddr-const=*mode*

-Xaddr-user=*mode*

These options direct that the named section class, **DATA**, **CONST**, etc., be addressed with the given addressing mode. *mode* is a hexadecimal number as given in the "code" column in Table 14-2.

Example: address all variables in the **DATA** section class with **far-data** addressing:

```
-Xaddr-data=0x21
```

The following table describes other command-line options that will affect the default *addr-mode*:

Table 14-3 **-X Option Settings Implied by Other -X Options**

Option	Sets All of	
-Xcode-absolute-near	**-Xaddr-const=0x10** **-Xaddr-string=0x10**	**-Xaddr-sconst=0x10** **-Xaddr-code=0x10**
-Xcode-absolute-far	**-Xaddr-const=0x11** **-Xaddr-string=0x11**	**-Xaddr-sconst=0x10** **-Xaddr-code=0x11**

Table 14-3 **-X Option Settings Implied by Other -X Options** (cont'd)

Option	Sets All of	
-Xcode-relative-near-all	**-Xaddr-const=0x40** **-Xaddr-string=0x40** **-Xaddr-data=0x40** **-Xaddr-user=0x40**	**-Xaddr-sconst=0x40** **-Xaddr-code=0x40** **-Xaddr-sdata=0x40**
-Xcode-relative-far-all	**-Xaddr-const=0x41** **-Xaddr-string=0x41** **-Xaddr-data=0x41** **-Xaddr-user=0x41**	**-Xaddr-sconst=0x40** **-Xaddr-code=0x41** **-Xaddr-sdata=0x41**
-Xdata-absolute-near	**-Xaddr-data=0x10** **-Xaddr-user=0x10**	
-Xdata-absolute-far	**-Xaddr-data=0x11** **-Xaddr-user=0x11**	
-Xdata-relative-near	**-Xaddr-data=0x20** **-Xaddr-user=0x20**	
-Xdata-relative-far	**-Xaddr-data=0x21** **-Xaddr-user=0x21**	

14.3 Access Mode — Read, Write, Execute

acc-mode defines how the section can be accessed and is any combination of:

`R`
Read permission.

`W`
Write permission.

`X`
Execute permission.

O

COMDAT — when the linker encounters multiple *identical* sections marked as "comdat", it collapses the sections into a single section to which all references are made and deletes the remaining instances of the section.

This is used, for example, with templates in C++. If COMDAT sections are disabled (**-Xcomdat-off**), the compiler generates a template instance for each module that uses a template, which can result in duplicate template instantiations. With the **-Xcomdat** option, the compiler uses "**O**" to mark sections generated for templates as COMDAT; the linker then collapses identical instantiations into a single instance. See *5.4.26 Mark Sections as COMDAT for Linker Collapse (-Xcomdat)*, p.66.

N

"not allocatable" —the section is not to occupy space in target memory. This is used, for example, with debug information sections such as **.debug** in ELF. **N** must be used by itself; it is ignored when it is combined with other flags.

acc-mode is used by the assembler and loader. It does not affect type-checking during compilation.

Default *acc-mode* values for the predefined section classes are shown in Table 14-1.

If **-Xconst-in-text=0** then the **CONST**, **SCONST**, and **STRING** section classes have will have access mode **RW** (read/write) rather than the default **R** (read only). See *Moving initialized Data From "text" to "data"*, p.245 for further details.

Multiple instances of a constant allocated to a section with no write access (**W**) may be collapsed by the compiler to a single instance.

Using #pragma section and #pragma use_section to Locate Variables and Functions at Absolute Addresses

There are two ways to put a variable or function in a specific section.

- 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:
  - Using the defaults, **ar1** is placed in the **DATA** section class (**.data**) and referenced using far-absolute addressing:

```
int ar1[100] = { 0 };
```

  - **ar2** is placed in section **.absdata** and referenced using near-absolute addressing:

```
#pragma section DATA ".absdata" near-absolute
    int ar2[100] = { 0 };
```

– **ar3** is again placed in the default **DATA** section class (**.data**) — because no *istring*, *ustring*, *addr-mode*, or *acc-mode* is given, the default values for these attributes as given in Table 14-1 are used.

```
#pragma section DATA
    int ar3[100] = { 0 };
```

– A variable or function can be placed by specifying a specific section in a **#pragma use_section**. Example:

  – **ar4** is placed in section **.absdata** and referenced using near-absolute addressing (see the next heading regarding the empty quotes in this example):

```
#pragma section VECTOR "" ".absdata" near-absolute RW
#pragma use_section VECTOR ar4
    int ar4[100];
```

Placing Initialized vs. Uninitialized Variables

When defining a data section class to hold variables, the **section** pragma can name two sections: one for initialized variables and one for uninitialized variables, or either section by itself. Repeating from the definition above (*section and use_section Pragmas*, p.233):

`#pragma section` *class_name* [*istring*] [*ustring*] ....

class_name
Required. Predefined or user-defined name to hold objects of a particular *class*, e.g., code, initialized variables, or uninitialized variables.

istring
Name of actual section to contain initialized data. *For variables, this means those declared with an initializer* (e.g., **int x=1;**). Use empty quotes if this section is not needed but the *ustring* is.

ustring
Name of actual section to contain uninitialized data. *For variables, this means those declared with no initializer* (e.g., **int x;**). This section may be omitted if not needed (which will assign the default value).

Consider these examples:

```
#pragma section DATA ".inits" ".uninits"
    int init=1;
    int uninit;
```

Assuming no earlier pragmas for class **DATA**, the pragma changes the section for initialized variables from **.data** to **.inits**, and changes the section for uninitialized variables from **COMMON** (which the linker adds to **.bss**) to **.uninits**. As a result,

variable **init** will be placed in the **.inits** section (because **init** has an initial value), while variable **uninit** will be placed in the **.uninits** section because it has no initial value.

The following shows a common error:

```
#pragma section DATA ".special"    /* probably error */
    init special;
```

The user presumably intends for variable **special** to be placed in section **.special**. But the pragma defines **.special** as the section for initialized variables. Because variable **special** is uninitialized, it will be placed in the default **COMMON** section. Changing the above to

```
#pragma section DATA "" ".special"
    int special;
```

achieves the intended result because **.special** is now the section for uninitialized variables.

Using the Address Clause to Locate Variables and Functions at Absolute Addresses

The **address**=*n* clause provides a way to place variables and functions at a specific absolute address in memory. With this form, the linker will put the designated code or data in an absolute section named "**.abs.***nnnnnnnn*" where *nnnnnnnn* is the value in hexadecimal, zero-filled to eight digits, of the address given in the **address=***n* clause.

NOTE: When using the **address=***n* clause, any section name given by *istring* or *ustring* will be ignored.

Advantages of using absolute sections (see *15.9.3 Accessing Variables and Functions at Specific Addresses*, p.268):

- I/O registers, global system variables, and interrupt handlers, etc., can be placed at the correct address from the compiled program without the need to write a complex linker command file.

  That is, if you know the address of an object at compile-time, the **address** clause of the **#pragma section** directive can be used in your source. If the location of the object is best left to link-time, use a **#pragma section** directive with a named section which can then by located via a linker command file.

- A symbolic debugger will have all information necessary for full access to absolute variables, including types. Variables defined in a linker command file cannot 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() {
// ...
}
```

Prototypes and the Placement of Sections

If function prototypes are present, the compiler and linker select sections and their attributes for functions and, in C++, **static** class variables, based on where the prototypes of the functions appear in the source, rather than where the function definitions appear.

The following example shows the wrong way to request the compiler and linker to place the function **fun( )** in the **.myTEXT** section.

```
int fun();                      // Prototype determines "fun" section
...
#pragma section CODE ".myTEXT" // #pragma before definition has no
int fun() {                     //   effect on placement of "fun"
...
}
```

In this example, the initial declaration of **fun( )** determines where it will appear in the executable; the subsequent **#pragma** is ignored. This is consistent with the behavior of the C++ compiler.

Implementation

The compiler will generate the assembly code for the different *addr-mode* settings as shown in Table 14-4. The corresponding code is as follows (the **#pragma use_section** is present to ensure that the variable **var** is placed in **DATA** rather than **SDATA** for simplicity).

```
#pragma use_section DATA var
int var=1;      /* var in DATA or SDATA (not in .bss or .sbss) */

reg = var;
func();         /* func in CODE */
```

Notes for Table 14-4:

- The compiler may select a different register for the **reg** variable than is shown in the table.
- To reproduce the code as shown, place the above code in a file, e.g. **test.c**, and use **-Xaddr-code** and **-Xaddr-data** to set the addressing modes, and **-g** to turn on debugging (this disables some minor optimizations which might otherwise be present). For example, for **standard** addressing mode:

```
dcc -g -S -Xaddr-code=0x01 -Xaddr-data=0x01 -Xpass-source test.c
```

Table 14-4 **Code Generated for Different Addressing Modes**

Mode	Reference to DATA: reg = var;	Reference to CODE: func( )
standard	`ldr r10,=var` `ldr r10,[r10,#0]`	`bl func`

far-absolute	`ldr r10,=var`
	`ldr r10,[r10,#0]`
		`ldr r12,=func`
		`ldr lr,=.L1`
		`mov pc,r12`
		`.L1:`

far-data	`ldr r10,=var@data`
	`ldr r10,[r10,r9]`
		`ldr r12,=func@`
		`add r12,r12,r9`
		`ldr lr,=.L1`
		`mov pc,r12`
		`.L1:`

Notes:

- The assembler uses some special ARM relocation types for the operators used in the table above. See *F.1.6 ELF Relocation Information*, p.584 for the complete list of relocation types. See also **include/elf_arm.h**.

Moving initialized Data From “text” to “data”

Sections that hold setable variables are generically referred to as “data” sections (and should be in RAM), while sections that hold code, constants like strings, and unchangeable **const** variables are generically referred to as “text” sections (and can be in ROM).

The **-Xconst-in-text** option provides a shortcut for controlling the default section for initialized data (*istring*) for the **CONST**, and **STRING** constant section classes. Its form is:

-Xconst-in-text=*mask*

where *mask* bit 0x1 controls **const** variables in the **CONST** section class, and 0x4 controls string data in the **STRING** section class.

If a *mask* bit is set to 1, variables or strings belonging to the corresponding section classes are placed in ROMable "text" sections; if set to 0, they are placed in "data" sections.

By default, **-Xconst-in-text=0xff**. This gives the behavior shown in the following table. (Note: the table shows section names for initialized sections. *See notes following the table for uninitialized sections.*)

Table 14-5 **-Xconst-in-text mask bits**

Section Class	Mask Bit	"text" Section With Mask Bit Set to 1	"data" Section With Mask Bit Set to 0
CONST	0x1	**.rodata** (default)	**.data**
STRING	0x4	**.rodata** (default)	**.data**

NOTE: Note that when a section is in "data" it will have access mode **RW** (read/write), while in "text", the access mode will be **R** (read only). See *14.3 Access Mode — Read, Write, Execute*, p.240. If a section is moved from its default by **-Xconst-in-text**, this will be a change from its usual default access mode.

While the option **-Xconst-in-text** is preferred, the older option **-Xconst-in-data** is equivalent to **-Xconst-in-text=0**, and thus requests that data for all constant sections, **CONST**, and **STRING** be placed in their corresponding "data" sections as given by the last column of the table above, and the older option **-Xstrings-in-text** is equivalent to **-Xconst-in-text=0xf**, and thus requests that data for all constant sections be placed in their default "text" sections.

The table above gives section names for *initialized* sections. There are no uninitialized **STRING** sections. Uninitialized **CONST** sections, if moved from "text" to "data", go in the **COMM** (common) section (which the linker puts at the end of the **.bss** section by default).

14.4 Local Data Area (-Xlocal-data-area)

The compiler supports a local data area (LDA) optimization. This optimization works as follows:

- The LDA optimization applies only to static and global variables of scalar types—not arrays, structures, unions, or classes (for C++).
- Like all optimizations, LDA optimization is enabled only if option **-O** or **-XO** is present. It can be disabled by setting **-Xlocal-data-area=0**.
- An LDA is allocated for each module, and static and global scalar variables *which are referenced at least once* are allocated to it except as noted above. To restrict the optimization to static variables, use **-Xlocal-data-area-static-only**. VxWorks developers are strongly advised to use **-Xlocal-data-area-static-only** so that asynchronous changes to global variables remain visible to the generated code.
- The variables in the LDA are addressed using efficient base register-offset addressing. The base register is chosen for the module by the compiler as part of its normal register assignment algorithms and optimizations.
- If at least one variable in the LDA is initialized, the LDA will be in the **.data** section for the module. If all are uninitialized, the LDA will be in the **.bss** section for the module.

NOTE: Note that this can change the usual behavior for uninitialized variables — without LDA optimization, uninitialized variables go into the **.bss** section. But with LDA optimization, variables to be put into the LDA are put there whether initialized or not; and if any LDA variables are uninitialized, the LDA is placed in the **.data** section for the module, and in that case, any uninitialized variables in the LDA will also be in the **.data** section.

- By default, the size of the LDA is 64 bytes. It may be set to a different size with option **-Xlocal-data-area=***n*. However, a value larger than the default will be less efficient because the default was chosen based on the size of the most efficient offset. If there are too many scalar variables to fit in the LDA, the overflow will be allocated as usual.

14.5 Position-Independent Data (PID)

By the nature of its instruction set, ARM code is automatically position-independent; that is, it may be located and run at any address.

In addition, the compiler also supports position-independent data. The **-Xdata-relative-far** option implements position-independent data by using register **r9** as a pointer to the data section, and making all references to it as offsets from that register.

F.1.6 ELF Relocation Information, 

NOTE: The libraries are compiled with default options and therefore do not use position-independent code and data.

Generating Initializers for Static Variables With Position-Independent Data

Position-independent addresses are not known at compile-time, so it is necessary to dynamically set pointers having constant initial values whose position will not be known until run-time, e.g., pointers to global variables, static local variables, static class variables, functions or methods, whenever these are in position-independent sections.

See *5.4.45 Generate Initializers for Static Variables (-Xdynamic-init)*, p.73 for instructions on storing data in the initialization section when generating position-independent data. Examples:

```
/* Always OK. */
    int i = 1;

/* Following two statements, if compiled with -Xdata-relative-...,
 * would also require -Xdynamic-init because variable i and the
 * string "abc" would be position-independent data and have unknown
 * addresses at compile-time
 */
    int *p = &i;
    char *s = "abc";

/* Following two statements, if compiled with -Xcode-relative-...,
 * would also require -Xdynamic-init because the address of
 * function f would be unknown at compile-time.
 */
    int f (int a);
    int (*f_p)(int) = f;
```

15

Use in an Embedded Environment

15.1 Introduction

Device software development differs significantly from development for native environments, in part because there is often 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 often needed in an embedded environment include:

- control over addressing to minimize code size and maximize execution speed
- 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

15.2 Compiler Options for Embedded Development

The following compile-time options and pragmas control code generation in various ways. All are documented in *5. Invoking the Compiler*.

-Xdollar-in-ident
Allow variable names containing "**$**"-signs.

-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. See *5.4.91 Treat All Variables As Volatile (-Xmemory-is-volatile, -X...-volatile)*, p.93.

-Xsize-opt
Minimize the size of the executable code.

-Xconst-in-text=0xf
Put strings and **const** data in the **.text** section together with code. See *Moving initialized Data From "text" to "data"*, p.245.

-Xmember-max-align
-Xstruct-min-align
Options to pack structures in different ways. See *5.4.90 Set Maximum Structure Member Alignment (-Xmember-max-align=n)*, p.93 and *5.4.133 Set Minimum Structure Member Alignment (-Xstruct-min-align=n)*, p.109.

#pragma interrupt *func*
Specify that a function *func* is an exception handler. See *interrupt Pragma*, p.126.

#pragma pack
Control packing of structures and the byte order of members. See the *pack Pragma*, p.129.

#pragma section ...
Control placement and addressing of variables and functions. See *section and use_section Pragmas*, p.233.

15.3 User Modifications

Since most embedded environments are unique, some things must be modified by the user:

- Startup code must initialize the processor and run-time.
- Hardware exceptions must be handled.
- A linker command file must specify where to allocate code and data.
- It may be necessary to modify library functions to make user-supplied operating system calls.

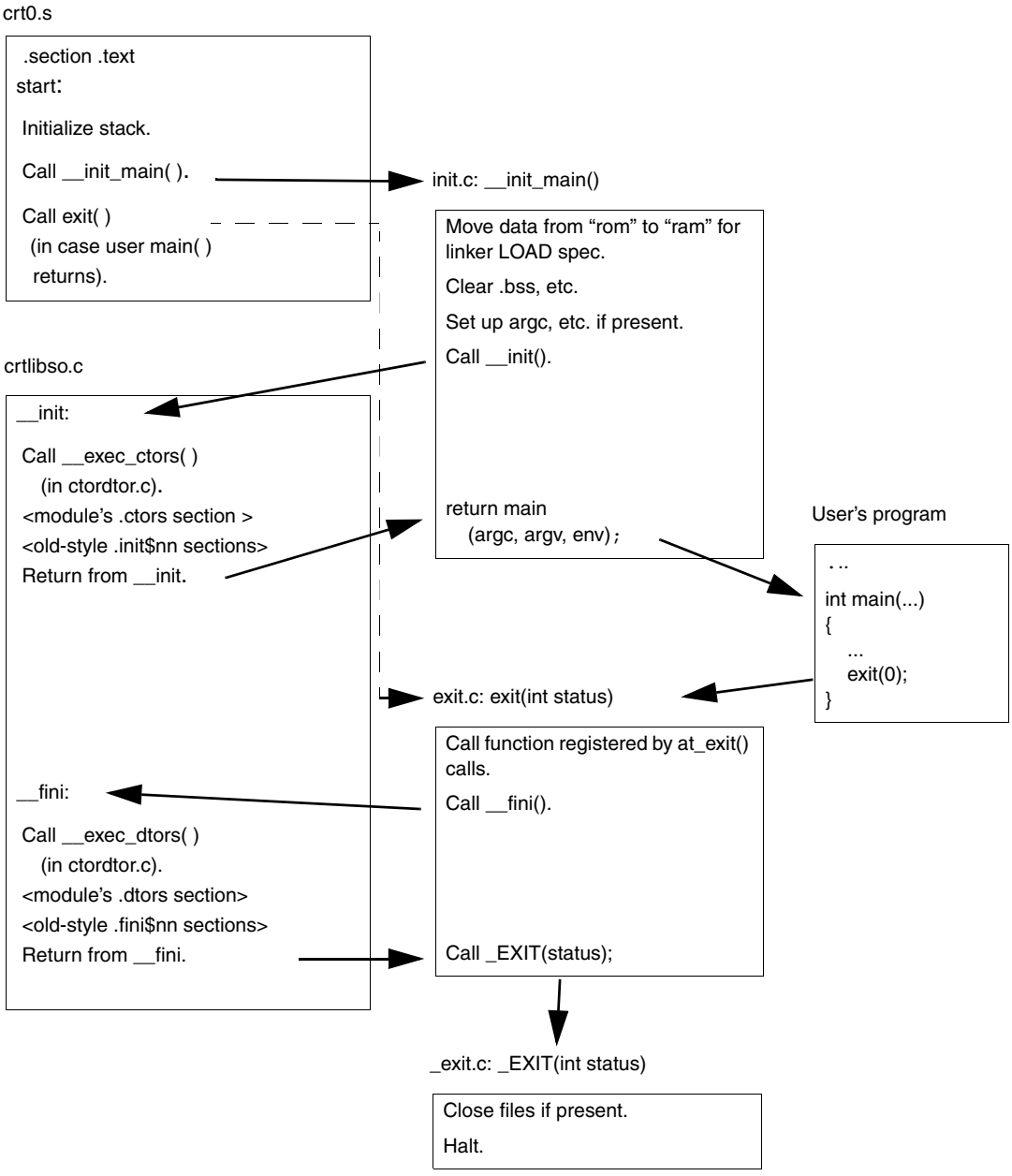
15.4 Startup and Termination Code

This section describes startup and termination for self-contained applications built with the compiler. Applications that run under an operating system (such as VxWorks or Linux) work differently.

As shipped, startup is carried out by four modules: **crt0.s**, **crtlibso.c**, **ctordtor.c**, and **init.c**. Termination is carried out by five modules: **exit.c**, **crt0.s**, **crtlibso.c**, **ctordtor.c**, and **_exit.c**. Read this section and examine these modules to determine whether any modifications are required for your target environment.

An overall schematic for startup and termination is shown in Figure 15-1. This figure applies to all supported targets and does not show some details. See the referenced modules for complete details. Notes, including source locations and modification hints, are in the sub-sections immediately following the figure.

Figure 15-1 **Startup and Termination Program Flow**



15.4.1 Location of Startup and Termination Sources and Objects

The source of **crt0.s** is located in the **src/crtarm** directory. Objects are in the library directories shown in Table 2-2.

init.c, **crtlibso.c**, **exit.c**, and **_exit.c** are in the **src** directory. Objects are in **libc.a**.

15.4.2 Notes for crt0.s

crt0.s begins at label **start**. This is the entry point for the target application.

crt0.s is brief, with most initialization done in **init.c**. Its first action is to initialize the stack to symbol **__SP_INIT**. This symbol is typically defined a *linker command file*. See Figure 25-1 for an example.

Insert assembly code as required to initialize the processor before **crt0.s** calls **__init_main( )** described in *15. Use in an Embedded Environment*. Refer to manufacturer's manuals for the target processor for information on initializing the processor.

To replace **crt0.o**:

- Copy and modify it as required.
- Assemble it with:

  ```
  das crt0.s
  ```

- Link it either by including it on a **dld** command line when invoking the linker, or by using the **-Ws** option if using the compiler driver, e.g.,

  ```
  dcc -Wsnew_crt0.o ... other parameters ...
  ```

  The **-Ws** option can be added to the **user.conf** configuration file to make it permanent.

15.4.3 Notes for crtlibso.c and ctordtor.c

By default, compiled modules generate special **.ctors** and **.dtors** sections for startup and termination code, including constructor functions, destructor functions, and global constructors in C++. The **.ctors** and **.dtors** sections contain pointers to initialization and finalization functions, sorted by priority. This code is invoked during initialization and finalization through calls to **__exec_ctors( )** and **__exec_dtors( )** from the **__init( )** and **__fini( )** functions in **crtlibso.c**. The source

code for **__exec_ctors( )** and **__exec_dtors( )**, along with symbols marking the top and bottom of **.ctors** and **.dtors**, is in **ctordtor.c**. (See Figure 15-1.)

crtlibso.c includes "wrapper" sections **.init$00**, **.init$99**, **.fini$00**, and **.fini$99**. These sections, which previous versions of the compiler used for startup and termination code, exist for backward compatibility.

For more information, see *15.4.8 Run-time Initialization and Termination*, p.260.

NOTE: The **malloc( )** function supplied with the compiler must be initialized. This is done automatically by code generated in the **.ctors** section. If you do not use the standard **crtlibso.c**, then include comparable code in your own startup file. Other library functions may also require initialization, so **__init( )** should be called in all cases.

See also *5.4.45 Generate Initializers for Static Variables (-Xdynamic-init)*, p.73.

15.4.4 Notes for init.c

Initialization code that can be written in C or C++ should be inserted in or called from **__init_main( )**, typically just before calling **main( )**, so that all other initialization done by **__init_main( )**—copying initial values from "rom" to "ram", clearing **.bss**, and so forth—can be done first.

Copying Initial Values From "ROM" to "RAM", Initializing .bss

In a typical embedded system, the initial values for non-**const** variables must be stored in some form of read-only memory, "ROM" for simplicity, while the code must refer to the variables themselves in writable memory, "RAM". At startup, the initial values must be copied from ROM to RAM. In addition, C and C++ require that uninitialized static global memory be initialized to zero.

init.c requires five symbols to "copy constants from ROM to RAM" (the traditional phrase) and to clear **.bss**. These five symbols, all typically defined in a *linker command file*, are:

__DATA_ROM
: Start of the *physical* image of the data section for variables with initial values, including all initial values—the location in "ROM" as defined using the **LOAD** specification in the linker command file.

__DATA_RAM
Start of the *logical* image of the data section — the location in "RAM" where the variables reside during execution as defined by an area specification ("**>** *area-name*") in the linker command file.

__DATA_END
End of the logical image of the data section. **__DATA_END - __DATA_RAM** gives the size in bytes of the memory to be copied.

__BSS_START
Start of the **.bss** section to be cleared to zero.

__BSS_END
End of the **.bss** section.

The code in **init.c** compares **__DATA_ROM** to **__DATA_RAM**; if they are different, it copies the data section image from **__DATA_ROM** to **__DATA_RAM**. It then compares **__BSS_START** with **__BSS_END** and if they are different sets the memory so defined to zero.

As noted, these symbols are typically defined in a linker command file. See *25.6 Command File Structure*, p.383 for an example.

Providing arguments to main and data for memory resident files

Examine the code in **init.c** to see how C-style **main( )** function arguments and environment variables can be set up. The variables used in this code, such as **__argv**[] and **__env**[], are defined in **src/memfile.c** and **src/memfile.h**. These variables, as well as data for memory resident files, can be created using the **setup** program. See *15.11 Target Program Arguments, Environment Variables, and Predefined Files*, p.270 for details.

Replacing init.c

To replace **init.c**:

- Copy and modify it as required.
- Include it as a normal C module in your build.

15.4.5 Notes for Exit Functions

Because embedded systems are often designed to run continuously, **exit( )** may not be needed and will not be included in the target executable if not called.

To replace **exit.c** or **_exit.c**:

- Copy and modify as required.
- Include with normal C modules in your build.

15.4.6 Stack Initialization and Checking

Stack Initialization

The initial stack is initialized by **crt0.s** to symbol **__SP_INIT**, typically defined in the linker command file. See *15.4.2 Notes for crt0.s*, p.256 and for an example see *25.6 Command File Structure*, p.383.

Stack Checking

If the **-Xstack-probe** option is used when compiling, the compiler inserts code in each function prolog to check for stack overflow and to transfer memory from the heap to the stack, if possible, on overflow.

Requirements

- Code compiled with **-Xstack-probe** or **-Xrtc=4** must be linked with the **librta.a** library (typically by using the **-lrta** option in the **dld** linker command line).
- The identifier **__SP_END** must be defined in the linker command file as the lower bound of the stack. See the file **conf/default.dld** for an example.

Code is added to the prolog of each function (but see the note below) to determine whether the stack exceeds **__RTC_SP_LIMIT** (declared in **rtc.h** and initialized to **__SP_END**, typically defined in the linker command file as in *25.1 Example "bubble.dld"*, p.378). If it does, the function **__sp_grow( )** (defined in the **librta.a** library) is called. If the stack and heap are contiguous and there is sufficient space available in the heap, **__sp_grow( )** will reassign the required memory from the heap to the stack and move the boundary recorded in **__RTC_SP_LIMIT**. If the stack cannot be extended, **__sp_grow( )** terminates the program by calling **__rtc_error("stack overflow")**. See *15.6 Library Exception Handling*, p.262 regarding termination.

NOTE: No stack-checking code is inserted in leaf functions (functions that do not call other functions) which require less than 64 bytes of stack. Non-leaf functions always allocate an additional 64 bytes on the stack to allow for this. Stack checking code is generated only for leaf functions which require more than that.

15.4.7 Dynamic Memory Allocation - the heap, malloc( ), sbrk( )

malloc( ) allocates memory from a heap managed by function **sbrk( )** in **src/sbrk.c**. There are two ways to create the heap:

- Define **__HEAP_START** and **__HEAP_END**, typically in a linker command file. See the files **conf/default.dld**, **conf/sample.dld**, and *25.6 Command File Structure*, p.383 for examples.
- Recompile **sbrk.c** as follows:

  ```
  dcc -ttarget -c -D SBRK_SIZE=n sbrk.c
  ```

  where *n* is the size of the desired heap in bytes.

The **malloc( )** function implements special features for initializing allocated memory to a given value and for checking the free list on every call to **malloc( )** and **free( )**. See *malloc( )*, p.503.

NOTE: To avoid excess execution overhead, **malloc( )** acquires heap space in 8KB master blocks and sub-allocates within each block as required, re-using space within each 8KB block when individual allocations are freed. The default 8KB master block size may be too large on systems with small RAM. To change this, call

```
size_t __malloc_set_block_size(size_t blocksz)
```

where *blocksz* is a power of two.

NOTE: **malloc( )** and related functions must be initialized by function **__init( )** in **crtlibso.c**. See the note at the end of the section *15.4.3 Notes for crtlibso.c and ctordtor.c*, p.256 for details.

15.4.8 Run-time Initialization and Termination

The compiler automatically generates calls to initialization and finalization functions, including C++ global constructors, through pointers in each module's **.ctors** and **.dtors** sections. Initialization and finalization functions can appear in any program module and are identified by the **constructor** and **destructor** attributes, respectively. Functions identified with the **constructor** and **destructor** attributes are executed when **__init( )** and **__fini( )** are called, as shown in Figure 15-1 and described in *15.4.3 Notes for crtlibso.c and ctordtor.c*, p.256.

NOTE: An archived object file containing constructors or destructors will not be pulled from its **.a** file and linked into the final executable unless it also contains at least one function that is explicitly called by the application. To ensure execution of startup and termination code, never create modules that contain only constructor and destructor functions.

The priority of initialization and finalization functions can be set through arguments to the **constructor** and **destructor** attributes; functions with *lower* priority numbers execute first. For each priority level assigned, the compiler creates a subsection called **.ctors.***nnnnn* **or .dtors.***nnnnn***,** where *nnnnn* is a five-digit numeral between 00000 and 65535; the *higher* the value of *nnnnn*, the earlier the functions in that section are called. For example, a function declared with **__attribute__ ((constructor(12)))** will be referenced in **.ctors.65523** (because 65523=65535–12). All of the **.ctors.***nnnnn* sections are grouped at link time into a single section called **.ctors**, and all of the **.dtors.***nnnnn* sections are grouped at link time into a single section called **.dtors**. For an example linker map, see **ctordtor.c.**

By default, user-defined initialization and finalization functions (as well as global class constructors) have the last priority, to ensure that compiler-defined initialization and finalization occurs first.

For more information on **constructor** and **destructor** attributes, see *constructor, constructor(n) Attribute*, p.141 and *destructor, destructor(n) Attribute*, p.142. To change the default priority for initialization and finalization functions, see *5.4.70 Control Default Priority for Initialization and Finalization Sections (-Xinit-section-default-pri)*, p.84.

Old-style Initialization and Termination

For backward compatibility, the compiler supports an older style of run-time initialization and termination that uses **.init$***nn* and **.fini$***nn* sections (instead of **.ctors** and **.dtors**). To use old-style initialization and finalization, enable **-Xinit-section=2** (see *5.4.69 Control Generation of Initialization and Finalization Sections (-Xinit-section)*, p.83). In this mode, the compiler also supports the use of special **_STI__***nn*_ and **_STD__***nn*_ prefixes (as well as **constructor** and **destructor** attributes) to identify initialization and finalization functions and set their priority. In cases where both **.init$***nn* and **.ctors** sections are present, the default **__init(**) function executes the code in **.ctors** first; similarly, in cases where both **.fini$***nn* and **.dtors** sections are present, the default **__fini(**) function executes the code in **.dtors** first.

15.5 Hardware Exception Handling

- Please refer to the *ARM Architecture Reference Manual* for a description of the exception (interrupt) handling by the hardware.

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.

15.6 Library Exception Handling

On error, many standard library functions set **errno** and return a null or undefined value as described for each function in *34. C Library Functions*. This is typical of, for example, file system functions.

Many math functions, **malloc( )**, and some other library functions call a central error reporting function (in addition to setting **errno**):

```
__diab_lib_error(int fildes, char *buf, unsigned nbyte);
```

where:

fildes
File descriptor index: 1 for **stdout**, 2 for **stderr** (the usual value for error reports).

buf
Buffer containing an ASCII string describing the error, e.g., "**stack overflow**".

nbyte
Number of characters in *buf* (excluding any terminating null byte).

__diab_lib_error( ) is defined in **src/lib_err.c** and may be modified as required. (The prototype for **__diab_lib_error( )** is not included in any user accessible header file; the prototype given above may be added to a user header file if it is desirable to call **__diab_lib_error( )** from user application code.) Unless the message is intercepted by another program, **__diab_lib_error( )** writes the

message to the file given by *fildes* and returns the number of bytes written. After calling **__diab_lib_error( )**, most functions continue execution (after setting **errno** if required).

15.7 Linker Command File

A linker command file:

- Can specify input files and options, although usually these are on the command line.
- Specifies how memory is configured.
- Specifies how to combine the input sections into output sections.
- Assigns addresses to symbols.

See *25. Linker Command Language* for more information about the command language, and *25.6 Command File Structure*, p.383 for an example.

When invoking a compiler driver such as **dcc**, specify a non-default linker command file using the **-Wm** option:

-Wm*pathname*

where *pathname* is the full name of the file. To use the same linker command file for all compilations, specify this option in the **user.conf** configuration file.

If no **-Wm** option is used, the linker will use file *version_path***/conf/default.dld**. Documentary comments are included in this file; please see it for details. See *5.3.28 Specify Linker Command File (-W mfile)*, p.43 for additional details on the **-W m** option.

Other linker command files written for some specific targets are also provided in the **conf** directory. These and **default.dld** may serve as examples for creating your own linker command file.

15.8 Operating System Calls

The source files available in the **src** directory implement or provide stubs for a number of POSIX/UNIX functions for an embedded environment. A partial set is documented in the subsections of this section. Examine the **.c** files to see the complete set.

The modules in the **src** directory are typically stubs which must be modified for a particular embedded environment. These modules have been compiled and the objects collected into two libraries:

libchar.a — basic operating systems functions using simple character input/output

libram.a — basic operating system functions using RAM-disk file input/output.

Variants of these libraries for different object module formats are found in the directories documented in Table 2-2.

To use these functions:

- Modify the above files or those such as **chario.c** discussed below. That is, replace the stub code with code which implements each required function using the facilities available in the embedded environment.
- Compile the files; the script **compile** can be used as is or modified to do this.
- Use **dar** to modify either the original or a copy of **libchar.a** or **libram.a** as appropriate, or simply include the modified object files in your link before the libraries. See *27. D-AR Archiver* for instructions.
- If a copy of **libchar.a** or **libram.a** was modified, see *32.2 Library Structure*, p.444 for a detailed description of how the libraries are structured and searched.

15.8.1 Character I/O

The predefined files **stdin**, **stdout**, and **stderr** use the **__inchar( )/__outchar( )** functions in *version_path***/src/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 predefined and mapped to these character I/O functions.

chario.c can be compiled for supported boards and simulators by defining one of several preprocessor macros when compiling **chario.c**. These macros are:

SingleStep debugger	SINGLESTEP
I.D.P. M68EC0x0 board	IDP
SB306 board	SBC306
EST Virtual Emulator	EST
MBUG monitor for 68k boards	MBUG

For example, all versions of **chario.o** in the supplied libraries are compiled for SingleStep as follows:

```
dcc -c -DSINGLESTEP chario.c
```

These preprocessor macros typically cause the inclusion of code which reads from or writes to devices on the board, or make system calls for doing so, or in the case of SingleStep, supports input/output to the SingleStep command window.

chario.c has three higher level functions:

- **inedit( )** corresponds to **stdin**; it reads a character by calling **__inchar( )** and calls **outedit( )** to echo the character.
- **outedit(...)** corresponds to **stdout**; it writes a character by calling **__outchar( )**.
- **outerror(...)** corresponds to **stderr**; it writes a character by calling **__outerrorchar( )**. This function is currently used only by SingleStep (when compiling **chario.c** with **-DSINGLESTEP**); other implementations write **stderr** output to **stdout**.

The lower level functions, **__inchar( )**, **__outchar( )**, and **__outerrorchar( )** implement the actual details of input/output for each of the boards for emulators listed above. Examine the code for details.

See the makefiles in the example directories (*version_path*/**example/**...) for suggestions on recompiling **chario.c** for the selected target board.

15.8.2 File I/O

A number of standard file I/O functions are implemented as a "RAM-disk". These functions are part of the standard **libc.a** library when **cross** is used as part of a **-t***tof*:**cross** option when linking (see Table 4-1).

For a convenient way to create RAM-disk files for use with these functions, see *15.11 Target Program Arguments, Environment Variables, and Predefined Files*, p.270.

Space required by the file I/O functions is allocated by calls to **malloc( )**.

The following functions are supported. For details on any of these functions, including header files containing their prototypes, lookup the function in *34. C Library Functions*.

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 filenames 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.

unlink( )
In **unlink.c**, removes a file from the file system.

write( )
In **write.c**, writes a buffer to a file.

15.8.3 Miscellaneous Functions

The following functions provide miscellaneous services.

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.

__diab_lib_err( )
In **lib_err.c**, reports errors caught by library functions. See *15.6 Library Exception Handling*, p.262.

_exit( )
In **_exit.c**, closes all open files and halts. See *15.4.5 Notes for Exit Functions*, p.258.

getpid( )
In **getpid.c**, returns a process number. Modify this if you have a multiprocessing system.

__init_main( )
In **init.c**, is called from the startup code and performs some initializations. See *15.4.4 Notes for init.c*, p.257.

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.

15.9 Communicating with the Hardware

The following features facilitate access to the hardware in an embedded environment.

15.9.1 Mixing C and Assembler Functions

The calling conventions of the compiler are well defined, and it is straightforward to call C functions from assembler and vice versa. See *9. Calling Conventions* 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.

15.9.2 Embedding Assembler Code

Use the **asm** keyword or direct functions to intermix assembler instructions in the compiler function. See *7. Embedding Assembly Code* for details.

15.9.3 Accessing Variables and Functions at Specific Addresses

There are four ways to place a variable or function at a specific 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 *Using the Address Clause to Locate Variables and Functions at Absolute Addresses*, p.243.

   Advantages of using absolute sections:

   - I/O registers, global system variables, and interrupt vectors and functions can be placed at the correct address from the program without the need to write a complex linker command file.
   - Absolute variables will have all symbolic information needed by symbolic debuggers. Variables defined using the linker command language cannot be debugged at a high level.

   Examples using absolute addressing at compile-time:

```
// 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 linker 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 order as given in the source program, without the optimizer eliminating any of the accesses.

4. By placing the variables or functions in a special named section during compilation and then locating the section via a linker command file.

See *25. Linker Command Language* for additional details.

15.10 Reentrant and "Thread-Safe" Library Functions

Most library functions are reentrant, although in some cases this is impossible because the functions are by definition not reentrant. In *34. C Library Functions*, the "Reference" portion of each function description includes "REENT" for completely reentrant functions and "REERR" for functions which are reentrant except that **errno** may be set. Functions not so marked are not reentrant. In some cases, standard functions are supplied in special reentrant versions, and functions that modify only **errno** can be made completely reentrant by modifying the **__errno_fn( )** function. See *34. C Library Functions*, for more information.

The reentrant functions are "thread-safe"—that is, they work in a multi-threaded or multitasking environment. Notable exceptions include **malloc( )** and **free( )**. Typically, real-time operating systems include thread-safe versions of these functions. You can also create thread-safe versions of **malloc( )** and **free( )** by implementing the functions **__diab_alloc_mutex( )**, **__diab_lock_mutex( )**, and **__diab_unlock_mutex( )**; these three functions are called by **malloc( )** (see **malloc.c** for their usage) but, as shipped, do nothing.

15.11 Target Program Arguments, Environment Variables, and Predefined Files

In a host-based execution environment, a program can be started with command-line arguments and can access environment variables and a file system.

The **setup** feature brings the same capabilities to programs running in an embedded environment without the need for an operating system or file devices.

Being able to pre-define arguments, environment variables, and files means:

- When porting an existing host-based program (e.g., a test program or benchmark), it may be possible to compile and run the program with little or no modification.
- A program can read large amounts of test or constant data from a "RAM-disk" file using the input/output functions described in *15.8.2 File I/O*, p.265.

The **setup** program provides initial values for arguments, environment variables, and RAM-disk files as follows:

- You run **setup** on your host system, giving it options which provide values for target-based "command-line options" and "environment variables" and which name host files.
- **setup** writes a file on your host system called **memfile.c**. The data for the arguments and environment variables and from the host files is included in **memfile.c**.
- You then treat **memfile.c** as part of your application: include it as a normal **.c** file in your makefile in order to compile and link it with your application.
- When you run your application on your target, the code in **memfile.c** and associated library functions will provide the data for the **argc** and **argv** arguments to **main**, for environment variables accessible through **getenv** calls, and for RAM-disk files. (See *15.4 Startup and Termination Code*, p.254 for related details.)

setup is run as follows:

`setup` [`-a` *arg*] [`-e` *evar*[`=`*value*`]`] [`-b` *file*`]` [`-t` *file*] ...

where the options are:

`-a` *arg*
Increments **argc** by one and adds *arg* to the strings accessible through **argv** passed to **main** in the usual way. The program name pointed to by **argv[0]** will always be "**a.out**".

`-e` *evar*[*=value*]
Creates an environment variable accessible through **getenv( )** in the usual way: **getenv** ("*name*") will return a null-pointer if *name* does not match any *evar* defined by **-e**, will return an empty string if there is a match but no *value* was provided, or will return "*value*" as a string.

`-b` *filename*
The contents of the given host file will be a binary file accessible as a RAM-disk file with the given name. (Any path prefix will be included in the *filename* exactly as given.)

`-t` *filename*
The contents of the host file will be a text file accessible as a RAM-disk file with the given name. (Any path prefix will be included in the *filename* exactly as given.)

Any combination and number of the different options are allowed. Invoking **setup** with no arguments will display a usage message.

Example

If you run **setup** as follows:

```
setup -a -f -a db.dat -e DEBUG=2 -b db.dat -t f1.asc
```

it will write **memfile.c** in the current directory.

When **memfile.c** is compiled and included in your application:

- The application's **main** function will act as if the application had been started with the command line:

  ```
  a.out -f db.dat
  ```

- The environment variable **DEBUG** will be set to "**2**" so that **getenv("DEBUG")** will return "**2**"".
- Binary file **db.dat** will be predefined and can be opened with **fopen( )** or **open( )** library calls.
- ASCII text file **f1.asc** will be predefined and can be opened as above.

setup is an ANSI standard C program supplied in source form as **setup.c** in the **src** directory. To use it, first compile and link it with any native ANSI C tools on your host system. Typically, it will be sufficient to change to the tools' **src** directory, enter the following command (assuming **cc** invokes an ANSI C compiler):

```
cc -o setup setup.c
```

and then move the executable file **setup** to your tools' **bin** directory or some other directory in your path.

15.12 Profiling in An Embedded Environment

Profiling collects information while your program executes. That information is then fed back to the compiler for more optimal code generation based on what your program actually does when it executes.

The compiler implements profiling through the **-Xblock-count** and **-Xfeedback** options. There are three main steps:

- Compile your code with **-Xblock-count** to insert counting code.

- Run your program; count data will be written as your program runs. Transfer the count data from the target to your host.
- Re-compile your code with **-Xfeedback** — the compiler will optimize based on the count data.

In more detail:

- Compile all modules to be profiled with the **-Xblock-count** option, e.g.:

  ```
  dcc -c -Xblock-count file1.c file2.c
  ```

  This causes the compiler to insert minimal *profiling code* to track the number of times each basic block is executed (a *basic block* is the code between labels and branches).

  This *profile data* is written by the profiling code to a target file named **dbcnt.out**. Thus, you must either have an environment in which target files may be connected to files on your host, or you may use the RAM-disk service (see *15.8.2 File I/O*, p.265).

- Copy library module *version_path***/src/_exit.c** and modify it to write the profiling data back to your host system. For example, if you used the RAM-disk feature, copy the data in target file **dbcnt.out** to **stdout** and collect the data into an ASCII file. The distributed **_exit.c** includes code to do this conditioned by two macros: PROFILING and RAMDISK. To use this code without further modification to **_exit.c**, recompile with:

  ```
  dcc -c -DPROFILING -DRAMDISK version_path/src/_exit.c
  ```

  See **_exit.c** for additional details.

- Compile the rest of your program and link as usual.
- Execute your program on the target system. When it terminates, it will write the profiling information back to the host system per your modification to **_exit.c**.
- If the profiling information was transferred back to the host in ASCII format, use the **ddump** command to convert it to a binary file (the **dbcnt.out** output filename is chosen because it is the default for the step after this).

  ```
  ddump -B -o dbcnt.out your-file-of-collected-profile-data
  ```

- Recompile the modules profiled with the **-Xfeedback** option:

  ```
  dcc -c -Xfeedback -XO file1.c file2.c
  ```

  (use **-Xfeedback=***profile-file*, where *profile-file* is the name of file of collected profile data in binary form if that file is not named **dbcnt.out**).

The compiler will optimize based on the profile data collected from the target. Make sure to use the **-XO** option as well to get the best code (either **-XO** or **-O** must be included or the profile data will be ignored).

PART III

Wind River Assembler

16 The Wind River Assembler

This chapter describes the assembler for ARM microprocessors. For in-depth information on the ARM architecture and instructions, please refer to the manufacturer's documentation.

16.1 Selecting the Target

The target for the assembler is selected by the same methods as for the compiler. See *4.1 Selecting a Target*, p.21 for details. When using the compiler drivers **dcc**, **dplus**, etc., the target for the assembler is selected automatically by the driver.

16.2 The das Command

The command to execute the assembler is as follows:

das [*options*] [*input-files*]

where:

das
Invokes the assembler.

options
Command-line options; see the following subsection for details. Options must precede the input files.

input-files
A list of filenames, paths permitted, separated by whitespace, naming the file(s) to be assembled; the default suffix is **.s**.

The assembler assembles the input file and generates an object file as determined by the selected target configuration. By default, the output file has the name of the input file with an extension suffix of **.o**. The **-o** option can be used to change the output filename.

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*.

Example: assemble **test.s** with a symbol named **DEBUG** equal to 2 for use in conditional assembly statements:

```
das -D DEBUG=2 test.s
```

16.3 Assembler Command-Line Options

The following command-line options are available. Also see the next section, *16.4 Assembler -X Options*, p.283.

NOTE: 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 following 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, the last instance is used.

Show Option Summary (-?)

-?, -h,
--help
Show synopsis of command-line options.

Define Symbol Name (-Dname=value)

-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 expression*, p.320 for more information.

Generate Debugging Information (-g)

-g
Generate debug line and file information. (ELF/DWARF format only). Equivalent to **-Xasm-debug-on**.

Include Header in Listing (-H)

-H
Print a header on the first line of each page of the assembly listing. See *Include Header in Listing (-Xheader...)*, p.284 for additional details and *22. Example Assembler Listing* for an example of an assembly listing.

Set Header Files Directory (-I path)

-I *path*
Specify a directory where the assembler will look for header files. May be given more than once. See the *.include "file"*, p.322 for more information.

Generate Listing File (-l, -L)

-l
Generate the listing file to *input-file*.**lst**. (To change the default extension of the output file, use **-Xlist-file-extension=**"*string*"; for example, **-Xlist-file-extension=".L"**.)

-L
Generate the listing file to standard output. See *22. Example Assembler Listing* for an example of an assembly listing.

Set outpUt File (-o file)

-o *file*
Write the object file to *file* instead of the default (*input-file*.s). Applies only to the first file if a list of files is presented; remaining files in the list use the default.

Remove the Input File on Termination (-R)

-R
May be used by tools to remove temporary files.

Specify Assembler Description (.ad) File (-T ad-file)

-T *ad-file*
Specify which assembler description (**.ad**) file to use. This is normally set automatically by using the **-t** option, defining the **DTARGET** and the **DOBJECT** environment variables, or using the **-WDDTARGET** and the **-WDDOBJECT** command-line options. It is primarily for internal use by Wind River.

Select Target (-ttof:environ)

-t*tof:environ*

Specifies with one command the **DTARGET** (*t*), the **DOBJECT** (*o*), the **DFP** (*f*), and the **DENVIRON** (*environ*) configuration variables. See *4. Selecting a Target and Its Components* for details.

Print Version Number (-V)

-v

Display the version number of the assembler on standard output.

Define Configuration Variable (-WDname=value)

-WD*name=value*

Set a configuration variable for use in the configuration files with the given *name* to the given *value*. Overrides an environment variable of the same name.

Select Object Format and Mnemonic Type (-WDDOBJECT=object-format)

-WDDOBJECT=*object*

Specify the object format and mnemonic type. Overrides the environment variable **DOBJECT** if it is also set.

Select Target Processor (-WDDTARGET=target)

-WDDTARGET=*target*

Specify the target processor. Overrides the environment variable **DTARGET** if it is also set.

Discard All Local Symbols (-x)

-x

Discard symbols not declared **.extern** or **.comm**.

Discard All Symbols Starting With .L (-X)

-x

Discard all symbols starting with **.L**; supports compilers using this form for automatically generated symbols, including the Wind River compiler.

Print Command-Line Options on Standard Output (-#)

-#

The output of this option can be directed to a file. This can be convenient when contacting Technical Services. The **-#** should immediately follow the **das** command (after a space).

Read Command-Line Options from File or Variable (-@name, -@@name)

-@*name*

Read command-line options from either a file or an environment variable. When -@name is encountered on the command line, the assembler first looks for an environment variable with the given name and substitutes its value. If an environment variable is not found then it 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 the assembler terminates.

-@@*name*

Same as *-@name*; also prints all command-line options on standard output.

Redirect Output (-@E=file, -@E+file, -@O=file, -@O+file)

-@E=*file*
-@E+*file*

Redirect any output to standard error to the given file.

-@O=*file*
-@O+*file*

Redirect any output to standard output to the given file.

In both cases, use of **+** instead of **=** appends the output to the file.

16.4 Assembler -X Options

The following options provide more detailed control of the assembler. The **-X** options are for use on the command line; **-X** options can also be set using the **.xopt** assembler directive. See *.xopt*, p.333.

Specify Value to Fill Gaps Left by .align or .alignn Directive (-Xalign-fill-text)

-Xalign-fill-text=*n*
Fill gaps left by the **.align** or **.alignn** directive with the value *n*, overriding the processor-specific default.

Interpret .align Directive (-Xalign-value, -Xalign-power2)

-Xalign-value
Interpret the value in an **.align** directive as the value to which the location counter is to be aligned, which must be a power of 2. Example: **-Xalign-value=8** means **.align** is to align on an 8-byte boundary.

-Xalign-power2
Interpret the value in an **.align** directive as the power of 2 to which the location counter is to be aligned. Example: **-Xalign-power2=3** means **.align** is to align on an 8-byte boundary

This is the default.

Generate Debugging Information (-Xasm-debug-...)

-Xasm-debug-off
Do not generate debug line and file information. This is the default.

-Xasm-debug-on
Generate debug line and file information. (ELF/DWARF format only).

Align Program Data Automatically Based on Size (-Xauto-align)

-Xauto-align-off
The assembler performs no data alignment. This is the default.

-Xauto-align
Align program data automatically based on size.

Set Instruction Type (-Xcpu-...)

-Xcpu-*target*
Accept instructions only for the target processor designated by *target*. This option is primarily for internal use and is set automatically by the driver in response to the user-level **-t***tof***:***environ* option. See Table 4-1 for details.

Set Default Value for Section Alignment (-Xdefault-align)

-Xdefault-align=*value*
Set the value use when calculating the default alignment for **.comm**, **.lcomm**, and **.sbss** directives, and the alignment used by the **.even** directive.

The default value of **-Xdefault-align** is 8 if no value is given.

Absent this directive, the default alignment for ELF sections is the maximum alignment of all objects in the section.

Note that for ELF modules, **-Xdefault-align** does not set the alignment of sections — it sets the default for used by the **.comm**, **.lcomm**, **.sbss**, and **.even** directives. Only if one of these directives is in fact used in a section will the alignment be as set by **-Xdefault-align** rather than the maximum alignment of all objects in the section.

Enable Local GNU Labels (-Xgnu-locals-...)

-Xgnu-locals-off
Disable local GNU labels. See *GNU-Style Locals*, p.297 for more information. The default setting is **-Xgnu-locals-on**.

-Xgnu-locals-on
Enable local GNU labels. See *GNU-Style Locals*, p.297 for more information. This is the default.

Include Header in Listing (-Xheader...)

-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** below 31.

-Xheader-off
Do not include a header in the listing file. This is the default.

See *22. Example Assembler Listing* for an example of an assembly listing.

Set Header Format (-Xheader-format="string")

-Xheader-format="*string*"
Define the format of the header in the assembly listing. (The header is enabled by options **-H** or **-Xheader** above). The header *string* can contain format specifications in any order introduced by a "**%**". Characters not preceded by "**%**" are printed as is, including spaces and escapes such as "**\t**" for tab.

Valid format specifications are:

%*n*E
Use *n* columns to display the error count.

%*n*F
Use *n* columns to display the filename.

%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"
```

See *22. Example Assembler Listing* for an example of an assembly file listing.

Set Label Definition Syntax (-Xlabel-colon...)

-Xlabel-colon
Require that all label definitions have a colon "**:**"appended. When this option is selected, some directives are allowed to start the line. This is the default.

Note that this applies to all directives, including **.equ** and **.set**. Thus, with this option:

```
TRUE:  .set  1          valid
TRUE   .set  1          invalid
```

-Xlabel-colon-off
Do not require label definitions to end with a colon "**:**". When this option is selected, directives are not allowed to start in column 1.

Set Format of Assembly Line in Listing (-Xline-format="string")

-Xline-format="*string*"
Define the format of each assembly line in a listing. The *string* can contain the following format specifications, in any order, starting with a "**%**". Characters not preceded by "**%**" are printed as is, including spaces and escapes such as "**\t**" for tab.

Valid format specifications are:

%*n***A**
Use *n* columns to display current address.

%*n***.mC**
Use *n* columns to display the generated code. A space is inserted at every *nth* column.

%*n***D**
Display a maximum of *n* generated bytes for each source line. *n* may have a value from 1 through 32. 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 Program Location Counter (PLC) 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"
```

See 22. *Example Assembler Listing* for an example of an assembly listing.

Generate a Listing File (-Xlist-...)

-Xlist-file
Generate a listing file to file *input-file*.**lst**. 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.

See *22. Example Assembler Listing* for an example of an assembly listing.

Specify File Extension for Assembly Listing (-Xlist-file-extension="string")

-Xlist-file-extension="*string*"
Use this option to override the default extension (**.lst**) of the listing file generated by **-l** or **-Xlist-file**. For example, **-Xlist-file-extension=".L"** specifies the file extension **.L**.

Set Delay of Literal Generation (-Xlit-marg-...)

-Xlit-marg-hard=*value*
The assembler attempts to delay the generation of literals for as long as possible. This option, along with **-Xlit-marg-soft** and **-Xlit-marg-thresh**, controls that delay. The branch over the literals is only meant to be generated under unusual conditions. Use caution in invoking these options, as unpredictable results may result if the parameters are set to values beyond the capabilities of the target hardware.

Each literal has two addresses:

- *faddr*—the first address where the literal can be generated
- *laddr*—the last address where the literal can be generated

Forced generation of literals occurs when

```
curPC >= faddr && {curPC + hard} > laddr
```

(**curPC** is the current PC.) Use **-Xlit-marg-hard** to define the forced threshold value *hard*. The default value is 16.

-Xlit-marg-soft=*value*
Define a soft (normal) threshold value. This threshold is used when the code contains an unconditional branch or return. The default value is 128.

Normal literal generation occurs when

```
curPC >= faddr && (curPC + soft) > laddr
```

-Xlit-marg-thresh=*value*

Define a threshold value, that is, how long the thread should continue once literal generation has commenced. The default is 256.

Literal generation continues as long as

```
curPC >= faddr && (curPC + thresh) > laddr
```

Set Line Length of Listing File (-Xllen=n)

-Xllen=*n*

Define the number of printable character positions per line of the listing file. The default is 132 characters. A value of 0 means unlimited line length. This value may also be set or changed by the **.llen** (*.llen expression*, p.324) and **.psize** (*.psize page-length [,line-length]*, p.326) directives.

See *22. Example Assembler Listing* for an example of an assembly listing.

Enable Blanks in Macro Arguments (-Xmacro-arg-space-...)

-Xmacro-arg-space-off

Do not permit blanks in macro arguments. This is the default.

-Xmacro-arg-space-on

Permit blanks in macro arguments.

Set Page Break Margin (-Xpage-skip=n)

-Xpage-skip=n

If *n* is zero (the default), page breaks in the listing file will be created using formfeed (ASCII 12). Otherwise each page will be padded with *n* blank lines, and these *n* blank lines included in the count set by **-Xplen** option. See *22. Example Assembler Listing* for an example of an assembly listing.

Set Lines Per Page (-Xplen=n)

-Xplen=*n*

Define the number of printable lines per page in the listing file. The default value of *n* is 60. See also **-Xpage-skip above**. This value may also be set or changed by the **.lcnt** (see *.lcnt expression*, p.323) and **.psize** (see *.psize page-length [,line-length]*, p.326) directives. See *22. Example Assembler Listing* for an example of an assembly listing.

Limit Length of Conditional Branch (-Xprepare-compress=n)

-Xprepare-compress=*n*
Change the maximum length of a conditional branch from the default, which is 32,766 bytes; if *n* is not specified, the length is set to 1024. If a conditional branch exceeds this limit, the assembler inserts a reverse conditional around an unconditional branch to the label.

Treat Semicolons As Statement Separators (-Xsemi-is-newline)

-Xsemi-is-newline
Treat the semicolon (**;**) as a statement separator instead of a comment character. This is useful for GNU compatibility.

Enable Spaces Between Operands (-Xspace-...)

-Xspace-off
Do not allow spaces between operands in an assembly instruction.

-Xspace-on
Allow spaces between operands in an assembly instruction. This is the default.

Delete Local Symbols (-Xstrip-locals..., -Xstrip-temps...)

-Xstrip-locals
Do not include local symbols in the symbol table. This is the same as the **-x** option. Local symbols are those not defined by **.extern** or **.comm**.

-Xstrip-locals-off
Include local symbols 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. This option can be used to suppress the temporary symbols generated by the compiler.

-Xstrip-temps-off
Include local symbols starting with **.L** in the symbol table. This is the default.

Set Subtitle (-Xsubtitle="string")

-Xsubtitle="*string*"
Define a subtitle that will be printed in the **%S** field of the header. See *Set Header Format (-Xheader-format="string")*, p.285, for more information.

Set Tab Size (-Xtab-size=n)

-Xtab-size=n
Define the number of spaces between tab stops. The default is 8.

Set Title (-Xtitle="string")

-Xtitle="*string*"
Define a title that will be printed in the **%T** field of the header. See *Set Header Format (-Xheader-format="string")*, p.285, for more information.

17 Syntax Rules

17.1 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.

If **-Xsemi-is-newline** is enabled, "**;**" (semicolon) can also serve as a statement separator.

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 *Enable Spaces Between Operands (-Xspace-...)*, p.289).

A comment starts with "#" as shown above. However, if using the preprocessor, then use "//" at the start of a line (because "#" will be misinterpreted as a preprocessor directive). See *Comment*, p.294 for additional comment details.

All fields are optional depending on the circumstances. In particular:

- Blank lines are permitted.
- A statement may contain only a *label*.
- The *opcode* must be preceded by a label or whitespace (one or more blanks or tabs). A statement may contain only an *opcode.* (Assembler directives may start in column one but only if the **-Xlabel-colon** option is given.)
- 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 (r2) 4-byte words from src (r1) to dest (r0)
        .globl  mv_word
        .text
mv_word:
        sub     r13,r13,#8
        mov     r3,r2
        cmp     r2,#0
        beq     .L2
        mov     r2,r2,lsl #2
        add     r0,r0,r2
        add     r1,r1,r2
        add     r1,r1,#4
        add     r0,r0,#4
.L4:
        sub     r1,r1,#4
        sub     r0,r0,#4
        ldr     r12,[r1,#0]
        str     r12,[r0,#0]
        subs    r3,r3,#1
        bne     .L4
.L2:
        add     r13,r13,#8
        mov     pc,lr
        nop
```

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 *17.2 Symbols*, p.294 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
4:                  # A local label
```

(See *17.5 Local Symbols*, p.296 for details on local labels.)

Opcode

The opcode of an assembly language statement identifies the statement as either a machine instruction or an assembler directive.

The opcode must be preceded by a label or whitespace (one or more blanks or tabs). 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, at least one blank is 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. Assembler directives may start in column one but only if the **-Xlabel-colon** option is given.

The assembler is case-insensitive regarding opcodes.

Operand Field

In general, an operand field consists of 0-3 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 are semicolon "**;**", at sign "@", pound sign "**#**". (To treat the semicolon as a statement separator, use **-Xsemi-is-newline**.)

In addition, the C++ comment marker "**//**" is valid at the start of a line and must be used there if pre-processing to select ARM vs. Thumb instructions (because the standard comment delimiter, "**#**", will be taken misinterpreted as a preprocessor directive if it begins a line). On the other hand, use "**#**" to begin a comment which is not the first text on the line (because "**//**" is invalid in that case).

An asterisk "*" in column 1 is also treated as a comment delimiter.

The comment field consists of all characters in a source line including and following the comment character through the end of the line (the next <Newline> character). These characters are ignored by the assembler.

17.2 Symbols

A symbol consists of a number of characters, with the following restrictions:

- Valid characters include A-Z, a-z, 0-9, period "**.**", dollar sign "**$**", and underscore "_".
- The first character must not be a "**$**" dollar sign.
- The first character must not be numeric except for local symbols (*17.5 Local Symbols*, p.296).

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 *symbol[:] .set expression*, p.329).

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. If a common symbol is not defined in any module, it will be allocated by the linker to the end of the **.bss** section. See *23.4 COMMON Sections*, p.352 for additional details.

17.3 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
```


17.4 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**, or **.export** directives.
- 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:
        mov     r2,#2           # return 2
        mov     pc,lr
```

External symbols are only declared to the assembler by the **.globl**, **.global**, **.xdef**, or **.export** directives. 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, which may be located in a different file, use the above defined labels:

```
        b       two
        ldr     r0,=table
```

Note that whenever a symbol is used that is not defined in the same file, it is considered to be a global undefined symbol by the assembler.

An external symbol is also declared by the **.comm** directive in one or more modules (see *.comm symbol, size [,alignment]*, p.315). For the rest of the assembly such a symbol, called a common symbol, will be treated as though it is an undefined global symbol. The assembler does not allocate storage for common symbols; this task is left to the linker. The linker computes the maximum size of each common symbol with the same name, allocates storage for it at the end of the final **.bss** section, and resolves linkages to it (unless the **-Xbss-common-off** is used; see *5.4.15 Control Allocation of Uninitialized Variables in "COMMON" and bss Sections (-Xbss-off, -Xbss-common-off)*, p.62).

17.5 Local Symbols

Local symbols provide a convenient way of generating labels for branch instructions. Use of local symbols reduces the possibility of attempting to define a symbol more than once 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. The assembler implements two styles of local symbols.

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 consists of one to five digits when defined. A GNU-style local symbol is referenced by the digits followed by the character **f** or **b**. When the digits are suffixed by an **f**, the nearest definition going forward (toward the end of the source) is referenced. When suffixed with the character **b**, the nearest definition going backward (toward the beginning of the file) is referenced. Example:

```
15:
        .long   15f      # Reference definition below.
        .long   15b      # Reference definition above.
15:
```

By default the GNU style local symbols are recognized by the assembler. This can be disabled with the option **-Xgnu-locals-off** (see *Enable Local GNU Labels (-Xgnu-locals-...)*, p.284).

17.6 Constants

The assembler supports both integral and floating point constants. Integral constants may be entered in decimal, octal, binary or hexadecimal form, or they may be entered as character constants. Floating point constants can only be used with the **.float** and **.double** directives.

Integral Constants

Internally, the assembler treats all integer constants as signed 32-bit binary two's complement quantities. Valid constant forms are listed below. The order of the list is significant in that it is scanned from top to bottom, and the first matching form is used.

'*c***'**
character constant

0x*hex-digits*
hexadecimal constant

0*octal-digits*
octal constant

/hex-digits
hexadecimal constant

@*octal-digits*
octal constant

%*binary-digits*
binary constant

decimal-digits
decimal constant

*octal-digits***o**
octal constant

*octal-digits***q**
octal constant

*binary-digits***b**
binary constant

Examples:

```
abc  =  12         12 decimal
bcd  =  012        12 octal (10 decimal)
cde  =  0x12       12 hex (18 decimal)
```

To represent special character constants, use the following escape sequences:

Constant	Value	Meaning
'\b'	8	backspace
'\t'	9	horizontal tab
'\n'	10	line feed (newline)
'\v'	11	vertical tab
'\f'	12	form feed
'\r'	13	return
'"'	39	single quote
'\\'	92	backslash

By using a **"***nnn***"** construct, where *nnn* is an octal value, any character can be specified:

```
'\101'   same as 'A (65 decimal)
'\60'    same as '0 (48 decimal)
```


Floating Point Constants

Floating point constants have the following format:

[+ | -]*i.i*{**e** | **E**}**[+ | -]***i*

where *i* is an integer. All parts are optional as long as the constant starts with a sign or a digit and contains either a decimal point or an exponent (**e** or **E** and a following digit). Also, +**NAN** and [+/-]**INF** are supported. Examples:

```
float   1.2, -3.14, 0.27172e1
double  -123e-45, .56, 1e23
```

String Constants

The form of a string is:

"*characters*"

where *characters* is one or more printable characters or escape codes.

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	Enter
\"	34	Double quote ""
****	92	Backslash "\"
nnn	*nnn* (octal)	Octal value of *nnn*

Some examples follow. The final two are equivalent.

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
`.ascii "Warning-",7,7,"\n"`	Same as previous line.

18

Sections and Location Counters

18.1 Program Sections

Assembly language programs are usually divided into sections to separate executable code from data, constant data from variable data, initialized data from uninitialized data, etc. Some important predefined sections are described below, with a reference to the assembler directive that switches output to each section.

.text, p.331
Instruction space.

.data, p.316
Initialized data.

.bss, p.314
Uninitialized data.

.rodata, p.326
Read-only data.

By invoking these directives, it is possible to switch among the sections of the assembly language program. New sections can also be defined with the **.section** directive (see *.section name, [alignment], [type]*, p.327).

The assembler maintains a separate location counter for each section. Thus for assembly code such as:

```
.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*.

ELF sections are aligned based on their contents or on a specified alignment in a **.section** directive. ELF sections are not extended to any boundary whether aligned or not.

Padding introduced into a code section (but not other types of sections) by means of an **.align** or **.alignn** directive is zero-filled (the nop instruction).

NOTE: See the **-f** linker option, *24. The dld Command*, for filling of gaps between input sections in an output section.

18.2 Location Counters

The assembly current location counter is represented by the character "**.**". In the operand field of any statement or assembly directive it represents the address of the first byte of the statement.

NOTE: A current location counter appearing as an operand in a **.byte** directive (see *.byte expression ,...*, p.314) always has the value of the address at which the first byte was loaded; it is not updated while evaluating the directive.

The assembler initializes 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 the assembler 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.***xxxxxxxx*, where *xxxxxxxx* is the hexadecimal address of the section, with leading zeros to fill to eight digits. The linker will then place this section at the specified address. For example:

```
. = 0xff0000
```

will create a section named **.abs.00ff0000** located at that address.

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.

19

Assembler 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 8 or 16 bits, the least significant 8 or 16 bits are used.

A *term* is a component of an expression. A *term* may be one of the following:

- A constant.
- A symbol.
- An expression or *term* enclosed in parentheses **( )**. Any quantity enclosed in parentheses 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)**.

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 directive, 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 *F. Object and Executable File Format* together with the relocation type used. The following demonstrates the use of relocatable expressions, where “alpha” and ”beta” are symbols:

```
alpha
    relocatable
```

`alpha+5`
relocatable

`alpha-0xa`
relocatable

`alpha*2`
not relocatable (error)

`2-alpha`
not 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

NOTE: In the following tables, the phrase "**expr** evaluates to ... offset from the ... base register" (or similar) means that the assembler generates a constant which is adjusted as necessary by the linker so that the final value in memory is an offset from the designated base register. These constructs are used for position-independent code or data. To execute correctly, the designated base register must be loaded with the base of the code or data area as appropriate. See the discussions of these topics in *14. Locating Code and Data, Addressing, Access*.

Unary Operators

The *unary* operators recognized by the assembler are:

%endof(*section-name***)**
Address of the end of the given section. Evaluates to **.endof.***section_name*, a symbol created by the linker. (See *23.2 Symbols Created By the Linker*, p.350.)

*expr***@data**
expr evaluates to a 32 bit offset from the data base register (**r9**), typically initialized to **__SDA_BASE_**.

*expr***@h**
The most significant 16 bits of *expr* are extracted. Provided for use in assembly language, but not generated by the compiler.

*expr***@ha**
High adjust: The most significant 16 bits of *expr* are extracted and adjusted for the sign of the least significant 16 bits of *expr*. See *High adjust operator*, p.307 below. Provided for use in assembly language, but not generated by the compiler.

expr@l
The least significant 16 bits of *expr* are extracted.

High adjust operator, p.307

%sizeof(*section-name***)**

Size of the given section. Evaluates to **.sizeof.***section_name*, a symbol created by the linker (see *23.2 Symbols Created By the Linker*, p.350).

%startof(*section-name***)**

Address of the start of the given section. Evaluates to **.startof.***section_name*, a symbol created by the linker (see *23.2 Symbols Created By the Linker*, p.350).

+
Unary add.

-
Negate.

~
Complement.

High adjust operator

Sometimes the compiler (or a hand-coded assembly language program, if not the compiler) uses two instructions to copy an address to or from a location in memory. Each instruction can include 16 bits of the address as an immediate value, and the two 16-bit parts of the address are added to form the full address.

For the purposes of this discussion:

- The first instruction has the higher 16 bits of the address.
- The second instruction has the lower 16 bits of the address.

In some cases, the second instruction sign extends the low 16 bits (for example, 0x8000 is sign-extended to 0xffff8000). If so, the first instruction must compensate so that the correct address is calculated when the two parts of the address are added.

How the first instruction compensates depends on the most significant bit of the lower 16 bits of the address:

- If it is zero, no adjustment is made.

- If it is 1, the first instruction adds 1 to the higher 16 bits of the address. The second instruction adds 0xffff, which is equivalent to -1. Thus, the two additions negate each other.

Binary Operators

The *binary* operators recognized by the assembler are:

Binary Operator	Description
+	add
-	subtract
*	multiply
/	divide
|	bitwise or
%	modulo
&	bitwise and
^	bitwise exclusive or
<<	shift left
>>	shift right
==	equal to
!=	not equal to
<=	less than or equal to
<	less than
>=	greater than or equal to
>	greater than

Operator Precedence

Expressions are evaluated with the following precedence in order from highest to lowest. All operators in each row have the same precedence.

Table 19-1 **Assembler Operator Precedence and Associativity**

Operator	Associativity
unary + − ~	right to left
@code @data **@h @ha** **@l** **%startof %endof %sizeof**	left to right
* / % (modulo)	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

20 Assembler Directives

20.1 Introduction

All the assembler directives (or just "directives") described here that are prefixed with a period "**.**" are also available without the period. Most are shown with a "**.**" except for those traditionally written without it.

If the **-Xlabel-colon** option is given (see *Set Label Definition Syntax (-Xlabel-colon...)*, p.285), then directives which cannot take a label may start in column 1. A directive which can take a label—that is, 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 *Enable Spaces Between Operands (-Xspace-...)*, p.289).

In addition to the directives documented in this chapter, the assembler recognizes the following directives generated by some compilers for symbolic debugging:

.d1_line_start, **.d1_line_end**, **.d1file**, **.d1line**, **.def**, **.endef**, **.ln**, .**dim**, **.line**, **.scl**, **.size**, **.tag**, **.type**, **.val**, **.d2line**, **.d2file**, **.d2_line_start**, **.d2_line_end**, **.d2string**,

.d2_cfa_offset, **.d2_cfa_register**, **.d2_cfa_offset_list**, **.d2_cfa_same_value_list**, **.d2_cfa_same_value**, **.uleb128**, **.sleb128**

The remainder of this chapter describes individual assembler directives.

20.2 List of Directives

symbol[:] = expression

See *symbol[:] .equ expression*, p.318. See **-Xlabel-colon-...** in *Set Label Definition Syntax (-Xlabel-colon...)*, p.285 regarding the initial colon.

symbol[:] =: expression

Equivalent to *symbol* **=** *expression* except that *symbol* will be made a global symbol. See **-Xlabel-colon-...** in *Set Label Definition Syntax (-Xlabel-colon...)*, p.285 regarding the initial colon.

.2byte

This is a synonym for **.short** (*.short expression ,...*, p.330) except that there are no alignment restrictions and an unaligned relocation type will be generated if required by the target.

.4byte

This is a synonym for **.long** (*.long expression ,...*, p.324) except that there are no alignment restrictions and an unaligned relocation type will be generated if required by the target.

.align expression

Aligns the current location counter to the value given by *expression* (which must be absolute). 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**.

There is no effect if the current location is already aligned as required.

In a section of type **TEXT**, if a "hole" is created, it will be zero-filled (the nop instruction) unless a different value is specified with **-Xalign-fill-text**.

Example:

```
.align  4
```

With **-Xalign-value**, aligns on a 4-byte boundary; with **-Xalign-power2**, aligns on a 2^4 = 16-byte boundary.

.alignn expression

Aligns the current location counter to the value given by *expression* (which must be absolute).

There is no effect if the current location is already aligned as required.

In a section of type **TEXT**, if a "hole" is created, it will be zero-filled (the nop instruction) unless a different value is specified with **-Xalign-fill-text**.

Example:

```
.alignn  4
```

Will align on 4 byte boundary.

.ascii "string"

The **.ascii** directive stores the internal representation of each character in the string starting at the current location. See *String Constants*, p.300 for rules for writing the "*string*".

The **.ascii** directive is actually a synonym of the **.byte** directive — its operands may be a list of expressions including non-strings. See **.byte** for details (*.byte expression ,...*, p.314).

.asciz "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. See *String Constants*, p.300 for rules for writing the "*string*".

.balign expression

See *.alignn expression*, p.313.

.blkb expression

See *.skip size*, p.330.

.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*, p.314 above.

.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 are separated by commas.

Any expression may be a string containing one or more characters. Each character in the string will be allocated one byte. See *String Constants*, p.300 for the rules for writing a string.

Example:

```
.byte   17,65,0101,0x41         # sets 4 bytes
.byte   0                       # sets a single byte to 0
.byte   7,7,"Warning",7,7,0     # sets 12 bytes
```

.comm symbol, size [,alignment]

Define *symbol* as the address of a common block with length given by expression *size* bytes and make it global. Contrast with **.lcomm**, (*.lcomm symbol, size [,alignment]*, p.323) which does not make the symbol externally visible.

The *size* and *alignment* expressions must be absolute.

All common blocks with the same name in different files will refer to the same block. The linker will collect and allocate space for all common blocks, and, by default, place this space at the end of the **.bss** section; see *23.4 COMMON Sections*, p.352 for details.

Optional alignment

The optional *alignment* expression specifies the alignment of the common block. It must be absolute. If not specified, the default value equals the greatest power of 2 which is less than or equal to the minimum of *size* and the value specified by **-Xdefault-align** (*Set Default Value for Section Alignment (-Xdefault-align)*, p.284), which defaults to 8.

See *Interpret .align Directive (-Xalign-value, -Xalign-power2)*, p.283 for options for giving the alignment by power of 2 or the value specified. The default is to treat the *alignment* value as a power of 2.

Examples (assume **-Xdefault-align=8**):

```
.comm   a1,100      # 100 bytes aligned on an 8-byte boundary.
.comm   a2,7,2      # 7 bytes aligned on a 4-byte boundary.
```

dc.b expression

See *.byte expression ,...*, p.314 above.

dc.l expression

See *.long expression ,...*, p.324.

dc.w expression

See *.word expression, ...*, p.333.

ds.b size

See *.skip size*, p.330.

.data

Switches output to the **.data** (initialized data) section.

.double float-constant ,...

Reserves space and initializes double 64-bit IEEE floating point values.

Example:

```
double 1.0, -123.45e-56
```

.dsect

See *.data*, p.316 above.

.eject

Forces a page break if a listing is produced by the **-L** or **-l** options. See *22. Example Assembler Listing* for an example of an assembly listing.

.else

The **.else** directive is used with the **.if***x* directives to reverse the state of the conditional assembly, i.e., if statements were skipped prior to the **.else** directive, statements following the **.else** directive will be processed, and vice versa. See *.if expression*, p.320 for an example.

.elseif expression

The **.elseif** directive must follow a **.if***x* or another **.elseif** directive in a conditional assembly block. If all prior conditions (at the same nesting level) have been false, then the *expression* will be tested and if non-zero, the statements following it assembled, else statements will be skipped until the next **.elseif**, **.else**, or **.endif** directive. The *expression* must be absolute. See *.if expression*, p.320 for an example.

.elsec

See *.else*, p.316 above.

.end

This directive indicates the end of the source program. All characters after the end directive are ignored.

.endc

See *.endif*, p.317 below.

.endif

This directive indicates the end of a condition block; each **.endif** directive must be paired with a **.if***x* directive. See *.if expression*, p.320 for an example.

.endm

This directive indicates the end of a macro body definition. Each **.endm** directive must be paired with a **.macro** directive. See *21. Assembler Macros* for a detailed description.

.entry symbol ,...

See *.global symbol ,...*, p.319.

symbol[:] .equ expression

The statement must be labeled with a symbol and sets the symbol to be equal to *expression*. See **-Xlabel-colon-...** in *Set Label Definition Syntax (-Xlabel-colon...)*, p.285, regarding the initial colon. Example:

```
nine:    .equ    9
```

NOTE: Symbols defined with **.equ** may not be redefined. Use the second form of the .set directive in *.set symbol, expression*, p.329, instead of **.equ** if redefinition is required.

.error "string"

Generate an error message showing the given string. See *String Constants*, p.300 for rules for writing the "*string*".

.even

Aligns the location counter on the default alignment value, specified by the **-Xdefault-align** option (*Set Default Value for Section Alignment (-Xdefault-align)*, p.284).

.exitm

Exit the current macro invocation.

.extern symbol ,...

Declare that each symbol in the symbol list is defined in a separate module. The linker supplies the value from the defining module during linking. Multiple **.extern** directives for the same symbol are permitted. Example:

```
.extern add,sub,mul,div
```

.export symbol ,...

See *.global symbol ,...*, p.319 below.

.file "file"

Specifies the name of the source file for inclusion in 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"
```

.fill count,[size[,value]]

Reserves a block of data that is *count***size* bytes big and initialized to *count* copies of *value*. The size must be a value between 1 and 4. The default *size* is 1 and the default *value* is 0.

.float float-constant ,...

Reserves space and initializes single 32-bit IEEE floating point values. Example:

```
.float 3.14159265, .089e4
```

.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 ,...*, p.319 above.

.ident "string"

Appends the character string to a special section called **.comment** in the object file. See *String Constants*, p.300 for rules for writing the "*string*". Example:

```
.ident "version 1.1"
```

.if expression

The **.if** construct provides for conditional assembly. The *expression* must be absolute. If the *expression* evaluates to non-zero, all subsequent statements until the next **.elseif**, **.else**, or **.endif** directive at the same nesting level are assembled. If the terminating statement was **.elseif** or **.else**, then all statements following it up to the next **.endif** at the same level are skipped.

If the *expression* is zero, all statements up to the next **.elseif**, **.else**, or **.endif** at the same nesting level are skipped. An **.elseif** directive is evaluated and statements following it are skipped or not in the same manner as for the initial **.if** directive. If an **.else** directive is encountered, the statements following it up to the matching **.endif** are assembled.

.if constructs may be nested. Example:

```
          .if     long_file_names
maxname:  .equ    1024
          .elseif medium_file_names
maxname:  .equ    128
          .else
maxname:  .equ    14
          .endif
```

The following directives are equivalent: **.else** and **.elsec**, and **.endif** and **.endc**.

.ifendian

.ifendian big
Assemble the following block of code if the mode is big-endian.

.ifendian little
Assemble the following block of code if the mode is little-endian.

Note: the "endian" mode is set automatically from the target options and may not be directly changed by the user.

.ifeq expression

.ifeq is an alias for **.if** *expression* == **0**. See "**.if** expression" above for more details.

.ifc "string1","string2"

.ifc is effectively an alias for **.if** "*string1*"**=**"*string2*" (**.if** does not allow string expressions). See *.if expression*, p.320 for more details. See *String Constants*, p.300 for rules for writing each "*string*".

For compatibility with other assemblers, either string may be enclosed in single quotes rather than double quotes. Within such a single-quoted string, two single quotes will be replaced by one single quote.

.ifdef symbol

Assemble the following code if the *symbol* is defined. See also *.ifndef symbol*, p.322 below. See *.if expression*, p.320 for more details on **.if** constructs.

.ifge expression

The **.ifge** is an alias for **.if** *expression* **>= 0**. See *.if expression*, p.320 for more details.

.ifgt expression

The **.ifgt** is an alias for **.if** *expression* **> 0**. See *.if expression*, p.320 for more details.

.ifle expression

The **.ifle** is an alias for **.if** *expression* **<= 0**. See *.if expression*, p.320 for more details.

.iflt expression

The **.iflt** is an alias for **.if** *expression* **< 0**. See *.if expression*, p.320 for more details.

.ifnc "string1","string2"

.ifnc is effectively an alias for **.if** "*string1*"**!=**"*string2*" (**.if** does not allow string expressions). See *.if expression*, p.320 for more details. See *String Constants*, p.300 for rules for writing each "*string*".

For compatibility with other assemblers, either string may be enclosed in single quotes rather than double quotes. Within such a single-quoted string, two single quotes will be replaced by one single quote.

.ifndef symbol

Assemble the following code if the *symbol* is not defined. See *.ifdef symbol*, p.321 above. See also *.if expression*, p.320 for more details on **.if** constructs.

.ifne expression

.ifne is an alias for **.if** *expression* **!= 0**. See *.if expression*, p.320 for more details.

.import symbol ,...

See *.extern symbol ,...*, p.318.

.incbin "file"[,offset[,size]]

Insert the content of a specified file into the assembly output. The assembler searches for the file in the current directory and all paths added using the **-I** option. If *offset* is specified, *offset* bytes are skipped at the beginning of the file. If *size* is specified, only *size* bytes are inserted into the assembly output.

.include "file"

Inserts the contents of the named file after the **.include** directive. May be nested to any level. Example:

```
.include "globals.h"
```

.lcnt expression

Set or change the number of lines on each page of the listing file. The default value is 60. This count may be set initially by option **-Xplen** (*Set Lines Per Page (-Xplen=n)*, p.288), and it includes any margin set by option **-Xpage-skip (***Set Page Break Margin (-Xpage-skip=n)*, p.288**).** See *22. Example Assembler Listing* for an example of an assembly listing. Example:

```
.lcnt   72
```

.lcomm symbol, size [,alignment]

Define a symbol as the address of a local common block of length *size* expression bytes in the **.bss** section.

Note that the symbol is not made visible outside the current module. Contrast with **.comm** .

The *size* and *alignment* expressions must be absolute. See *Optional alignment*, p.315 for a description of the *alignment* parameter and its default value. Example:

```
.lcomm  local_array,200   # 200 bytes aligned on 8 bytes by default
```

.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. See *22. Example Assembler Listing* for an example of an assembly listing.

.literals

Flushes the assembler's accumulated literal pool for the current section. This is used to emit the literal table at the exact point desired.

Whenever the distance to the most remote instruction referencing a literal gets close to the maximum, the assembler automatically emits part of table. It tries to generate the table after a branch if possible; otherwise it will insert a branch around the table.

.llen expression

Set the number of printable character positions per line of the listing file. The default value is 132. A value of 0 means unlimited line length. This count may be set initially by option **-Xllen** (*Set Line Length of Listing File (-Xllen=n)*, p.288). See *22. Example Assembler Listing* for an example of an assembly listing. Example:

```
.llen   132
```

.llong expression ,...

Reserves 8 bytes (64 bits) for each expression in the operand field and initializes the value of the word to the corresponding expression. Example:

```
.llong   0xfedcba9876543210,0123456,-75  # 24 bytes
```

.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           # 12 bytes
```

name.macro [parameter ,...]

Start definition of macro *name*. All lines following the **.macro** directive until the corresponding **.endm** directive are part of the macro body. See *21. Assembler Macros* for a detailed description.

.mexit

Exit the current macro invocation. Synonymous with *.exitm*, p.318.

.name "file"

See *.file "file"*, p.319.

.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. See *22. Example Assembler Listing* for an example of an assembly listing.

.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.***xxxxxxxx*, where *xxxxxxxx* is the hexadecimal address of the section (with leading zeros as required to fill to eight digits). The linker will then place this section at the specified address. Example:

```
.org    0xff0000
```

will produce a section named **.abs.00ff0000** located at that address.

.p2align expression

Aligns the current location counter to 2 to the power of *expression*. The **.p2align** directive is equivalent to **.align** when the **-Xalign-power2** option is enabled.

.page

See *.eject*, p.316.

.pagelen expression

See *.lcnt expression*, p.323.

.plen expression

See *.lcnt expression*, p.323.

.previous

Assembly output is directed to the program section selected prior to the last **.section**, **.text**, **.data**, etc. directive.

.psect

See *.text*, p.331.

.psize page-length [,line-length]

Set the number of lines per page and number of character positions per line of the listing file. This directive is exactly equivalent to setting *page-length* with the *.lcnt expression*, p.323 and setting *line-length* with the *.llen expression*, p.324; see them for additional details. See *22. Example Assembler Listing* for an example of an assembly listing.

Example:

```
.psize  72,132
```

.rdata

Switches output to the **.rodata** (read-only data) section.

.rodata

Switches output to the **.rodata** (read-only data) section.

.sbss [symbol, size [,alignment]]

With no arguments, switch output to the **.sbss** section (short uninitialized data space).

With arguments, define a symbol as the address of a block of length *size* expression bytes in the **.sbss** section and make it global.

The *size* and *alignment* expressions must be absolute. See *Optional alignment*, p.315 for a description of the *alignment* parameter and its default value. Examples:

```
.sbss                   # switch to .sbss section
.sbss  local_array,200  # reserve space in .sbss section
```

.sbttl "string"

See *.subtitle "string"*, p.331.

.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*. *type* is one or more of the following characters, written as either as a quoted "*string*" or without quotes. If *type* is not specified, the default is **d** (data).

Table 20-1 **Section Type**

Type Character	Linker Command File Section Type[a]	Description of Section Contents
b	**BSS**	zero-initialized data
c	**TEXT**	executable code
d	**DATA**	data
m	**TEXT DATA**	mixed code and data
n	**COMMENT**	not allocatable — the section is not to occupy space in target memory; for example, debugging information sections such as **.debug** in ELF

Table 20-1 **Section Type** (cont'd)

Type Character	Linker Command File Section Type[a]	Description of Section Contents
o	not applicable[b]	**COMDAT** section (see *23.5 COMDAT Sections*, p.353)
r	**CONST**	readable data
w	**DATA**	writable data
x	**TEXT**	executable code

a. See *Type Specification: ([=]BSS), ([=]COMMENT), ([=]CONST), ([=]DATA), ([=]TEXT), ([=]BTEXT); OVERLAY, NOLOAD*, p.388.
b. '**o**', for **COMDAT**, is an additional attribute of a section and is usually used with another type specification character. If "**o**" is used with another section type character, the linker command file section type will be that of the other section type character; if used by itself, the default will be **COMMENT**.

The *alignment* expression must evaluate to an integer and specifies the minimum alignment that must be used for the section.

The compiler uses the **b** type with the **#pragma section** directive to specify an uninitialized section. Example: direct assembly output to a section named "**.rom**", with four-byte alignment, containing read-only data and executable code:

```
.section ".rom",4,rx
```

.section n

The assembly output is directed into the program section named "**_S***n*". Example: direct assembly output to a section named "**_S1**":

```
.section 1
```

.sectionlink section-name

This directive will cause the current section to be linked as if it had the name *section-name*. This directive is available only for ELF object output.

.set option

The following **.set** *option* directives are available:

reorder
noreorder
When processed by the **reorder** program before assembly, enable/disable reorder optimizations (thus, the **.set reorder** and **.set noreorder** directives are actually "reorder" directives rather than assembler directives). Code generated for modules compiled with optimization includes a **.set reorder** directive. Use **.set noreorder** in **asm** strings and **asm** macros in such code to disable reordering changes to these hand-coded assembly inserts. Follow with **.set reorder** to re-enable reordering optimization. See *7.4 Reordering in asm Code*, p.160.

.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
```

NOTE: Using this form of .set, the symbol may not be redefined later. Use the next form of **.set** with the symbol first on the line if redefinition is required

symbol[:] .set expression

Defines *symbol* to be equal to the value of *expression*. This form of the **.set** is different from the **.equ** directive or the form of the **.set** directive immediately above in that it is possible to redefine the value of *symbol* later in the same module. See **-Xlabel-colon-...** in *Set Label Definition Syntax (-Xlabel-colon...)*, p.285, regarding the initial colon.

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 the ELF object file format uses 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*, p.330 above.

.string "string"

See *.ascii "string"*, p.313.

.strz "string"

See *.asciz "string"*, p.314.

.subtitle "string"

Sets the subtitle to the character string. This string replaces the %*n***S** format specification in the format the string defined by the **-Xheader-format** option (see 284). The subtitle may be set any number of times. The default subtitle is blank. See *String Constants*, p.300 for rules for writing the "*string*".

```
.subtitle "string search function"
```

.text

Switches output to the **.text** (instruction space) section.

.title "string"

Sets the title to character string. The title may be set any number of times. The default title is blank. See *String Constants*, p.300 for rules for writing the "*string*". Example:

```
.title "program.s"
```

.ttl "string"

See *.title "string"*, p.331 above.

.type symbol, type

Mark *symbol* as *type*. The *type* can be one of the following:

#object
@object
object
symbol names an object

#function
@function
function
symbol names a function

@tfunc
symbol names a Thumb function

Note that only the ELF object file format uses type information.

.uhalf

This is a synonym for **.short** (*.short expression ,...*, p.330) except that there are no alignment restrictions and an unaligned relocation type will be generated if required by the target.

.ulong

This is a synonym for **.long** (*.long expression ,...*, p.324) except that there are no alignment restrictions and an unaligned relocation type will be generated if required by the target.

.ushort

This is a synonym for **.short** (*.short expression ,...*, p.330) except that there are no alignment restrictions and an unaligned relocation type will be generated if required by the target.

.uword

See *.ulong*, p.332 above.

warning "string"

Generate a warning message showing the given string. See *String Constants*, p.300 for rules for writing the "*string*".

.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 the ELF object file format supports weak external symbols. Example:

```
.weak add,sub,mul,div
```

For a further description of weak symbols see *weak Pragma*, p.134.

.width expression

See *.llen expression*, p.324.

.word expression, ...

Reserves one word (32 bits) 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 ,...*, p.319.

.xref symbol ,...

See *.extern symbol ,...*, p.318.

.xopt

Pass **-X** options to the assembler using the format:

`.xopt` *option name*`[=`*value*`]`

Example:

```
.xopt align-value
```

has the same effect as using **-Xalign-value** on the command line. In case of a conflict, **.xopt** overrides the command-line option. Also, some **-X** options are only tested before the assembly starts; in that case, the **.xopt** directive will have no effect. This option is primarily for internal use; the command-line options are preferred.

21 Assembler Macros

21.1 Introduction

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:

```
.macro movindirect reg1,reg2 // macro definition
ldr   r0,[reg1,#0]
str   r0,[reg2,#0]
.endm

movindirect  r8,r9             // macro invocation #1
movindirect  r10,r11           // macro invocation #2
```

This will produce the following code:

```
ldr   r0,[r11,#0]          // macro expansion #1
str   r0,[r10,#0]
ldr   r0,[r9,#0]           // macro expansion #2
str   r0,[r8,#0]
```

21.2 Macro Definition

A macro definition has the form:

label : **.macro** [*parameter* **,...**]

macro body

.endm

where *label* is the name of the macro, without containing any period. In addition, the following syntax is valid but is not recommended:

.macro *name* [*parameter* **,**...]

macro body

.endm

The optional parameters can be referenced in the macro body in two different ways. The following two examples show a macro which calculates

```
par1 = par2 + par3
```

(where the parameters are assumed to be in registers).

1. By using the parameter name:

```
add3:   .macro  par1,par2,par3    # definition

        add     par1,par2,par3
        .endm

        add3    r9,r10,r11        # invocation
```

   produces

```
        add     r9,r10,r11
```

2. By using \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.

```
add3:   .macro                  # definition
        add     \1,\2,\3
        .endm

        add3    r9,r10,r11      # invocation
```

produces

```
        add     r9,r10,r11
```

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   dregp,sregp     // definition
        ldr.\0   r1,[sregp,0]
        str.\0   r1,[dregp,0]
        .endm

        move.h   r10,r11         // invocation
```

produces

```
        ldr      r1,[r11,0]
        str      r1,[r10,0]
```

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
        add     par1,par2,0x&&hcnst
        .endm

        xadd    r4,r5,ff00             # invocation
```

produces

```
        add    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.

The following macro defines a string of up to four bytes in the **.data** section at a uniquely generated label (however the length of the string is not checked), and then generates code to load the contents at that label (the string itself) into a register.

```
lstr:   .macro  string,reg      ; definition
        .data
.Lm\@:
        .byte   string,0
        .previous
        ldr     reg,=.Lm\@
        .endm

        lstr    "abc",d0        ; invocation
```

produces

```
        .data
.Lm.0001:
        .byte   "abc",0
        .previous
        ldr     r0,=.Lm.0001
```

NARG Symbol

The special symbol NARG represents the actual number of non-blank parameters passed to the macro (not including any **\0** parameter):

```
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
```

21.3 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 an actual parameter that includes special characters, such as blanks, commas and comment symbols, angle brackets "< >" may be used. Everything in between the brackets is regarded as one parameter.

If the option **-Xmacro-arg-space-on** is given, blanks may be included in an actual parameter without using brackets. Example:

```
init:   .macro  command,list
        .data
        command list
        .previous
        .endm

        init    byte,<0,1,2,3>
```

produces

```
        .data
        .byte   0,1,2,3
        .previous
```

21.4 Macros to "Define" Structures

Although **struct** is not part of the assembly language, the macros shown below allow you to assign offsets to symbols so they can refer to structure members. These macros do not allocate memory; they merely assign values to symbols. The value of a structure "member" is its offset from the beginning of the structure.

The macros use **CURRENT_OFFSET_VALUE** to set the offsets of structure members: the **STRUCT** macro sets **CURRENT_OFFSET_VALUE** to 0; the **MEMBER** macro defines a symbol named for the member and having as its value **CURRENT_OFFSET_VALUE**, then increments **CURRENT_OFFSET_VALUE** by the size of the member.

```
STRUCT                  .macro
CURRENT_OFFSET_VALUE    .set        0
                        .endm

MEMBER                  .macro name, size
name = CURRENT_OFFSET_VALUE
CURRENT_OFFSET_VALUE    .set CURRENT_OFFSET_VALUE + size
                        .endm
```

CURRENT_OFFSET_VALUE must be incremented with this form of the **.set** directive because it allows the symbol so set to be set again later in the module. See *symbol[:] .set expression*, p.329 for details.

Also, note that:

- The **MEMBER** macro cannot be labeled.
- These macros cannot be used to define nested structures because there is only one **CURRENT_OFFSET_VALUE** used for all instances.
- A final **MEMBER** can be used to define the size of the structure.

Example

The macros define the symbols **first_name**, **middle_initial**, and **last_name** with values 0, 20, and 21 respectively, and define **name_size** as the total size of the "structure" with a value of 46.

```
STRUCT
MEMBER      first_name,20
MEMBER      middle_initial,1
MEMBER      last_name,25
MEMBER      name_size,0
```

One might use this, for example, as follows:

```
    .data
rec1:
    .skip       20      # reserve space for a first name
    .skip       1       # ... middle initial
    .skip       25      # ... and last name
```

Then an expression such as **rec1+last_name** in an instruction would access the **last_name** "member" of the **rec1** "structure".

22

Example Assembler Listing

If the **-l** or **-L** option is specified, a listing is produced. The **-l** option produces a listing file with the default extension **.lst** (or the extension specified with **-Xlist-file-extension**="*string*"). The **-L** option sends the listing to standard output.

The listing contains the following:

Location
: Hexadecimal value giving the relative address of the generated code within the current section.

Pl
: "Pl" stands for "Program Location counter number". Maps one-to-one to the section number in the object file (but not necessarily in the same order). When the same section is used at several discontinuous places in the source, the same section number will be used for all instances.

Code
: Generated code in hexadecimal.

Line
: Source line number.

Source Statement
: Source code lines.

To change the format of the assembly line, see *Set Format of Assembly Line in Listing (-Xline-format="string")*, p.286.

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. To change the format of the header, see *Set Header Format (-Xheader-format="string")*, p.285.

Errors are not included in the listing but are always written to **stderr**.

The following shows a listing produced by assembling an extract from file **swap.s** with the command:

```
dcc -tARMEN -c -Xpreprocess-assembly -Wa,-l -Wa,-H swap.s.
```

swap.s is used with the bubble sort example in the *Getting Started* manual. Note the use of the **dcc** command and special options passed to the assembler using the **-Wa** options (rather than the usual **das** command). This is because the actual ARM swap.s assembly file contains code for both the ARM and Thumb processors, and the **dcc** preprocessor is used to select the desired variant. The listing contains "..." lines where the code not selected or comments were deleted for brevity. Also, note the first line: the actual file assembled, **/var/tmp/d**, was the temporary output file from the preprocessor. See file **swap.c** for details.

Figure 22-1 **Assembly Listing File Swap.lst**

```
                          File: /var/tmp/d Errors    0
Location Pl  Code         Line     Source Statement
                            ...
                            24             .name       "swap.s"
                            25             .section    .text2,,c
                            26             .align      4
                            27             .xdef       swap
                            28
                            29     swap:
                            ...
00000000 01  e24d d004     33             sub         sp,sp,#4
00000004 01  e58d e000     34             str         lr,[sp,#0]
                            ...
00000008 01  e590 1000     37             ldr         r1,[r0,#0]
0000000c 01  e590 2004     38             ldr         r2,[r0,#4]
00000010 01  e580 2000     39             str         r2,[r0,#0]
00000014 01  e580 1004     40             str         r1,[r0,#4]
                            ...
00000018 01  e59d e000     45             ldr         lr,[sp,#0]
0000001c 01  e28d d004     46             add         sp,sp,#4
00000020 01  e1a0 f00e     47             mov         pc,lr
```

PART IV

Wind River Linker

23

The Wind River Linker

This section describes the linker for ARM microprocessors and is organized as follows:

- This chapter is a brief introduction to the linking process, including an example, description of special symbols created by the linker, and treatment of special sections.
- *24. The dld Command*, describes the command to invoke the linker and its options.
- *25. Linker Command Language*, describes the language used in *linker command files*.

In addition, *F. Object and Executable File Format*, describes the format of object files processed by the linker and special relocation types for those requiring such detailed information.

23.1 The Linking Process

This section provides an introduction to the linking process. Readers familiar with linker operation may proceed to *23.2 Symbols Created By the Linker*, p.350.

The linker is a program that combines one or more *binary object modules* produced by compilers and assemblers into one *binary executable file*. It may also write a text *map* file showing the results of its operation.

Each object module/file is the result of one compilation or assembly. Object files are either stand-alone, typically with the extension **.o**, or are collected in *archive libraries*, also called *libraries*. Library files typically have the extension "**.a**".

An object module contains *sections* of code (also called "text"), and "data", with names such as **.text**, **.data** (variables having initial values), **.bss** ("blank sections — uninitialized variables), and various housekeeping sections such as a symbol table or debug information.

The linker reads the sections from the object modules input to it, and based on command-line options and a *linker command file*, combines these *input sections* into *output sections*, and writes an *executable file* (usually; it is also possible to output a file which can be linked again with other files in a process called incremental linking).

A section may contain a reference to a symbol not defined in it — an *undefined external*. Such an external must be defined as *global* in some other object file. A global definition in one object file may be used to *satisfy* the undefined external in another.

As compiled or assembled into an input object file, the first byte of each input section is at address 0 (typically). But when finally located in memory as part of some output section, the input section will not be at address 0 (except for the first input section in an output section that is actually located at 0). Any absolute references to bytes in the section from within the section will therefore be "wrong" and will require *relocation*. The input object file contains sections of *relocation*

information which the linker will use to adjust such absolute references. Relocation information is used to make other similar adjustments as well.

Given the definitions above, in the abstract, the linking process consists of six steps:

1. Read the command line and linker command file for directions.
2. Read the input object files and combine the input sections into output sections per the directions in the linker command file. Globals in one object file may satisfy undefined externals in another.
3. Search all supplied archive libraries for modules which satisfy any remaining undefined externals.
4. Locate the output sections at specific places in memory per the directions in the linker command file.
5. Use the relocation information in the object files to adjust references now that the absolute addresses for sections are known.
6. If requested, write a *link map* showing the location of all input and output sections and symbols.

Linking Example

This section provides an example of the above linking process. Consider the following two C files:

File **f1.c**:

```
int a = 1;
int b;

main()
{
    b = 2;
    f2(3);
}
```

File **f2.c**:

```
extern int a, b;

f2(int c)
{
    printf("a:%d, b:%d, c:%d\n", a, b, c);
}
```

The compilation command

```
dcc -O -c f1.c f2.c
```

generates the object files **f1.o** and **f2.o**.

The contents of the two object files are shown in Table 23-1.

Table 23-1 **Linking Example Files**

Section	Type of Data	Contents
`f1.o:`		
`.text`	Code	Instructions of function **main( )**.
`.data`	Variables	Initialized variable **a**.
`.rela.text`	Relocation entries	Reference to the variable **b** inside **main( )**. Reference to the function **f2** inside **main( )**.
`.symtab`	Symbol table entries	Symbol **main**, defined in **.text** section. Symbol **a**, defined in the **.data** section. Symbol **b**, **COMMON** block of size 4. Symbol **f2**, undefined external symbol.
`f2.o:`		
`.text`	Code	Instructions of function **f2( )** and the string used in **printf( )**.
`.rela.text`	Relocation entries	Reference to the variable **a** inside **f2( )**. Reference to the variable **b** inside **f2( )**. Reference to the function **printf** inside **f2( )**. Reference to the **printf** string inside **f2( )**.
`.symtab`	Symbol table entries	Symbol **_f2**, defined in the **.text** section. Symbol **_a**, undefined external symbol. Symbol **_b**, undefined external symbol. Symbol **_printf**, undefined external symbol. Local symbol for the **printf** string, defined in the **.text** section.

Invoking the linker explicitly using the **dld** command is fully described in *24. The dld Command*. However, the easiest way to invoke the linker is to use one of the compiler drivers, for example, **dcc**, as follows:

```
dcc f1.o f2.o -o prog
```

The driver notes that the input files are objects (**f1.o** and **f2.o**) and invokes the linker immediately, supplying default values for the library, library search paths, linker command file, etc. To see how the linker is invoked, add the option **-#** to the above command; this option directs the driver to display the commands it uses to invoke the subprograms.

Schematically, the result will be as follows:

```
dcc f1.o f2.o -o prog -#
dld -YP,search-paths -l:crt0.o f1.o f2.o -o prog -lc
    version_path/conf/default.dld
```

The **-YP** option specifies directories which the linker will search for libraries specified with "**-l**" options and files specified with "**-l:***filename*" options. **crt0.o** is the C start-up module. The **-lc** option directs the linker to search for a library named **libc.a** in the paths specified by **-YP**.

With this command, the linker will proceed as follows:

1. The text file is assumed to be a linker command file (**default.dld** here), input object files are scanned in order (**crt0.o**, **f1.o**, and **f2.o**), and archive libraries are searched as necessary for undefined externals (the library filename **libc.a** is constructed from the option **-lc**).

   In this link, the file **printf.o** is loaded from **libc.a** because **printf** is not defined in the **f1.o** or **f2.o** objects. **printf.o**, in turn, needs some other files from **libc.a**, such as **fwrite.o**, **strlen.o**, and **write.o**.

2. Per the directions in the **default.dld** linker command file, input sections with the same name are combined into one output section. In this instance all **.text** sections from **crt0.o**, **f1.o**, **f2.o**, **printf.o**, **fwrite.o**, etc. are concatenated into a single output **.text** section. This also done for the other input sections. The linker command language can be used to specify how sections should be grouped together and where they should be placed in memory.

3. All "common blocks" not defined in **.text** or **.data** are placed last in the **.bss** section. See *23.4 COMMON Sections*, p.352 for details. In this case four bytes for the variable **b** are allocated in **.bss** section.

4. Once the location of all output sections is known, the linker assigns addresses to all symbols. By default, the linker puts the **.text** section in one area of memory, and concatenates the **.data** and the **.bss** sections and locates the result

in another area in higher memory. However these defaults are seldom adequate in an embedded system, and memory layout is usually controlled by a linker command file (*version_path*/**conf/default.dld** in this example).

5. All input sections are copied to the output file. While copying the raw data, the linker adjusts all address references indicated by the relocation entries. Note that there is no space in the input object file or output executable for **.bss** sections because they will be initialized by the system at execution time.
6. An updated symbol table is written (unless suppressed by the **-s**, strip option).

NOTE: While **.bss** sections do not occupy space in the linker output file, if converted to Motorola *S*-Records, *S*-Records will be generated to set the space to zeros. To suppress this, use the **-v** option to the **-R** command for **ddump** in *29. D-DUMP File Dumper*.

23.2 Symbols Created By the Linker

If necessary, the linker creates the following symbols at the end of the link process. (The linker does not recreate symbols that the user has already defined or create symbols that are never referred to in any module.) These can be used in C or assembly programs, for example, in startup code to initialize **.bss** sections to zero, or to "copy ROM to RAM" (see *Example 25-8Copying Code from "ROM" to "RAM"*, p.401, for an example of the latter).

That is, a module may declare these symbols as external and use them without ever defining them in any module. The linker will then create the symbols as described during the linking process, and satisfy the external by referring to the created symbol.

.endof.*section-name*
Address of the last byte of the named section. See Note 1.

.sizeof.*section-name*
Size in bytes of the named section. See Note 1.

.startof.*section-name*
Address of the first byte of the named section. See Note 1.

etext, **_etext**
First address after final input section of type **TEXT**. See Note 2.

edata, **_edata**
First address after final input section of type **DATA**. See Note 2.

end, **_end**
First address after highest allocated memory area.

sdata, **_sdata**
First address of first input section of type **DATA**. See Note 2.

stext, **_stext**
First address of first input section of type **TEXT**. See Note 2.

__GLOBAL_OFFSET_TABLE_,
__PROCEDURE_LINKAGE_TABLE_,
__DYNAMIC
Base addresses for access to data in the **.got**, **.plt**, and **.dynami**c sections.

Notes:

1. **.endof.**..., **.sizeof.**..., and **.startof.**... cannot be used in C code because C identifiers must include only alphanumeric characters and underscores. But they can be used in assembly code. See *Unary Operators*, p.306.
2. See *type-spec* in *Type Specification: ([=]BSS), ([=]COMMENT), ([=]CONST), ([=]DATA), ([=]TEXT), ([=]BTEXT); OVERLAY, NOLOAD*, p.388, for a discussion of output section types **DATA** and **TEXT**. As noted there, if an output section contains more than one type of input section, then its type is a union of the input section types. In this case, the symbols related to the **DATA** and **TEXT** sections as described above are not well-defined.

For example, the following prints the first address after the highest allocated memory area:

```
extern char end;

main() {
    printf("Free memory starts at 0x%x\n",end);
}
```

23.3 .abs Sections

Input files may contain sections with names of the form **.abs.***nnnnnnnn*, where *nnnnnnnn* is eight hexadecimal digits (zero-filled if necessary). Such sections will automatically be located at the address given by *nnnnnnnn*.

The compiler generates such sections in response to **#pragma section** directives of the form

#pragma section *class_name* [*addr_mode*] [*acc_mode*] [**address=***n*]

where the value given the **address=***n* clause becomes the *nnnnnnnn* in the section name.

23.4 COMMON Sections

Common variables are public variables declared either:

- In compiled code outside of any function, without the **extern** or **static** qualifier, and which are not initialized, e.g. at the module level:

```
int x[10];
```

- With **.comm** or **.lcomm** in assembly language.

Such variables are assigned to an artificial **COMMON** section.

The linker gathers all common variables together and appends them to the end of the output section named **.bss**; that is, the combined artificial **COMMON** sections for all modules becomes the end of the **.bss** output section.

These are the standard actions if the **-Xbss-common-off** option is *not* used. If the **-Xbss-common-off option is used:**

- There must be exactly one definition of each such variable in the modules of a link, with all other declarations being **extern** or **.xref**, or the linker will report an error.
- Each such variable will be part of the **.bss** section for the module in which it is defined. Because the location of individual sections may be controlled on a per file basis when linking, such variables can be located more precisely.

If an incremental link is requested (option **-r), COMMON** sections are allocated only if the **-a** option is also given.

Linker Command File Requirements with COMMON

As noted above, by default the linker places **COMMON** sections at the end of output section **.bss**. If there is no **.bss** section, then the linker command file must include a *section-contents* of the form **[COMMON]** (see *Section Contents*, p.386).

SCOMMON Section

The linker can process an **SCOMMON** section, typically holding "small" common variables and sometimes produced by other tools. This section is not normally used by the Wind River tools. Just as the **COMMON** section is appended to the **.bss** output section, the **SCOMMON** section, if present, is appended to the **.sbss** output section; if there is no **.sbss** output section, it is appended to the **.bss** output section. If neither the **.sbss** nor **.bss** output section exists, then the linker command file must contain a *section-contents* of the form **[SCOMMON]** (see *Section Contents*, p.386).

23.5 COMDAT Sections

A COMDAT section is created by using "**o**" for the section type in an assembler **.section** directive (see *.section name, [alignment], [type]*, p.327), or by using the compiler option **-Xcomdat-on** which causes sections generated for templates and run-time type information to be marked COMDAT (see *5.4.26 Mark Sections as COMDAT for Linker Collapse (-Xcomdat)*, p.66). See this latter discussion for the an example of the use of COMDAT sections.

When the linker encounters identical COMDAT sections, it removes all except one instance and resolves all references to symbols in the COMDAT section to the single instance.

If a non-COMDAT section is present along with one or more identical COMDAT sections, the linker will still collapse the COMDAT sections to one instance, but will treat the symbols in the COMDAT section as *weak*. See *weak Pragma*, p.134 for the treatment of weak symbols.

23.6 Sorted Sections

The **GROUP** definition described in *GROUP Definition*, p.392, is the usual way for a user to explicitly control the order of input sections in an output section. A second mechanism for controlling input section order, called *sorted sections* is described here.

An input section is a *sorted section* if its name begins with a period and ends with "**$***nn*", where *nn* is a two-digit decimal number, for example **.init$15**. The first part of the name (before the **$***nn*) is called the *common section name* and the **$***nn* part is called the *priority*. Input sections can also be assigned priority in the linker command file.

As described beginning on *Section-Definition*, p.385, a *section-definition* defines an *output section* and may include a list of input sections. The order in the output section of the input sections is undefined. However if the list of input sections includes a common section name, then all input sections having that common section name will be placed together and will be sorted in the output section in order of their ascending priority numeric priority.

An input section having the common section name but no priority suffix is given priority 50. The order among sorted sections with the same priority is undefined.

This sorted section feature is used by the compiler to order sections when generating initialization code. See *15.4.8 Run-time Initialization and Termination*, p.260 for details.

23.7 Warning Sections

If a section is named **.warning**, the linker prints the text from that section to standard output as a warning message if any section is loaded from the file. The warning is printed only during the final linking; incremental linking will put such sections into the output file. This is useful when the library has stub functions that need to be replaced.

Example:

```
#pragma section DATA ".warning" N

char __warning[] = "No chario output routine has been given.\n"
    "Printing through write() or printf() will not work.\n";

#pragma section DATA

int __outchar(int c, int last)
{
}
```

The linker prints the following message:

```
dld: warning:
No chario output routine has been given.
Printing through write() or printf() will not work.
```

23.8 .frame_info sections

The compiler generates **.frame_info** sections for C++ programs when exception-handling is enabled. A section is created for any function that might appear on the call stack between a **try** and a **throw**; the linker concatenates these into a searchable table that is used for stack-unwinding and object clean-up after an exception occurs. For each function, the table contains a small (8- to 24-byte) record that includes pointers to structures in the **.data** section. Since the C++ support functions in **libd.a** are compiled with exception-handling enabled, most C++ programs have at least some **.frame_info** data.

By default, C functions do not have **.frame_info** sections. To generate **.frame_info** sections for C functions—essential in mixed programs in which C++ exceptions may propagate back through C functions—use the **-Xframe-info** compiler option. Throwing an exception through C code that is not compiled with **-Xframe-info** results in a call to the C++ standard-library **terminate( )** function. Pure C++ applications and applications that only call C from C++, never the other way around, do not need to use **-Xframe-info**.

23.9 Interworking

For processors that implement the Thumb instruction set, ARM and Thumb code can be mixed. Small code sections called veneers are generated by the linker to change the instruction-set state. Use the option **-Xinterwork** if Thumb subroutines in the object file might need to return to ARM code or if ARM subroutines might need to return to Thumb code. See the *ARM Developer Guide* for more details.

24

The dld Command

24.1 The dld Command

The linker is invoked by the following command:

`dld` [*options*] *input-file* . . .

Options are described in *24.4 Linker Command-Line Options*, p.361 and *24.5 Linker -X options*, p.369.

The linker decides what to do with each *input-file* given on the command line by examining its contents to determine its type. Each file is either an object file, an archive library file, or a text file containing directives to the linker:

- **Object files**: These are loaded in the order given on the command line.
- **Archive files**: If there is a reference to an unresolved external symbol after loading the objects, then any archive library files given on the command line

(or specified with **-l** options) are searched for the symbol, and the first object module defining the missing symbol is loaded from the libraries.

Library search order depends on the use of the **-L**, **-Y L**, **-Y U**, **-Y P**, and **-Xrescan-libraries** options. See *Specify Search Directories for -l (-Y L, -Y P, -Y U)*, p.368 and *Re-scan Libraries (-Xrescan-libraries...)*, p.374 for details.

Archive libraries may be built with the **dar** tool. Archive libraries built by other archivers must conform to the ELF format accepted by the linker.

- **Text files**: A text file is interpreted as a file of linker commands. These commands are described in *25. Linker Command Language*. More than one linker command file is allowed.

Linker Command Structure

A typical linker command will be as follows in outline (where "..." means repetition, and on one line when entered):

dld -YP,*search-paths* `-o` *output-file-name* **-l:***startup-object-file object-file ...*
*library... -l**libs... linker-command-file***

where:

-YP,`search-paths`
Directories to search for files named by **-l** options and **-l:** options. The paths for the default directories are based on the default target. See *Select Target Processor and Environment (-t tof:environ)*, p.368.

(Search paths can also be specified using other **-Y** options and the **-L** option as described later).

-o *output-file-name*
Options to specify the name of the output file (the default is **a.out** if no **-o** option is given).

-l:*startup-object-file*
Startup object file. Link this file first to help establish the order of various initialization sections. Searched for in the directories specified by **-Y** or **-L** options (no path prefix allowed). Because the first character after **-l** is "**:**" the search is for a file with the exact name following the colon. Contrast with **-l** below.

Alternatively, the startup object file can be named directly on the command line, in which case a path prefix is allowed.

object-file...
The object files to be linked.

library... **-l***libs*...
Libraries to be searched for modules defining otherwise undefined external symbols. Libraries can be given directly on the command line with path prefix, or searched for in the **-Y** or **-L directories** by using the **-l***name* form. In the latter case, the library name **lib***name***.a** is constructed from *name* and no path prefix is allowed.

linker-command-file
Text file of linker commands. A path prefix is allowed.

To get a map to **stdout**, add the **-m**, **-m2**, or **-m6** option (with increasing detail).

A good way to gain experience with linker command lines, and to see default values for the parts of the command line outlined above, is to invoke **dcc** or **dplus** with the **-#** option to show the command line for each subprogram. For example, the following command line:

```
dcc -# -o hello.out hello.c -m > hello.map
```

would effectively invoke the linker with the following command line (assumes default of no floating point, and shows each argument on a separate line for readability):

```
dld -Y P,/diab/4.x/ARMEN/simple:/diab/4.x/ARMEN:
        /diab/4.x/ARME/simple:/diab/4.x/ARME
    -l:crt0.o
    hello.o
    -o hello.out
    -lc
    /diab/4.x/conf/default.dld
    -m > hello.map
```

where:

-Y P,/diab/4.*x*/...
Directories to search for files named by **-l** options.

-l:crt0.o
Startup object file from the directories specified by the **-YP** option.

hello.o
The object module to be linked.

-o hello.out
The name of the output file instead of **a.out**.

-lc
Search for library **libc.a** for modules defining unresolved externals in the directories specified by **-YP**.

/diab/4.*x*/conf/default.dld
Use the default linker command file.

-m > hello.map
Request a minimal map and redirect it from **stdout** to **hello.map** (the driver **dcc** passes any option it does not recognize, the **-m** in this case, to the linker).

24.2 Defaults

In addition to application input object files, the linker typically needs a linker command file to direct the link, libraries to satisfy undefined externals, and often a startup object file.

When the linker is invoked explicitly with the **dld** command, there will be no default linker command file, no libraries, and no startup file — all must be specified using command-line options as described in this chapter.

When the linker is invoked automatically by the **dcc** or **dplus** drivers, it is invoked with options which specify default linker command file, libraries, and startup object file.

These defaults are as follows:

- Linker command file: the default is *version_path*/**conf/default.dld**. To specify a different linker command file when using **dcc** or **dplus**, use the **-Wm***file* option (*5.3.28 Specify Linker Command File (-W mfile)*, p.43). Note that **-Wm** is an option to the compiler driver directing its sub-invocation of the linker; **-Wm** is not a linker option. To provide a linker command file when invoking the linker directly, just name it on the **dld** command line.
- Libraries: the defaults are libraries **libc.a** and, for C++, **libd.a** from the directories associated with the default target, and/or as specified with **-l**, **-L,** and/or **-Y** options on the command line as documented later in this manual.
- Startup object file: the default is **crt.o from the selected** target subdirectory. To specify a different startup object file when using **dcc** or **dplus**, use the **-Ws***file* option (*5.3.29 Specify Startup Module (-W sfile)*, p.43). As with **-Wm** this is a driver, not a linker, option.

To see the defaults for a particular case, execute **dcc** or **dplus** with the **-#** option to display the command line for the compiler, assembler, and linker as each is automatically invoked.

NOTE: Linker command files formerly used an extension of **.lnk**. As of version 4.2, this is changed to **.dld** because **.lnk** is used by Windows to designate a shortcut. In the **conf** directory, identical copies of each linker command file using each extension will be present for an interim period.

24.3 Order on the Command Line

Options and files may be intermixed and may be given in any order except that an option which specifies a search directory for **-l, that is -L or -Y,** must be given before a **-l** to which it is to apply. However the following order is recommended:

- options
- object files
- libraries and **-l** options which name libraries
- linker command file

Other options may be mixed in any order. While libraries and objects may be in any order (with the default setting of **-Xrescan-libraries**, see *Re-scan Libraries (-Xrescan-libraries...)*, p.374), a link will be faster if there is no need to re-scan a library. The linker may also be more efficient in processing a linker command file if its has encountered all objects first.

24.4 Linker Command-Line Options

This section contains standard command-line options common to many linkers. The next section documents **-X** options which provide additional detailed control over the linker (beginning with *24.5 Linker -X options*, p.369).

For a concise list of all options, see the table of contents.

Show Option Summary (-?, -?X)

-?, -h
--help
Show synopsis of command-line options.

-?X, -hX
Show synopsis of **-X** options (see *24.5 Linker -X options*, p.369).

Read Options From an Environment Variable or File (-@name, -@@name)

-@ *name*
Read command-line options from environment variable *name* if it exists, else from file *name*.

In an environment variable, separate options with a space. In a file, place one or more options per line, separated by a space.

-@@*name*
Same as **-@***name*; also prints all command-line options on standard output.

Redirect Output (-@E=file, -@E+file, -@O=file, -@O+file)

-@E=*file*
Redirect any output to standard error to the given *file*.

-@O=*file*
Redirect any output to standard output to the given *file*.

Use of "**+**" instead of "**=**" will append the output to the file.

Link Files From an Archive (-A name, -A...)

-A *filename*
-A -l*name*
-A -l:*filename*
Link all files from the specified archive. The **-A** option affects only the argument immediately following it, which can be a filename or **-l** option. (See *Specify Library or File to Process (-lname, -l:filename)*, p.365.) If *filename* or *name* is not an archive, **-A** has no effect.

Sections can still be dropped with the **-Xremove-unused-sections** option.

-A1...
Same as **-A**.

-A2...
Same as **-A**, but overrides **-Xremove-unused-sections** for the specified archives.

-A3...
Same as **-A2**, but also overrides **-s** and **-ss** for the specified archives.

Allocate Memory for Common Variables When Using -r (-a)

-a
Common variables are not normally allocated when an incremental link is requested by the **-r** option. The **-a** option forces allocation in this case. See *23.4 COMMON Sections*, p.352 for details.

Set Address for Data and tExt (-Bd=address, -Bt=address)

-Bd=*address*
-Bt=*address*
Allocate **.text** section and other constant sections to the given *address*. The **-Bd** and **-Bt** options provide a simple way to define where to allocate the sections without having to write a linker command file. If either **-Bd** or **-Bt** is specified, the linker will use the following command specification:

```
SECTIONS {
  GROUP Bt-address : {
    .text (TEXT) : {
          *(.text) *(.rdata) *(.rodata)
          *(.init) *(.fini)
    }
    .sdata2 (TEXT) : {}
  }
  GROUP Bd-address: {
    .data (DATA) : {}
    .sdata (DATA) : {}
    .sbss (BSS) : {}
    .bss (BSS) : {}
}
```

If the **-N** option is given, the **.data** section is placed immediately after the **.text** section.

NOTE: The **-Bd** and **-Bt** options are ignored if a linker command file is present. The **default.dld** linker command file will be present by default if the linker is invoked implicitly by **dcc** or **dplus**. To use **-Bd** and **-Bt**, suppress the use of the default linker command file with the **-W m** option with no name on the **dcc** or **dplus** command line

Bind Function Calls to Shared Library (-Bsymbolic)

When creating a shared library, bind function calls, if possible, to functions defined within the shared library. For VxWorks RTP application development.

Define a Symbol At An Address (-Dsymbol=address)

-D*symbol=address*
Define specified symbol at specified address.

Define a Default Entry Point Address (-e symbol)

-e *symbol*
symbol is made the default entry address and entered as an undefined symbol in the symbol table. It should be defined by some module.

Specify "fill" Value (-f value, size, alignment)

-f *value*
-f *value*, *size*
-f *value*, *size*, *alignment*
Fill all "holes" in any output section with 16-bit *value* rather than the default value of zero. Optional *size* and *alignment* are specified in bytes; the default is 2, 1.

Specify Directory for -l search List (-L dir)

-L *dir*
Add *dir* to the list of directories searched by the linker for libraries or files specified with the **-l** option. More than one **-L** option can be given on the command line.

Must occur prior to a **-l** option to be effective for that option.

Specify Library or File to Process (-lname, -l:filename)

-l*name*
Specify a library with the constructed name **lib***name***.a** to be searched for object modules defining missing symbols.

-l:*filename*
Process the given *filename* (without modification, no path prefix allowed): an object file is linked, an archive is searched as necessary, a text file is taken as a linker command file.

For both forms, search for the file is performed in the following order:

- The directories given by **-L** *dir* options in the order these options are encountered.
- The directories as given by any **-Y L**, **-Y P**, or **-Y U** options (see these options in *Specify Search Directories for -l (-Y L, -Y P, -Y U)*, p.368).

Any **-L** or **-Y** option must occur prior to all **-l** options to which it applies.

If no **-L** or **-Y** option is present, search a set of directories based on the selected target and environment. See *4.2 Selected Startup Module and Libraries*, p.24 for details.

Generate link map (-m, -m2, -m4)

-m (equivalent to **-m1**)
Generate a link map of the input and output sections on the standard output.

-m2
Generate a more detailed link map of the input and output sections, including symbols and addresses, on the standard output. **-m2** is a superset of **-m1.**

-m4
Generate a link map with a cross reference table.

-m6
Equivalent to **-m2** plus **-m4**: generated a detailed link map and cross reference table.

The value following "**m**" is converted to hexadecimal and used as a mask; thus, **-m3** is equivalent to **-m2.** Undefined bits in the mask are ignored**.**

Allocate .data Section Immediately After .text Section (-N)

-N
This option is used in conjunction with options **-Bd** and **-Bt**. See them for details (*Set Address for Data and tExt (-Bd=address, -Bt=address)*, p.363).

Change the Default Output File (-o file)

-o *file*
Use *file* as the name of the linked object file instead of the default filename **a.out**.

Perform Incremental Link (-r, -r2, -r3, -r4, -r5)

-r
The linked output file will still contain relocation entries so that the file can be re-input to the linker. The output file will not be executable, and no unresolved reference complaints will be reported.

-r2
Link the program as usual, but create relocation tables to make it possible for an intelligent loader to relocate the program to another address. Absent other options, a reference to an unresolved symbol is an error.

-r3
Equivalent to the **-r2** option except that unresolved symbols are not treated as errors.

-r4
Link for the VxWorks loader.

-r5

Equivalent to the **-r** option except that **COMDAT** sections are merged and converted to normal sections.

The **-r** options are required only for *incremental* linking, not when producing an ordinary absolute executable.

Rename Symbols (-R symbol1=symbol2)

-R *symbol1=symbol2*

Rename symbols in the linker output file symbol table. The order of the symbol names is not significant; **-R** *symbol1=symbol2* does the same thing as **-R** *symbol2=symbol1*. If both symbols exist, both are renamed: *symbol1* becomes *symbol2* and *symbol2* becomes *symbol1*.

Search for Shared Libraries on Specified Path (-rpath)

-rpath *path*

Search for shared libraries on specified *path*, a colon-separated list of directories. (If no search path is specified, the linker looks in the directory where the executable resides.) For VxWorks RTP application development.

Do Not Output Symbol Table and Line Number Entries (-s, -ss)

-s

Do not output symbol table and line number entries to the output file.

-ss

Same as **-s,** plus also suppresses all **.comment** sections in the output file.

Specify Name for Shared Library (-soname)

-soname=*libraryName*

Use *libraryName* as the name of the shared object containing compiled library code. For VxWorks RTP application development.

Select Target Processor and Environment (-t tof:environ)

-t t*of*:*environ*
Select the target processor, object format, floating point support, and environment libraries. See the **-t** option in *4. Selecting a Target and Its Components* for details. This option is not valid in a linker command file.

Define a Symbol (-u symbol)

-u *symbol*
Add *symbol* to the symbol table as an undefined symbol. This can be a way to force loading of modules from an archive.

Print version number (-V)

-V
Print the version of the linker.

Do Not Output Some Symbols (-X)

-X
Do not output symbols starting with **@L** and **.L** in the generated symbol table. These symbols are temporaries generated by the compiler.

Specify Search Directories for -l (-Y L, -Y P, -Y U)

-Y L,*dir*
Use *dir* as the first default directory to search for libraries or files specified with the **-l** option.

-Y P,*dir*
dir is a colon-separated list of directories. Search each of the directories in the list for libraries or files specified with the **-l** option.

-Y U,*dir*
Use *dir* as the second default directory to search for libraries or files specified with the **-l** option.

Notes:

1. These options must occur prior to all **-l** options to which they are to apply.
2. The **dcc** and **dplus** programs (but not **dld** itself) generate a **-Y P** option suitable for the selected target and environment. Unless you are replacing the libraries, you should not normally use this option. Use the **-L** option to specify libraries to be searched before the Wind River libraries. (See *Specify Directory for -l search List (-L dir)*, p.365.)
3. If no **-Y** or **-l** options are present on the **dld** command line, the linker will automatically search the directories associated with the default target. See *4.2 Selected Startup Module and Libraries*, p.24 for details.
4. If a **-Y** option is used, **-Y P** is recommended. The older **-Y L** and **-Y U** options are provided for compatibility. Use of **-Y P** together with **-Y L** or **-Y U** is undefined.

24.5 Linker -X options

The following **-X** options provide additional detailed control over the linker. Many are present to improve compatibility and ease of conversion from other tool sets.

For a concise list of all options, see the table of contents.

Use Late Binding for Shared Libraries (-X)

-Xbind-lazy
Bind each shared-library function the first time it is called. (By default, binding occurs when the module is loaded.) For VxWorks RTP application development.

Check Input Patterns (-Xcheck-input-patterns)

-Xcheck-input-patterns
Check that every input section pattern in the linker command file matches at least one input section. Emit a warning if an unmatched pattern is found.

-Xcheck-input-patterns=2
Same as **-Xcheck-input-patterns**, but emit a message of severity level "information" instead of "warning". (For use with **-Xstop-on-warning**.)

Check for Overlapping Output Sections (-Xcheck-overlapping)

-Xcheck-overlapping
Check for overlapping output sections and sections which wrap around the 32-bit address boundary.

Force Linker to Continue After Errors (-Xdont-die)

-Xdont-die
Force the linker to continue after errors which would normally halt the link. For example, issue warnings rather than errors for undefined symbols and out-of-range symbols.

When the linker is forced to continue it produces reasonable output and returns error code 2 to the parent process. By default, the make utility stops on such errors; if you want it to continue you must handle this error code in the makefile explicitly.

Do Not Create Output File (-Xdont-link)

-Xdont-link
Do not create a linker output file. Useful when the linker is started only to create a memory map file.

Use Shared Libraries (-Xdynamic)

-Xdynamic
Link against shared libraries (**.so** files). For VxWorks RTP application development.

Use ELF Format for Output File (-Xelf)

-Xelf
This is the default.

ELF Format Relocation Information (-Xelf-rela-...)

-Xelf-rela
Use RELA relocation information format for ELF output. This is the default.

-Xelf-rela-off
-Xelf-rela=0
Use REL relocation information format for ELF output.

Do Not Export Symbols from Specified Libraries (-Xexclude-libs)

-Xexclude-libs=*list*
Do not automatically export symbols from the libraries specified in the comma-delimited *list*. (Use the same library names, prefixed with "l", that you would use with the **-l** option.) Example: **-Xexclude-libs=lc,lm**. For VxWorks RTP application development.

Do Not Export Specified Symbols (-Xexclude-symbols)

-Xexclude-symbols=*list*
Do not export the symbols specified in the comma-delimited *list* when creating a shared library. Example: **-Xexclude-symbols=function1,function2**. For VxWorks RTP application development.

Write Explicit Instantiations File (-Xexpl-instantiations)

-Xexpl-instantiations
Cause the linker to write the source lines of an explicit instantiations file to **stdout**. To minimize space taken by template classes, the output from **-Xexpl-instantiations** can be used to create an explicit instantiations file (necessary header files must still be added); see *Templates*, p.223. This option is deprecated.

Store Segment Address in Program Header (-Xgenerate-paddr)

-Xgenerate-paddr
Store the address of each segment in the **p_paddr** field of the corresponding entry in the program header table. Without this option, the **p_paddr** value will be 0.

Generate RTA Information (-Xgenerate-vmap)

-Xgenerate-vmap
Generates special information used by the RTA.

Do Not Align Output Section (-Xold-align)

-Xold-align
Do not align output sections.

Without this option (the default), each output section is given the alignment of the input section having the largest alignment. Output sections must be aligned to support position-independent code.

With this option, output sections are not aligned, and each output section begins immediately after the previous output section. (In this later case, input sections will still be aligned per their requirements, potentially leaving a gap from the start of the output section to the start of the first input section within it.)

Pad Input Sections to Match Existing Executable File (-Xoptimized-load)

-Xoptimized-load=*n*
-Xoptimized-load
Minimize the difference between the already existing executable file (if any) and the new file by padding input sections. *n* specifies how much relative space the linker can use for padding, where 0 means no padding and 100 is the default. The larger the value of *n*, the more similar the images are likely to be.

The linker saves the old executable file with the **.old** extension and generate a diff file with the **.blk** extension.

Add Leading Underscore "_" to All Symbols (-Xprefix-underscore)

-Xprefix-underscore
Add a leading underscore "_" to all symbols in the files specified after this command. Use **-Xprefix-underscore=**0 to turn off this feature. The default is off.

Use Workaround for ELF Relocation Bug (-Xreloc-bug)

-Xreloc-bug
Enables a workaround for a bug in the ELF relocation information generated by some compilers. Do not use with object files created by the Wind River compiler.

Remove Unused Sections (-Xremove-unused-sections)

-Xremove-unused-sections
-Xremove-unused-sections-off
Remove all unused sections. By default the linker keeps unused sections.

A section is used if it:

- Is referred to by another used section.
- Has a program entry symbol—that is, a symbol defined with the **-e** option (*Define a Default Entry Point Address (-e symbol)*, p.364) or one of **__start**, **_start**, **start**, **__START**, **_START**, **_main**, or **main** (order reflects priority).
- Is not referenced by any section and has a name that starts with **.debug**, **.fini**, **.frame_info**, **.init**, **.j_class_table**, or **.line**.
- Defines a symbol used in an expression in the linker command file.
- Defines a symbol specified with the **-u** option (*Define a Symbol (-u symbol)*, p.368).

NOTE: This option is especially useful in combination with **-Xsection-split** (*5.4.116 Generate Each Function in a Separate CODE Section Class (-Xsection-split)*, p.104) to reduce code size. When both options are used, each function in a module will generate a separate **CODE** section, and thus functions which are not called will be removed.

Re-scan Libraries (-Xrescan-libraries...)

-Xrescan-libraries
-Xrescan-libraries-off
Request that the linker re-scan libraries to satisfy undefined externals. This is the default. It solves the ordering problem which occurs when one library uses symbols in another and vice-versa.

Use **-Xrescan-libraries-off** to force the linker to scan libraries and object files in precisely the order given on the command line.

Re-scan Libraries Restart (-Xrescan-restart...)

-Xrescan-restart
-Xrescan-restart-off
If **-Xrescan-libraries** is on, when more than one library is presented to the linker, force the linker to rescan the libraries from first to last in order for each undefined symbol. This is the default.

Use **-Xrescan-restart-off** with **-Xrescan-libraries** to cause the linker, after finding symbols in one library, to continue with the next library for the rest of the undefined symbols.

Align Sections (-Xsection-align=n)

-Xsection-align=*n*
Force COFF input sections to have an alignment of *n* instead of the default 8. (Ignored for ELF output.)

Build Shared Libraries (-Xshared)

-Xshared
Build shared libraries (rather than stand-alone executables). For VxWorks RTP application development.

Sort .frame_info Section (-Xsort-frame-info)

-Xsort-frame-info
-Xsort-frame-info-off
To enable sorting of the **.frame_info** section, use **-Xsort-frame-info**. By default, sorting is disabled (**-Xsort-frame-info-off**).

Link to Static Libraries (-Xstatic)

-Xstatic
Link against static (**.a**) libraries rather than shared (**.so**) libraries. Use this option when both static and shared libraries are available. For VxWorks RTP application development.

Stop on Redeclaration (-Xstop-on-redeclaration)

By default, the linker issues a warning each time it encounters a redeclaration. If **-Xstop-on-redeclaration** is specified, the linker halts with an error on the first redeclaration.

Stop on Warning (-Xstop-on-warning)

-Xstop-on-warning
Request that the linker stop the first time it finds a problem with severity of warning or greater.

Suppress Leading Dots "." (-Xsuppress-dot)

-Xsuppress-dot
Suppress leading dots "**.**" in the object files following this option.

Suppress Section Names (-Xsuppress-section-names)

-Xsuppress-section-names
Do not output section names to the symbol table. This option is for other tools which cannot process these names.

Suppress Paths in Symbol Table (-Xsuppress-path)

-Xsuppress-path
In the symbol table, suppress any pathname in "file" symbols (type **STT_FILE**, see Table F-3).

Suppress Leading Underscores '_' (-Xsuppress-underscore)

-Xsuppress-underscore
Suppress leading underscores "_" in the object files following this option. Note that for symbols with more than one leading underscore, only the first will be removed.

Remove/Keep Unused Sections (-Xunused-sections...)

-Xunused-sections-remove
Same as **-Xremove-unused-sections** (*Remove Unused Sections (-Xremove-unused-sections)*, p.373).

-Xunused-sections-keep
Same as **-Xremove-unused-sections-off** (*Remove Unused Sections (-Xremove-unused-sections)*, p.373).

-Xunused-sections-list
Print a list of removed sections.

25

Linker Command Language

The linker command language can:

- Specify input files and options.
- Specify how to combine the input sections into output sections.
- Specify how memory is configured and assign output sections to memory areas.
- Assign addresses or other values to symbols.

A default linker command file, **default.dld**, is present in the **conf** directory. See *24.2 Defaults*, p.360 for its use.

25.1 Example "bubble.dld"

Some examples in this chapter are drawn from the **bubble.dld** command file on the next page for the "bubble sort" program in the *Getting Started* manual. This example is distributed with the compiler suite in directory *version_path***/example/arm**. The chapter ends with additional unrelated examples. Some notes follow the figure.

Figure 25-1 **bubble.dld Linker Command File Extract**

Linker Commands	Explanation

`MEMORY`
`{`
`  rom1:  org = 0x20000, len = 0x10000`
`  rom2:  org = 0x30000, len = 0x10000`
`  ram:  org = 0x80000, len = 0x30000`
`  stack:  org = 0xb0000, len = 0x10000`
`}`
	Define four memory areas.

`SECTIONS`
`{`
`  .text : { *(.text) *(.rodata)`
`  *(.init) *(.fini)`
	Collect code sections from all input files into a single output **.text** section and locate it in **rom1** (except for **.text2** code sections).

`  .ctors ALIGN(4):`
`  { ctordtor.o(.ctors) *(.ctors) }`
`  .dtors ALIGN(4):`
`  { ctordtor.o(.dtors) *(.dtors) }`
	.ctors and **.dtors** sections for startup and termination invocation.

`  } > rom1`	

`  .text2 : { *(.text2)`
`  __DATA_ROM = .;`
`  } > rom2`
	Collect all **.text2** sections and locate in **rom2**.
	Define **__DATA_ROM** as equal to the current location. The symbols defined this way are used within this file and during initialization.

`  GROUP : {`	Group **.data** and **.bss** output sections together in the order given.
`  __DATA_RAM = .;` `  .data LOAD(__DATA_ROM) : {}`	Collect initialized data sections (**.data**) from all input files “{}” into a single output **.data** section and logically locate in **ram**.
`  __DATA_END  = .;`	But use **LOAD** to place the actual data after the **.text2** section in **rom2**. **__init_main( )** will move the actual data from **rom2** to **ram**.

`  __BSS_START  = .;`
`  .bss : {}`
`  __BSS_END = .;`
	Reserve space for all **.bss** sections in **ram** after the **.data** section. Any remaining space will be used as heap by **malloc( )**.

`  ___HEAP_START = .;` `  } > ram`	Define other symbols used by **crt0.s**, **init.c**, and **sbrk.c** to control initialization and memory allocation: ... Start of heap memory for **sbrk.c**.
`}`	
`__HEAP_END  = ADDR(ram)+SIZEOF(ram);`	... End of heap memory for **sbrk.c**.
`__SP_INIT  = ADDR(stack)+SIZEOF(stack);`	... Initial address of stack pointer for **crt0.s**.
`__SP_END  = ADDR(stack);`	... Only used when stack probing by **sbrkbc**.

Notes for bubble.dld

Two features of **bubble.dld** are especially noteworthy:

- The use of the **LOAD** specification to create two images of variables having initial values, a *physical* image containing the initial values and intended for

some form of read-only memory, and a *logical* image where the variables will reside during execution. See the *LOAD Specification*, p.390 and *Copying Initial Values From "ROM" to "RAM", Initializing .bss*, p.257 for details.

- The definition of nine of the symbols:
  - **__DATA_ROM**, **__DATA_RAM**, and **__DATA_END** used in copying the initial values and **__BSS_START** and **__BSS_END** used in clearing static uninitialized variables (see *Copying Initial Values From "ROM" to "RAM", Initializing .bss*, p.257).
  - **__HEAP_START** and **__HEAP_END** to define the heap for use by **malloc( )** and related functions. See *15.4.7 Dynamic Memory Allocation - the heap, malloc( ), sbrk( )*, p.260.
  - **__SP_INIT** and **__SP_END** to define the stack. See *15.4.6 Stack Initialization and Checking*, p.259.

25.2 Syntax Notation

Italic words such as *area-name* represent items you must supply. The required type of each item — symbol name or number, can be gathered from the examples.

The following special characters are parts of commands and are required where shown:

```
{  }  (  )  ,  ;  >  *
```

The following characters are used only in the command descriptions and not in the linker command language itself. They have the meanings shown:

|
"or"

[ ]
The enclosed construct is optional. When several optional items are adjacent, they may be given in any order.

...
The preceding item or construct may be repeated.

For example

```
a [b | c] ...
```

means that **a** is required, then any number of **b** or **c**.

Note that the "{" and "}" characters are part of commands and do *not* indicate a set of alternatives from which one must be chosen.

Long lists of alternative tokens are given by following the phrase "one of" with a list of the tokens on one or more lines, as in

```
assign-op: one of
    =   +=  -=  *=  /=
```

25.3 Numbers

Several linker commands require a number, for example to specify an address or a size.

Numbers are hexadecimal if they begin with "0X" or "0x", else octal if they begin with "0", else decimal. Hexadecimal digits are "0" - "9", "a" - "f", and "A" - "F"; octal digits are "0" - "7"; decimal digits are "0V - "9".

25.4 Symbols

A *symbol*, once defined, may be used anywhere a number is required except in a **MEMORY** command. Symbols are defined in object files or by assignment commands (see *25.9 Assignment Command*, p.393).

A *symbol* defined in an assignment command is an identifier following the rules of the C language with the addition of "**$**" and "**.**" as valid characters. Symbols may be up to 1,000 characters long.

NOTE: A symbol or filename which does not follow these rules may be given by quoting it with double-quote characters, for example, an object file named "**1234o.o**".

25.5 Expressions

A linker *expression* is allowed anywhere a number is required, and is one of the following forms from the C language:

number
symbol
unary-op expression
expression binary-op expression
expression **?** *expression* **:** *expression*
(*expression* **)**

where the operators are the following operators from the C language:

unary-op: one of

```
!  ~  -
```

binary-op: one of

```
*  /  %
+  -
>>  <<
==  !=  ><  <=  >=
&
|
&&
||
```

The operators have their meaning and precedence as in C. Parentheses can be used to change the precedence.

When a symbol name is used in an expression, the address of that symbol is used. The symbol"**.**" means the current location counter (allowed only within a statement list in a **SECTIONS** command).

The following pseudo functions are valid in expressions. Forward references are permitted.

SIZEOF (*section-name***)**
Size of the named section (see *Example 25-6Empty Sections*, p.397 for an important limitation when using the **SIZEOF** operator).

SIZEOF (*memory-area-name***)**
Size of a memory area defined with the **MEMORY** command.

ADDR (*section-name***)**
Address of the named section.

ADDR (*memory-area-name***)**
Address of a memory area.

NEXT (*expr***)**
First multiple of *expr* that falls into unallocated memory.

HEADERSZ
Total size of all the headers.

FILEOFFSET (*section-name***)**
File offset of the named section.

ALIGN (*value***)**
((. + *value*–1) & ~(*value*–1))

25.6 Command File Structure

A command file is a list of commands. These are:

MEMORY { *memory-area-definition* **}**
SECTIONS { *section-or-group-definition ...* **}**
assignment-command
object-filename
archive-filename
command-line-option

The above commands may each be repeated as many times as required and may be given in any order as long as names are defined before use.

Each of these commands is described below except for the last three: in addition to, or instead of, being given as arguments on the command line, object and archive library files and command-line options may be given as commands.

NOTE: While different object files may be named on both the command line and in a linker command file, do not duplicate the same object filename in both places. This may cause sections from the duplicated object file to be duplicated in memory.

The command language is free format. More than one command may be given on a line, and a command may be written on multiple lines without need for any special continuation character.

Identifiers are as in C with the addition of period "**.**" and "**$**" as a valid identifier characters; identifiers may be up to 1,000 characters long.

Whitespace is generally required as in C around identifiers and numbers but not special characters.

C-style comments are allowed anywhere whitespace would be.

25.7 MEMORY Command

```
MEMORY {
    area-name : { origin | org | o } = start-address [ , ]
                { length | len | l } =  number-of-bytes [ , ]
    ...
}
```

The **MEMORY** command names one or more areas of memory, e.g. "rom", "ram". Each area is defined by a start address and a length in bytes. A later *section-definition* command can then direct that an output section be located in a named area. The linker will warn if the total length of the sections assigned to any area exceeds the area's length. Example:

```
MEMORY {
    rom1:   org = 0x010000, len = 0x10000
    rom2:   org = 0x020000, len = 0x10000
    ram:    org = 0x100000, len = 0x70000
    stack:  org = 0x170000, len = 0x10000
}
```

Symbols (*25.4 Symbols*, p.381) cannot be used within the **MEMORY** command; *start-address* and *number-of-bytes* must be numeric expressions.

25.8 SECTIONS Command

```
SECTIONS {
    section-definition | group-definition
    ...
}
```

The **SECTIONS** command does most of the work in a linker command file. Each input object file consists of *input sections*. The primary task of the linker is to collect input sections and link them into *output sections*. The **SECTIONS** command defines

each output section and the input sections to be made part of it. Within the **SECTIONS** command, a **GROUP** statement may be used to collect several output sections together.

The components of the **SECTIONS** command are described next. See Figure 25-1 for example illustrating many of the possibilities.

Section-Definition

At a minimum, each *section-definition* defines a new *output section* and specifies the *input sections* that are to be put into that output section. Optional clauses may:

- Specify an address for the output section or place the output section in a memory area defined by an earlier **MEMORY** command.
- Align the section.
- Fill any holes in the section with a fixed value.
- Define symbols to be used later in the linker command file or in the code being linked.

The full form of a *section-definition* is shown in Figure 25-2. For clarity, each clause is written on a separate line and is identified to its right for description below.

Figure 25-2 **section-definition**

Syntax	**Element**
output-section-name [(**[=]{BSS** | **COMMENT** | **CONST** | **DATA** | **TEXT** | **BTEXT}[OVERLAY][NOLOAD**)]	*type-spec*
[*address-value* | **BIND** (*expression*)]	*address-spec*
[**ALIGN** (*expression*)]	*align-spec*
[**LOAD** (*expression*)]	*load-spec*
[**OVERFLOW** (*size-expression* , *overflow-section-name*)	*overflow-spec*
: { *section-contents* }	
[=*fill-value* | =(*fill-value*[, *size*[, *alignment*]])]	*fill-spec*
[>*area-name*]	*area-spec*

Note that most clauses are optional, and section modifiers (those preceding the ":") may be in any order. Thus, the minimum *section-definition* has the form:

output-section-name **:** { *section-contents* }

NOTE: Exercise caution when naming custom sections. Section names that begin with a dot (.) may conflict with the compilation environment's namespace.

Section Contents

section-contents is required in a *section-definition*. *section-contents* is a sequence of one or more of the forms from Figure 25-1 separated by whitespace or comment:

<empty>
That is, { } with no explicitly named *section-contents*: include in the output section all sections from all input object files which have the same name as the *output-section-name*. Example:

```
.data : { }
```

NOTE: The <empty> form is processed only after the linker has examined and processed all other input specifications. Thus, input sections loaded directly or indirectly as a result of other more explicit specifications will not be re-loaded by an { } form, even if they appear after it.

filename
Include all sections from the named object file which have the same name as the *output-section-name*. Example:

```
.data : { test1.o, test2.o }
```

`*` `(` *input-section-spec* ...)
input-section-spec may be one of four forms:

section-name
Include the named sections from all input object files *but do not include input sections already included earlier.* Example:

```
.data : { *(.data) }
```

section-name[*symbol*]
Include the section defining the given symbol. The "[" and "]" characters do not mean "optional" in this case but rather are to be used as shown. Example:

```
.text : { *(.text[malloc]) }
```

This form is especially useful with option **-Xsection-split**. See *5.4.116 Generate Each Function in a Separate CODE Section Class (-Xsection-split)*, p.104.

*
Include all sections.

input-section-spec=*n*
Include sections according to *input-section-spec* and assign them priority *n*. (See *23.6 Sorted Sections*, p.354.)

object-filespec **(** *input-section-spec* ... **)**
Include the named sections from the named object file, where *input-section-spec* is as defined immediately above and *object-filespec* is a pattern expression.* Example:

```
.rom1 : { rom1.o(.data), rom1.o(.sdata) }
```

archive-filespec[*member-name*] (*input-section-spec* ...)
Include the named sections from the named object file, where *input-section-spec* is as defined above, and *archive-filespec* and *member-name* are pattern expressions.* Example:

```
.text : { libproj.a[malloc.o](.text) }
```

[COMMON]
For explicit placement of **COMMON** sections. See *Linker Command File Requirements with COMMON*, p.353 for additional information.

[SCOMMON]
For explicit placement of **SCOMMON** sections. See *SCOMMON Section*, p.353 for additional information.

assignment-command
Define a symbol or change the program counter to create a “hole” (which may be filled by a *fill-value*). See *25.9 Assignment Command*, p.393.

ASSERT (*expression*[**,** *text*] **)**
Evaluate *expression* and display an error message if *expression* is zero. Optional *text* is included in error message.

STORE (*expression*, *size-in-bytes* **)**
Reserve and initialize storage (see *STORE Statement*, p.392).

*A *pattern expression* has the syntax:
filename | { *expression* }

where *expression* is one of the following:

! *expression*
expression | *expression*
expression **&** *expression*
(*expression* **)**
filename

and *filename* can include the following special characters:

***** matches any string, including the null string.
? matches any single character.
[...**]** matches any one of the enclosed characters. A pair of characters separated by a comma denotes a range.

Note that any pattern more complex than * should be enclosed in double quotes. For example,

```
text_libimpfp.a (TEXT) :
    libimpfp.a[*] (.text)
```

will read in all **.tex**t sections from all object files beginning with *libimpfp.a*. To read in sections named **.text** and **.Text** from only those object files whose names begin with *libimpfp.asfpf*, use

```
text_libimpfp.a (TEXT) :
    libimpfp.a[“sfpf*”] (“.[Tt]ext”)
```

For more information, consult documentation on POSIX regular expressions.

The order of the sections listed in the *section-contents* is undefined as is the order of output sections in a **SECTIONS** command. A **GROUP** definition may be used to ensure the order of a set of output sections. (See *GROUP Definition*, p.392.)

NOTE: A section-contents specification must have at least one non-COMMENT input section, e.g., a BSS, CONST, DATA, or TEXT section, or the type of the output section will default to COMMENT, and it will not be allocated any memory. See below regarding section types.

Type Specification: ([=]BSS), ([=]COMMENT), ([=]CONST), ([=]DATA), ([=]TEXT), ([=]BTEXT); OVERLAY, NOLOAD

The *type-spec* clause sets the type of the output section. If absent, the type will be determined by the types of the input sections. If all input sections in a given output section are of the same type, the type of the output section will be that of the input sections and no *type-spec* clause is necessary. Mixing input sections of different types in a single output section is not recommended. If input sections do have different types, the linker will choose a type from the input sections in the following order from highest priority to least: **TEXT**, **CONST**, **DATA**, **BSS**, and **COMMENT**.

To force the linker to choose the specified type regardless of the types of the input sections, use the “**=**” form. For example, **(=DATA)** will force the output section to have the DATA type.

type-spec can also be used when linking files produced by third-party tools which do not tag each section with its type.

The alternative type specifications indicate the expected contents of the section:

(BSS)
Section contains uninitialized data space.

(COMMENT)
Section debug or other information not part of the program memory space.

(CONST)
Section contains initialized data space.

(DATA)
Section contains initialized variables.

(TEXT)
Section contains code and/or constants.

(BTEXT)
Blank text section.

OVERLAY tells the linker that the section can overlap other sections. The section should have **BIND** specification; memory is not allocated for it. Example:

```
.text1 (TEXT OVERLAY) BIND(ADDR(.text)) : { .... }
```

NOLOAD tells the linker not to mark the section as loadable.

Address Specification

The form of the *address-spec* is:

address-value | **BIND (** *expression* **)**

The *address-spec* clause specifies the address for the first byte of the output section. It is either an absolute address, *address-value*, or the word **BIND** followed by an expression that can contain the functions **SIZEOF**, **ADDR**, and **NEXT** (see *25.5 Expressions*, p.382). An *address-spec* is not allowed inside a **GROUP** (see *GROUP Definition*, p.392).

NOTE: A section with an address specification (address-spec) does not need a memory-area specification (area-spec), since the linker automatically marks the corresponding address range as reserved. If both an address-spec and an area-spec are provided, the linker checks that the address range is completely inside the memory area and displays a warning if it is not.

ALIGN Specification

The form of the *align-spec* is:

ALIGN (*expression* **)**

An *align-spec* clause causes the linker to align the section on the byte boundary given by the value of *expression*.

LOAD Specification

The form of the *load-spec* is:

LOAD (*expression* **)**

In a typical embedded system, the values for all variables with explicit initialization must be stored in some type of read-only memory before the system is "powered up". During execution, the variables must themselves be located in RAM so they can be set (except for **const** variables which can remain in ROM). Thus, during startup, the initial values for these variables must be copied from ROM to RAM.

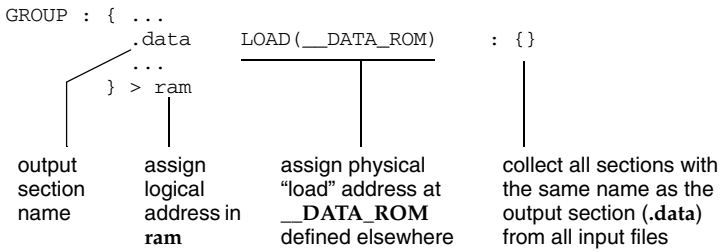
To distinguish these two locations, we refer to the *physical* and *logical* addresses of the output section.

- *physical address*: This is the address given by the *expression* in the *load-spec*. It is this address which is used in the section header when the section is written to the linked output file. Thus, if a dynamic loader loads the section, or the section data is burned into a ROM, it will be at this *physical* address.
- *logical address*: This address is set by an *address-spec* or an *area-spec* in the *section-definition*. This will be actual address of the section during execution. Thus, when linking references to a variable in the section, the linker will use the variable's *logical* address.

NOTE: The *load-spec* only controls the physical/logical addressing of the section. Typically, assignment statements are used to define symbols for the physical and logical addresses of the section and its length. These symbols are then used by startup code to copy the physical data from ROM to its logical location in RAM. See the examples in this chapter, as well as **default.dld** in the **conf** directory and **crt0.s** in the appropriate target directory for the startup copying code.

Also, copying code in the startup module, **init.c**, copies only a single contiguous physical section. Thus, while more than one **LOAD** specification is permitted, the output sections named in the *expressions* must be contiguous.

The following example is from Figure 25-1:



OVERFLOW Specification

The overflow specification enables you to specify the size limit of a section and to request that the linker place input sections which will not fit into the initial section into a different section, called the *overflow section*.

The form of the *overflow-spec* is:

OVERFLOW (*size-expression*, *overflow-section-name*)

The *size-expression* specifies the size of the initial section in bytes, and *overflow-section-name* names the section that is to receive the input sections that cannot fit into the initial section.

Fill Specification

The form of the *fill-spec* is

=*fill-value*

or

=(*fill-value*[**,** *size*[**,** *alignment*]])

The *fill-spec* instructs the linker to fill any holes in an output section with a two-byte pattern. A hole is created when an assignment statement is used to advance the location counter "**.**". The linker also creates holes to align input sections according to *alignment*. *size* and *alignment* are in bytes; valid values are 1, 2, and 4.

Area Specification

The form of the *area-spec* is

> *area-name*

where *area-name* is defined by an earlier **MEMORY** command (see *25.7 MEMORY Command*, p.384).

An *area-spec* causes the linker to locate the output section at the next available location in the given area (subject to any **ALIGN** clause, see *ALIGN Specification*, p.390).

STORE Statement

The **STORE** statement reserves and initializes memory space. Its form is:

STORE (*expression*, *size-in-bytes* **)**

where *expression* is the value to be stored at the current address, and *size-in-bytes* is the size of the storage area, normally 4 for 32-bit values. Example:

```
_ptr_to_main = .;
STORE(_main, 4)
```

will create a label **_ptr_to_main** that contains the 4-byte pointer to the label **_main**.

GROUP Definition

A **SECTIONS** command may contain *group-definitions* as well as *section-definitions* (see *25.8 SECTIONS Command*, p.384).

A group treats several output sections together and ensures they are located in a continuous memory block in the order given in the *group-definition*. When sections are not in a group, their order is not defined, although it may be dictated implicitly by, for example, *address-spec* clauses.

The full form of a *group-definition* is shown below. For clarity, each clause is written on a separate line and is identified to its right.

```
GROUP
    [ address-value | BIND ( expression ) ]                    address-spec
    [ ALIGN ( expression ) ]                                    align-spec
    :

    { section-definition ... }
    [ > area-name ]                                             area-spec
```

The clauses in a **GROUP** are defined above: *address-spec* in *Address Specification*, p.389, *align-spec* in *ALIGN Specification*, p.390, *section-definition* in *Section-Definition*, p.385, and *area-spec* in *Area Specification*, p.391.

NOTE: The *address-value* and **BIND** clauses may not be used on a *section-definition* inside a **GROUP**, only on the **GROUP** itself.

Both a *section-definition* and a *group-definition* can end with an *area-spec*. Usually when defining a group, an *area-spec* is used only on the *group-definition* and not on the *section-definitions* enclosed within it.

25.9 Assignment Command

An *assignment* command defines or redefines the value of a symbol. Assignment commands are allowed at the outer-most level of a linker command file, and as items in the *section-contents* of a *section-definition* (see *Section Contents*, p.386).

An assignment command may have either of the following forms:

symbol assign-operator expression ;
create an absolute symbol and assign it the value of *expression*

symbol @ {*section-name* | *symbol2* }*assign-operator expression* ;
create a symbol in the given section, or the same section as *symbol2*, and assign it the value of *expression*

where:

symbol and *symbol2*: an identifier following the rules of the C language with the addition of "**$**" and "**.**" as valid characters and limited to 1,000 characters.

assign-operator: one of

```
=     +=     -=     *=     /=
```

The assign"**;**" is required.

When the assignment is inside a *section-definition*, the special symbol "**.**" is allowed on either the left or right and refers to the current location counter.

A "hole" can be created in a section by incrementing the "**.**" symbol. If the *fill-spec* is used on the *section-definition*, the reserved space is filled with the *fill-value*.

Example: create a 100 byte gap in a section:

```
. += 100;
```

Example - define the beginning of the stack for use by initialization code:

```
__SP_INIT = ADDR(stack) + SIZEOF(stack);
```

25.10 Examples

Example 25-1 **Avoiding Long Command Lines**

A simple command file to avoid having to give a long command line when invoking the linker could look as follows:

```
main.o
load.o
read.o
arch.a
-m2
```

This means: load files **main.o**, **load.o** and **read.o**, search archive **arch.a**, and generate a detailed memory map.

The output sections for the above, not being defined in the command file itself, and absent **-Bd** and/or **-Bt** options on the command line, will be as described for these options (see *Set Address for Data and tExt (-Bd=address, -Bt=address)*, p.363), and using default addresses for each which are appropriate to the target.

Example 25-2 **Basic**

The command file:

```
MEMORY
{
    mem1 : origin = 0x2000, length = 0x4000
    mem2 : origin = 0x8000, length = 0xa000
}

SECTIONS
{
    .text : {} > mem2
    .data : {} > mem1
    .bss  : {} > mem1
}

_start_addr = start;
```

means that all **.text** sections are collected together and positioned in the memory area starting at 8000 hex. The sections **.data** and **.bss** are placed in order in the

mem1 area beginning at 2000 hex. The symbol **_start_addr** is defined to be the same as the address of the symbol **start** from one of the input files.

The input object files for the above linker command file are those given on the command line (and any others extracted from libraries to satisfy unresolved external symbols in those files).

Example 25-3 **Define a Symbol, Create a "Hole"**

The command file

```
SECTIONS
{
    .text : {}
    .data ALIGN(8) :
    {
        f1.o ( .data )
        _af1 = .;
        . = . + 2000;
        * ( .data )
    } = 0x1234
    .bss : {}

}
```

means first load the **.text** sections. Align on 8 and load the **.data** section from the file **f1.o**. Set the symbol **_af1** to the current address. Create a hole in the output section with a size of 2000 decimal bytes. Load the rest of the **.data** sections from the files given on the command line. Fill the hole with the value 0x1234. Load the **.bss** sections thereafter.

Example 25-4 **Groups**

The command file

```
MEMORY
{
    a: org = 0x100a8, len = 0x7ffeff58
}

SECTIONS
{
    .text BIND((0x10000 + HEADERSZ+7) & (~7)) :
    {
        *(.init) *(.text)
    }
```

```
    GROUP BIND(NEXT(0x10000) +
        ((ADDR(.text) + SIZEOF(.text)) % 0x2000)) :
    {
        .data : {}
         .bss : {}
    }
}
```

means that all input sections called **.init** or **.text** are combined into the output section **.text**. This output section is allocated at the address "0x10000 + size of all headers aligned on 8".

If **HEADERSZ** is 0xe0, the address becomes 0x100e0.

The sections **.data** and **.bss** are grouped together and put at the next multiple of 0x10000 added to the remainder of the end address of **.text** divided by 0x2000.

If **.text** is 0x23450 bytes long, the values are defined to be:

```
NEXT(0x10000) = 0x40000
ADDR(.text) = 0x100e0
SIZEOF(.text) = 0x23450
(ADDR(.text)+SIZEOF(.text))%0x2000 = 0x01530
address of .data = 0x41530
```

This is a typical default algorithm in a paged system where it is important to align the section addresses on the file-offset in the executable file.

Example 25-5 **Document With C-Style Comments**

The following command file is documented with C-style comments.

```
/*
 * The following section defines two memory areas:
 * one 1 MB RAM area starting at address 0
 * one 1 MB ROM area starting at address 0x1000000
 */
MEMORY
{
    ram: org = 0x0, len = 0x100000
    rom: org = 0x1000000, len = 0x100000
}

/*
 * The following section defines where to put the
 * different input sections. .text contains all
 * code + optionally strings and constant data, .data
 * contains initialized data, and .bss contains
 * uninitialized data.
 */
```

```
SECTIONS
{
    /* Allocate code in the ROM area. */

    .text : {} > rom

    /*
     * Allocate data in the RAM area.
     * Initialized data is actually put at the end of the
     * .text section with the LOAD specification.
     */
    GROUP : {
        .data LOAD(ADDR(.text)+SIZEOF(.text)) : {}
        .bss : {}
    } > ram
}
```

Note the use of the **LOAD** clause to allocate the **.data** section to a physical address in ROM, after the **.text** section, while the logical address (the address used during execution) is in the RAM. The initialized data in **.data** has to be moved from the physical address to the logical address during start up.

Example 25-6 **Empty Sections**

It may be an error to define a section without any input sections. This extended example begins with a sample linker command file extract likely to be faulty, and then discusses some potential workarounds. Recommended solutions are at the end of the example. While some of the workarounds are not recommended, they serve to illustrate a number of principles in linker command file construction.

Consider the following example:

```
SECTIONS
{
    ...
    .stack : {
        stack_start = .;
        stack_end   = stack_start + 0x10000;
    } > ram
    ...
}
```

The above is apparently intended to reserve space for a stack and to define symbols marking its beginning and end.

There are four potential problems:

- The address of the current location, "**.**", and therefore of **stack_start**, is not well-defined. If there are no input sections named **.stack** in the input files, then **stack_start** will be at the "next" unfilled location in **ram**, or at the beginning of

the **ram** memory area if no other commands directing output to **ram** precede the above **.stack** output section definition.

However, if **.stack** sections do appear in the input files, these will be automatically included in this **.stack** output section — but whether they will appear before or after the address given to **stack_start** is undefined (the rules are complex and subject to change, so no guarantee of order is made for this poorly constrained case).

If **.stack** sections do appear in the input files, the definition of "." and therefore of **.stack_start** can be made well defined by adding an input section specification as follows:

```
.stack ALIGN(4) : {
    stack_start = .;
    *(.stack)
    stack_end   = .;
} > ram
```

- **stack_start** may not be aligned as required. Lacking an *align-spec* as in the case above, the alignment will be 1, which may not be valid if the **.stack** section definition is preceded by a section with, for example, an odd length.

  This problem could be solved by providing an *align-spec*:

  ```
  .stack ALIGN(4) : { ... }
  ```

- The assignment to **stack_end** will as expected define it to be **stack_start** plus 0x10000 bytes, *but this assignment in and of itself does not allocate/reserve memory.* If other section definitions result in object bytes in what is intended to be the stack area, the linker will not warn of the conflict.

  This problem could be solved by incrementing the current location:

  ```
  stack_start = .;
  . += 0x10000;
  stack_end = .;
  ```

  Incrementing "**.**" creates a "hole". The hole will be zero-filled (absent specification of a different constant with option **-f** — see *Specify "fill" Value (-f value, size, alignment)*, p.364).

  A reminder: the current location symbol, "**.**", may appear only in a **SECTIONS** command, either between section definitions, or within a *section-definition* (*Section-Definition*, p.385) or a *group-definition* (*GROUP Definition*, p.392).

- Creating a hole by incrementing "**.**" actually uses space in the output image (which could be more of an issue with larger stack). If the area reserved for the stack is expected to be 0, this unnecessary space in the output image can be

eliminated by a **BSS** *type-spec* (*Type Specification: ([=]BSS), ([=]COMMENT), ([=]CONST), ([=]DATA), ([=]TEXT), ([=]BTEXT); OVERLAY, NOLOAD*, p.388):

```
.stack (BSS) ALIGN(4) : { ... }
```

Combining all of the above, the following is at least valid and likely to produce an acceptable result if there are no **.stack** sections in input files.

```
SECTIONS
{
    ...
    .stack (BSS) ALIGN(4): {
        stack_start = .;
        . += 0x10000;
        stack_end   = .;
    } > ram
    ...
}
```

However, because of its potential problems as described in this example, this approach is not recommend. A recommended way to define a stack, especially in combination with a heap, is to use **GROUP** definitions to locate sections in the desired order, and then to define a stack and heap from the end of the final **GROUP** (using assignment commands as above). Another way is to define a separate memory area for the heap or stack with the **MEMORY** command. These approaches are combined in the **default.dld** linker command file. See *25. Linker Command Language* for details.

Example 25-7 **Right and Wrong Ways to Use SIZEOF**

Adding the size of a section to its address is *not* a reliable way to calculate the address of the next section to follow because there may be an alignment gap between the sections. For example, the following figure shows incorrect and correct ways to define the physical address in a **LOAD** specification and to define a heap symbol. Incorrect commands in the incorrect method and changes in the correct method are in bold.

Figure 25-3 **Correct and Incorrect Use of SIZEOF**

```
MEMORY                              (Used by both incorrect and correct examples.)
{
    rom1:   org = 0x20000, len = 0x10000  /* 3rd 64KB      */
    rom2:   org = 0x30000, len = 0x10000  /* 4th 64KB      */
    ram:    org = 0x80000, len = 0x30000  /* 512KB - 703KB */
    stack:  org = 0xb0000, len = 0x10000  /* 7043B - 768KB */
}
```

Incorrect LOAD Specification and Symbol Definition Using SIZEOF

```
SECTIONS
{
    GROUP : {
        .text : { *(.text) *(.init) *(.fini) }
        .ctors ALIGN(4):{ ctordtor.o(.ctors) *(.ctors) }
        .dtors ALIGN(4):{ ctordtor.o(.dtors) *(.dtors) }
    } > rom1

    .text2 : { *(.text2) } > rom2

    GROUP : {
        .data  LOAD(ADDR(.text2) + SIZEOF(.text2)) : {}
        .bss : {}
    } > ram
...

__HEAP_START    = ADDR(.bss  ) + SIZEOF(.bss  );    (Alignment gap after .bss could
                                                    make __HEAP_START wrong.)

__HEAP_END      = ADDR(ram   ) + SIZEOF(ram   );    (Memory areas are fixed size;
                                                    SIZEOF use is correct.)
```

Figure 25-3 **Correct and Incorrect Use of SIZEOF** (cont'd)

Corrected

```
SECTIONS
{
    GROUP : {
        .text : { *(.text) *(.init) *(.fini) }
        .ctors ALIGN(4):{ ctordtor.o(.ctors) *(.ctors) }
        .dtors ALIGN(4):{ ctordtor.o(.dtors) *(.dtors) }
    } > rom1

    .text2 : { *(.text2) } > rom2

    __DATA_ROM= .;                              (Define symbol for use in LOAD.)

    } > rom2


    GROUP : {
        .data LOAD(__DATA_ROM) : {}
        .bss : {}
    } > ram
...

__HEAP_END = ADDR(ram   ) + SIZEOF(ram   );       Memory areas are fixed size;
__SP_INIT  = ADDR(stack ) + SIZEOF(stack );       SIZEOF use is correct.)
__SP_END   = ADDR(stack );
```

Example 25-8 **Copying Code from "ROM" to "RAM"**

In embedded systems, code and data are typically burned into a ROM-type device, and then initial values for global and static variables are copied to RAM during system startup. The startup code can automatically copy such initial values as described in *Copying Initial Values From "ROM" to "RAM", Initializing .bss*, p.257, which makes reference to the linker **LOAD** specification. (See *LOAD Specification*, p.390.)

Copying code, not just initial data values, to high speed RAM can increase performance because it can be much faster to access than ROM. This example shows how to modify a simplified version the *version_path***/conf/sample.dld** file shipped with the compiler suite to support this. In addition, a new **copy_to_ram( )** function is required, and **crt0.s** is modified to call it.

This example assumes an understanding of the startup code and the **LOAD** specification referred to above.

The first part of this discussion describes changes that are made to the linker command file. The following **SECTIONS** directive can be used to locate code physically in ROM but logically in RAM:

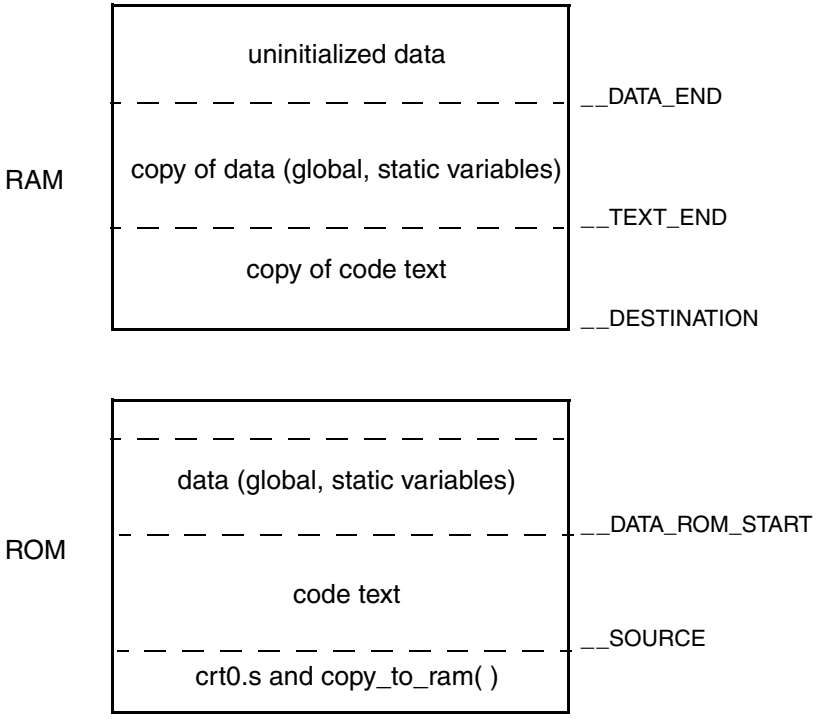
```
SECTIONS
{
    .text LOAD (ROM_ADDRESS) :{}
} > ram
```

The **LOAD** instruction tells the linker where code is to be loaded in ROM at load time — the *physical* address (for example, when the PROM is burned). The area specification (the **> ram** part of the statement) tells the linker where the code will be during execution — the *logical* address. Note that this **SECTIONS** directive does not copy the data from ROM to RAM; it only tells the linker where to resolve references to functions, labels, string constants located with code, and so forth. In this example a user-supplied function called **copy_to_ram( )** does the actual copying of code from ROM to RAM during system startup.

If a **LOAD** directive and an area specification such as those shown above are used for the *initialization code*, that code will not be accessible. This is because the linker would resolve references to the initialization code in the **ram** area, and so the initialization code would never be found. One solution to this "chicken and egg" problem is to refrain from copying the initialization code, **crt0.o** and **copy_to_ram( )**, to RAM, leaving it in ROM.

Here are the details:

1. Locate initialization code into ROM only, in a section called .**startup**. The startup code consists of **crt0.o** and **copy_to_ram( )**.
2. Locate the rest of the code, and all global and static variables, physically in ROM but logically in RAM, except for uninitialized variables, which is only placed in RAM.
3. Assign symbols to keep track of important addresses in RAM and ROM. See the diagram below.



The symbols **__SOURCE** (in ROM) and **__DESTINATION** (in RAM) mark the beginning of the code areas (not including the initialization code). **__DATA_ROM_START** marks the beginning of data in ROM, and **__TEXT_END** marks the end of the **.text** section in RAM. **__DATA_END** marks the end of the code and variable sections that are to be copied.

The next two pages show the simplified **sample.dld**, before and after changes are made. Comments have been reduced to improve readability and unnecessary details have been omitted; changes appear in bold text in the second version of **sample.dld**. See **bubble.dld** for another example of more complete linker command files in *25. Linker Command Language*.

In the "after" linker command file (Figure 25-5), note that **__DATA_ROM** and **__DATA_RAM** are made equal to each other in order to prevent **crt0.o** from redundantly copying data. (**crt0.o** copies data from ROM to RAM if those symbols are not equal; see *Copying Initial Values From "ROM" to "RAM", Initializing .bss*, p.257.)

Figure 25-4 **sample.dld As It Is Distributed**

```
MEMORY                                          Specify memory layout.
{
  rom:    org=0x0,      len=0x100000
  ram:    org=0x100000, len=0x100000
  stack:  org=0x300000, len=0x100000
}

SECTIONS                                        The first GROUP contains code and constant
{                                               data, and is allocated in the rom memory area.
  GROUP :
  {
    .text (TEXT)  :{
      *(.text) *(.rodata) *(.rdata)
      *(.frame_info) *(.j_class_table)
      *(.init) *(.fini)
    .ctors ALIGN(4):{ ctordtor.o(.ctors)
        *(.ctors) }
    .dtors ALIGN(4):{ ctordtor.o(.dtors)
        *(.dtors) }
    }

    __DATA_ROM  = .;
  } > rom

  GROUP : {                                     The second GROUP allocates space for
    __DATA_RAM  = .;                            initialized and uninitialized data in the ram
                                                memory area, as directed by > ram at the end of
                                                the GROUP. This is the "logical" location;
                                                references to symbols in the GROUP are to
                                                ram.

    .data (DATA) LOAD(__DATA_ROM) :             But the LOAD specification on the .data output
        { *(.data) *(.j_pdata) }                section causes that section to follow be at
                                                __DATA_ROM in the GROUP above in the
                                                actual image (the "physical" address).

    __DATA_END  = .;

    __BSS_START  = .;                           Allocate uninitialized sections.
    .bss  (BSS)  : {}
    __BSS_END  = .;

    __HEAP_START= .;
  } > ram
}
```

Figure 25-5 **sample.dld Highlighting Changes Made for Copying from ROM to RAM**

```
MEMORY { ... }

SECTIONS
{
  .startup (TEXT) : {
    crt0.o(.text)
    *(.startup)
    __SOURCE = (. + 3) & ~3;
  } > rom

  GROUP :
  {
    __DESTINATION = .;
    .text (TEXT) LOAD(__SOURCE) : {
        *(.text) ...
    }


    __TEXT_END = .;
    __DATA_ROM_START = __SOURCE +
        __TEXT_END - __DESTINATION;

    .data (DATA) LOAD(__DATA_ROM_START) :
        { *(.data) *(.j_pdata) }

    __DATA_END = .;

    __BSS_START  = .;
    .bss  (BSS)  : {}
    __BSS_END  = .;

    __HEAP_START  = .;
  } > ram
}

__DATA_ROM   = 0;
__DATA_RAM   = __DATA_ROM;
```

Create a startup section for initialization code, **crt0.o** and **copy_to_ram( )**, that will only be placed in ROM. **__SOURCE** is the beginning address for the ROM to RAM copy.

Make sure **__SOURCE** is aligned.

Combine the rest of the code and data into a group located in RAM. Use **LOAD** directives to place all of this group (except uninitialized data) in ROM. **__DESTINATION** is the address in RAM for the ROM-to-RAM copy. Some details (such as **.ctors** and **.dtors**) have been removed.

__TEXT_END marks the end of code.

__DATA_ROM_START marks the beginning of data in ROM.

__DATA_END marks the end of data to be copied.

Allocate uninitialized sections.

Make **__DATA_ROM** and **__DATA_RAM** equal so initialization code will not copy initial values from ROM to RAM.

A simple copy program can be used to copy from ROM to RAM, using **__DATA_END** and **__DESTINATION** to calculate the number of bytes to copy.

```
/* These symbols are defined in a linker command file. */

extern int __SOURCE[], __DESTINATION[], __DATA_END[];

#pragma section CODE ".startup"

void copy_to_ram(void) {
```

```
    unsigned int  i;
    unsigned int  n;
    /* Calculate length of the region in ints */
    n = __DATA_END - __DESTINATION;

    for (i = 0; i < n; i++) {
        __DESTINATION[i] = __SOURCE[i];
    }
}
```

crt0.s must call **copy_to_ram( )**. The following is added after the comment "insert other initialization code here," before calling **__init_main( )**.

```
bl      copy_to_ram
```

NOTE: An alternative to using **copy_to_ram( )**, which is implemented with a **for** loop, would be to call **memcpy( )** from **crt0.o**, but then **memcpy( )** would remain in ROM, with its slow access.

PART V

Wind River Compiler Utilities

26 Utilities

The following chapters describe utility tools that accompany the compiler suite.

26.1 Common Command-Line Options

All tools in the Wind River suite accept the following command-line options where meaningful. They are repeated here for convenience.

Show Option Summary (-?)

-?, -h,
--help
Show synopsis of command-line options.

Read Command-Line Options from File or Variable (-@name, -@@name)

-@*name*
Read command-line options from either a file or an environment variable. When **-@***name* is encountered on the command line, the tool first looks for an environment variable with the given *name* and substitutes its value. If an environment variable is not found then it 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 the tool terminates.

-@@*name*
Same as **-@***name*; also prints all command-line options on standard output.

Redirect Output (-@E=file, -@E+file, -@O=file, -@O+file)

-@E=*file*
Redirect any output to standard error to the given file.

-@O=*file*
Redirect any output to standard output to the given file.

Use of "**+**" instead of "**=**" will append the output to the file.

27 D-AR Archiver

27.1 Synopsis

Create and maintain an archive of files of any type, with special features for object files.

27.2 Syntax

dar *command* [*position-name*] *archive-file* [*name*] ...

27.3 Description

The **dar** command maintains files in an archive. Archives can contain files of any kind. However, object files are handled in a special way. If any of the included files is an object 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.

NOTE: An archive file consisting only of object files is also called a library, and so the archiver is often referred to as a *librarian*.

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-file is the archive file pathname.

name is one or more files in the archive. Multiple *name* arguments are separated by whitespace.

27.3.1 dar Commands

dar commands and modifiers are as follows. Modifiers are shown in brackets. See also *26.1 Common Command-Line Options*, p.409.

-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 append the named files at the end of the archive without checking whether the files already exists. If the archive contains any object 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.

-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. Used 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 27-1 **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*. Same as "**i**" modifier.
c	**-q -r**	Does not display any message when a new archive *archive-file* is created.

Table 27-1 **dar Command Modifiers** (cont'd)

	Use With Commands	
D *pathname*		
	-q -r	When adding to or replacing files in an archive, prefix *pathname* to name of each file to be stored to access it in the file system (but do not store the additional *pathname* in the symbol table).
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*. Same as "**b**" modifier.
j	**-q -r**	Store a path prefix if given with an object file in the archive symbol table instead of just the base filename. **NOTE**: The path prefix becomes part of the name in the archive. Thus, if a single file **x.o** is added once as **x.o** and a second time as **lib/x.o** using the "**j**" option, it will be stored twice in the archive.
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 object files in the archive so that the linker does not have to scan the symbol table in multiple passes.

27.4 Examples

Some later examples build on earlier examples.

Example 27-1 **New Archive**

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
```

Example 27-2 **Modify Above Archive: Replace File, Add File**

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
```

Example 27-3 **Alternative command for Example 2**

Example 27-1 - Example 27-2 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 *27.3.1 dar Commands*, p.412.

```
dar -r -a -v f.o lib.a f.o g.o
```

Example 27-4 **Quick Append to Archive**

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
```

Example 27-5 **Cleanup Archive After Quick Appends**

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
```

Example 27-6 **Extract File from Archive Without Changing Archive**

Extract **file.c** from archive **source.a** and place it in the current directory. The archive is unchanged.

```
dar -x source.a file.c
```

Example 27-7 **Delete File from Archive Permanently**

Delete **file.c** files from archive **source.a**. The file is deleted without being written anywhere:

```
dar -d source.a file.c
```

28

D-BCNT Profiling Basic Block Counter

28.1 Synopsis

Display profile data collected from one or more runs of a program.

28.2 Syntax

dbcnt [**-f** *profile-file*] [**-h** *n*] [**-l** *n*] [**-n**] [**-t** *n*] *source-file*, ...

28.3 Description

The **dbcnt** command displays the number of times each line in a source program has been executed. It can also be used to show "coverage" information (see *28.6 Coverage*, p.420).

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.

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 pathname of the actual file with the line counting information. See below for examples.

dbcnt options are as follows. See also *26.1 Common Command-Line Options*, p.409.

28.3.1 dbcnt Options

-f *file*
Read profile data from *file* 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**.

28.4 Files

NOTE: Files processed by **dbcnt** must be unique in their first 16 characters.

28.4.1 Output File for Profile Data

dbcnt.out
Default output file for profile data.

DBCNT
Environment variable giving the name of the profile data file.

28.5 Examples

The file *file.c* (shown annotated below) 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);
     }
```

NOTE: 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.

The following will find the 100 most frequently executed source lines in a program:

```
dbcnt -n -t100 *.c
```

28.6 Coverage

The following will find all source lines which did not execute in a program:

```
dbcnt -h0 -l0 -n *.c
```

(The second option, **-l0**, is hyphen, lower-case L, 0.)

28.7 Notes

The functions **__dbinic( )** 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.

For information on support for file I/O and environment variables in an embedded environment, see *15.8.2 File I/O*, p.265 and *15.11 Target Program Arguments, Environment Variables, and Predefined Files*, p.270.

See *15.12 Profiling in An Embedded Environment*, p.272 for an additional example.

29

D-DUMP File Dumper

29.1 Synopsis

Dump or convert all or parts of object files and archive files.

29.2 Syntax

ddump [*command*] [*modifiers*] *file*, ...

29.3 Description

An 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 is processed by the **ddump** command. **ddump** can generate debugging information only for code that is fully bound at link time; it does not work on relocatable object files.

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., **-Rv** versus **-R -o** *name*).

See also *26.1 Common Command-Line Options*, p.409.

29.3.1 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. Whitespace (spaces, tabs, and newlines) are ignored. Unless the **-o** modifier is used, the output file will be named **bin.out**.

-C
Generate a difference file (either a SingleStep **.blk** file or an S-Record) from two ELF executable files. Usage:

`ddump -C` [*modifiers*] *file1 file2*

The following special modifiers are available:

-h
Generate differences for read-only sections and a complete dump for writable sections. Useful when the original executable has already run on the target and has modified some writable information.

-v
Generate differences for initialized sections. Useful when the executable has initialized uninitialized data.

-p2
Generate an S-Record instead of a **.blk** file.

-c
Dump the string table in each object file.

-D
Dump the DWARF debugging information in each object file.

-F
Demangle C++ names entered interactively, one per line (no files are processed). Enter **Ctrl-C** or the end-of-file character to terminate interactive mode. If combined with other options, prints demangled names. See *13.5 C++ Name Mangling*, p.225 for details on how names are mangled.

-f
Dump the file header in each object file.

-g
Dump the symbols in the global symbol table in each 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.

-l
Dump the line number information in each object file.

-N
Dump the symbol table information in each object file. Similar to the UNIX **nm** command. The following special modifiers are available. See also the **-t** option below for a more readable dump but without further options.

-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.

-g
Emulate GNU **nm** output.

-o
Dump the optional header in each object file.

NOTE: **-o** is both a command and an option. If any of the commands **-B**, **-I**, or **-R** are encountered, then a following **-o** is assumed to specify the output file for the **-B**, **-I**, or -**R** command. If **-o** is encountered first, then it is the command. See the **-o** modifier on *29.3.1 ddump commands*, p.422.

-R
Convert an executable (usually, or object) file to different formats, especially Motorola S-Record format. The output file will be named **srec.out** unless the **-o** modifier is used (see *29.3.1 ddump commands*, p.422). Sections may be selected with the **-n** or **-d** and **+d** modifiers as usual.

The following special modifiers are available:

-m*t*
Write S-Records of the given type: 1 for 16-bit addresses, 2 for 24 bit-addresses, 3 for 32-bit addresses (the default). No space is permitted between "**m**" and *t*.

-p
Write a plain ASCII file in hexadecimal (not S-Record format).

-u
Write a binary file (not S-Record format). Inter-section gaps of size less than or equal to 10KB are filled with 0. The size may be changed with the **-y** option described in *29.3.1 ddump commands*, p.422. A larger gap will cause an error.

-v
Do not output the **.bss** or **.sbss** section (applies to all output formats).

Without **-v**, S-Records will be generated to set **.bss** and **.sbss** sections to 0. This will increase transmission or programming time when sending S-Records to PROM programmers or other devices and may not be desirable.

-w*n*
Set the line width of the S-records to represent *n* data characters. The actual line length is 2*n* plus the size of other fields such as the address field. The default value of *n* is 20. 2*n* is used instead of n because it takes 2*n* hex digits to represent *n* characters. No space is permitted between "**w**" and *n*.

-r
Dump the relocation information in each object file.

-S

Display the size of the sections. Similar to the UNIX **size** command. By using the **-f** modifier, the section names will be included in the output. By default, only the **.text**, **.data**, and **.bss** sections will be included. By using the **-v** modifier, all sections will be included.

-s
Dump the section contents in each object file.

NOTE: Use of the **v** modifier, that is, **-sv**, is highly recommended.

-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 29-1 **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 *namelist*	**-h -l -R -s -t**	Dump the information associated with each section name in a comma-separated list of section names.
-o *name*	**-I -R**	Specify an output filename for the **-B**, **-I**, and **-R** commands. (See note regarding the -o command in *29.3.1 ddump commands*, p.422.)
-p	any but **-I**	Suppress printing of headers. 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.
-u	any	Underline filenames. Special meaning with **-R**.
-v	any	Dump information in verbose mode. Special meaning with **-R**.
-y*n*	**-Ru**	Change the size of the gap zero-filled by the **-Ru** command to *n* (see *29.3.1 ddump commands*, p.422). For example: `ddump -Ru -y20000 ...` will permit gaps from 1 through 20,000 bytes.

29.4 Examples

Example 29-1 **Dump File Header and Symbol Table for Files in Archive**

Dump the file header and symbol table from each object file in an archive in verbose mode:

```
ddump -ftv lib.a
```

Example 29-2 **Convert Executable File to Motorola S-Records**

Convert an executable file named **test.out** to Motorola S-Record format, naming the output file **test.rom**. Use the **-v** option to suppress the **.bss** section (without **-v**, S-Records would be generated to fill the **.bss** section with zeros).

```
ddump -Rv -o test.rom test.out
```

Example 29-3 **Generate S-Records Only for "data" Sections**

Same as the prior example but convert and output only section **.data** and call the result **data.rom**.

```
ddump -R -n .data, -o data.rom test.out
```

Example 29-4 **Display Section Sizes**

Use **-Sf** to show the size of all sections loaded on the target. See below:

```
ddump -Sf a.out
9056(.text+.sdata2) + 772(.data+.sdata) + 428(.sbss+.bss) =
    10256
```

Example 29-5 **Demangle C++ Names**

Demangle C++ names with **ddump -F**:

```
ddump -F                    command entry
mymain__FiPPc               user entry
mymain(int , char **)       demangled result
init__7myclassFv            user entry
myclass::init(void )        demangled result
```

30 dmake Makefile Utility

30.1 Introduction

Rebuilding the Wind River libraries requires the special make utility, **dmake**, by Dennis Vadura. **dmake** is shipped and installed automatically with the tools.

dmake supports the standard set of basic rules and features supported by most "make" utilities — see the documentation for other "make" utilities for details.

30.2 Installation

The **dmake** executable is shipped in the **bin** directory and requires no special installation.

30.3 Using dmake

Use **dmake** as a typical "make" utility. For example, enter **dmake** without parameters to cause it to look for a makefile named, on Windows, **makefile** (case-insensitive), and on UNIX, first **makefile** and then **Makefile**.

Enter **dmake -h** for a list of command-line options.

dmake requires a "startup" file unless the **-r** option is given on the command line, and will look for the file in the following locations in order:

- The value of the macro **MAKESTARTUP** if defined on the command line.
- The value of the **MAKESTARTUP** environment variable if defined.
- The file *version_path***/dmake/startup.mk** (supplied as shipped).

31

WindISS Simulator and Disassembler

31.1 Synopsis

WindISS, the Wind River Instruction Set Simulator, is a simulator for executables and a disassembler for object files and executables. The disassembler mode provides both batch and interactive disassembly. The three modes of operation are selected by:

`windiss` ...
: Simulation (with no **-i** option).

`windiss -i` ...
: Batch disassembly.

`windiss -ir` ...
: Interactive disassembly.

The modes of operation are described the next three sections.

31.2 Simulator Mode

In simulator mode, **windiss** can take command-line arguments, input from standard input, and send output to standard output.

Table 31-1 **Syntax (Simulator Mode)**

`windiss`	[**-b** *binary-offset*]	Load file at address; requires -t option.
	[**-d** *debug-mask*]	Write debugging information.
	[**-D**]	Trace execution, show disassembly and register state.
	[**-Df** *trace-file*]	Send -D... trace output to file.
	[**-Di** *trigger-address*[. . *stop-address*] [, *trace-count*]]	Trace only on execution in address range; trace for count instructions.
	[**-Dm** *range-start*[. . *range-stop*] [, *trace-count*]]	Start trace on first read/write in address range. trace for count instructions.
	[**-Ds** *skip-count*[, *trace-count*]]	Start trace after *skip-count* instructions; trace for count.
	[**-Dx** *max-count*]	Execute *max-count* instructions, then stop.
	[**-e** *entry-point*]	Set entry point address.
	[**-h** *hex-offset*]	Load at offset; requires **-t** option.
	[**-I** *mem-init-value*]	Initialize memory to low byte of value, else to 0.
	[**-m** *mem-size*]	Set memory size in bytes; suffixes K (kilo) or M (mega).
	[**-ma**]	Allocate memory automatically when accessed.
	[**-mm** *range-start*[. . *range-end*] [**r**][**w**][**x**] [, *range-start*[. . *range-end*] [**r**][**w**][**x**]] [, . . .]]	Specify memory map in address range(s); **r**, **w**, and **x** set memory type to read, write, and execute.
	[**-M** *address-mask*]	Specify address mask applied to simulated target.
	[**-N** *nice-value*]	Run with lower priority on windows; *nice-value* can be 0 (default) to 6 (lowest priority).

Table 31-1 **Syntax (Simulator Mode)**

[**-p**]	Generate count profile without using **-Xprof...** options.
[**-q**]	Quiet mode — no messages except user output.
[**-r**]	Internal use by RTA.
[**-s** *clock-speed*]	Set clock speed (in megahertz).
[**-S** *stack-address*]	Specify initial value of stack and environment area.
[**-t** *target-name*]	Set target. *target-name* may be set to ARM.
[**-v**]	Display version number.
[**-x** *exception-mask*]	Set exception mask.
filename [*argument...*]	Executable file to simulate and arguments to it if any.

31.2.1 Compiling for the WindISS Simulator

The simulator is easiest to use with ELF files that were compiled for the **windiss** environment, without hardware floating point support (which **windiss** does not provide). To select the **windiss** environment when compiling, assembling, and linking, either:

- Use **-t***of***:windiss** on the compiler, assembler, or linker command line.
- Use **dctrl -t** to specify the target and environment. When **dctrl** prompts **Select environment**, select **other**, and then enter **windiss**.

If object files were not compiled with ELF object file coding, the linker option **-Xelf** can be used to produce ELF file executables. Also, special switches described below allow for simulation using binary and hex files.

31.2.2 Simulator Mode Command and Options

The following shows options for running **windiss** in simulator mode. The space between the option and its value is optional unless otherwise noted. When an option has multiple values, no other spaces are allowed. All numeric values may be specified in decimal or hex, e.g., 16 or 0x10.

-b *address*

Load binary file at *address*. The **-t** option must be used to indicate the target.

-d *debug-mask*

Write debugging information using *debug-mask* to indicate options. Mask bits may be ORed and are specified in hex, e.g. 0xc. Mask bits not listed below are reserved. The mask bits are as follows:

1, 2	Turn logging on for the RTA server. Bit 2 requests more detail than bit 1.
4	Cannot be used without bit 8. When used with bit 8, **windiss** displays the contents of buffers for POSIX calls.
8	Log POSIX calls.
16	Log exceptions, if exceptions are enabled. For example, the timer interrupt can be logged.
64	Log target memory handling.

-D

Show initial register state; trace execution, showing disassembly for all instructions; show values for all registers that are changed.

-Df ⊥*trace-file*

Direct output from all **-D** tracing options (**-D, -Di**, **-Dm**, and **-Ds**) to the *trace-file*.

-Di *trigger-address*
[**..** *stop-address*]
[**,** *trace-count*]

Enable tracing, displaying each instruction as it executes and any registers modified by it on **stdout**. No space is allowed in the arguments except after **-Di**.

Start tracing when the PC enters the range from *trigger-address***..***stop-address*. The default for *stop-address* gives a range of one instruction at the *trigger-address*.

Addresses may be symbols.

Stop tracing when execution reaches the *stop-address* or after *trace-count* instructions. If neither is present, tracing continues until the program terminates. Note that the program does not terminate when tracing stops — the program always runs until completion unless the **-Dx** option is present.

If *trace-count* is 0, tracing is enabled as long as the PC is within the specified function or range. When the PC is outside of range (e.g. when executing a subroutine), tracing is disabled.

Program output to **stdout** is intermixed with trace output unless the **-Df** option is used to redirect trace output to a different file. Examples:

windiss -Di main hello.out
Trace beginning at **main**.

windiss -Di main,1 hello.out
Trace one instruction beginning at **main**.

windiss -Di main..printf hello.out
Trace from **main** through the first entry to **printf**.

windiss -Di printf,0 hello.out
Trace **printf**, skipping subroutine calls.

Note: simulation is slower with this option.

-Dm *range-start* [. . *range-stop*] [, *trace-count*]
Start tracing on the first read or write to any memory location in the given range. Stop tracing after *trace-count* instructions if present.

See **-Di** for other details and related examples.

-Ds *skip-count*[, *trace-count*]
Execute at full speed until *skip-count* instructions have been executed and then begin tracing each instruction as executed. Stop tracing after *trace-count* instructions if present.

See **-Di** for other details and related examples.

-Dx *max-count*
Execute *max-count* instructions and then stop.

-e *entry-point*
Specify the entry point of binary file.

-El
-Eb
Specify endianity for a binary file: **-Eb** for big-endian, or **-El** for little-endian.

-h *address*
Load hex file at *address*. The **-t** option must be used to indicate the target.

-I *mem-init-value*
Initialize memory to the low-order byte of the given value. Memory is cleared to 0 without this option.

-m *mem-size*
Specify size of memory in simulator. Sizes can be specified in bytes, kilobytes with "**k**" or "**K**", or megabytes with "**m**" or "**M**". For example, the following are equivalent: **-m 2M**, **-m 2048K**, **-m 2097152**, and **-m 0x200000**. The program terminates with an error if the end of memory is reached.

-ma
Use automatic memory allocation. Memory is allocated when accessed.

-mm *range-start*[. . *range-end*] [**r**][**w**][**x**] [, *range-start*[. . *range-end*] [**r**][**w**][**x**]] [, ...]
Specify a memory map starting at *range-start* and ending at *range-end*. The **r**, **w**, and **x** flags set the memory type to read, write, and execute; the default is **rwx**. Multiple ranges can be specified.

-M *memory-mask*
Specify an address mask to be applied to all target addresses before access to the simulated memory. Used to mask off high address bits to fit applications linked to high memory.

-N *nice-value*
Run **windiss** using lower priority on windows. *nice-value* can be 0 to 6, where 0 is the default (normal execution) and 6 is the lowest priority.

(none) or **-?**
Use **windiss** alone on the command line to see a list of **windiss** options.

-p
Generate count profile data even for programs not compiled with **-Xprof-...** options, effectively using **-Xprof-count** (100; hierarchical profile data not available). Without **-r,** upon program completion, the profile data is written to **stdout**. With **-r**, the **RTA** collects the profile data.

-q
Run in quiet mode: do not print messages other than output from the user's program.

-r
Not for direct use. Used for connection to the RTA.

-s *clock-speed*
Set simulated clock speed in megahertz. The default is 10 (MHz). *clock-speed* must be an integer. This does not change the execution speed of **windiss** itself; rather, it changes the simulated time reported by **windiss**.

-S *stack-address*
Specify the initial value of the stack and environment area. The default is to use the highest available memory address, or 0x80000000 if automatic memory allocation is used (see **-ma** above).

-t *target-name*
Specify target processor for program. Not needed for ELF files. Abbreviated names are used for specifying target processors: ARM, M32R, MC68K, MCF (for ColdFire), MCORE, MIPS, NEC, PPC, SH, SPARC, and X86. (Note that these abbreviated names are only the initial part of the *t* component of the

-t*tof*:*environ* option to the compiler, linker and assembler. Only the abbreviated forms shown are currently permitted with **windiss**.)

-v
Print **windiss** version.

-x *exception-mask*
Not implemented at this time for ARM microprocessors.

31.3 Batch Disassembler Mode

31.3.1 Syntax (Disassembler Mode)

windiss -i[**o** | **e** | **l**] [*label*] [**-R1** *start-address* [**-R2** *end-address*]] [**-R3** *section*] *filename*

label is used only with the **l** modifier.

For the **-ir** option, see *31.4 Interactive Disassembler Mode*, p.438.

31.3.2 Description

Batch disassembly mode is selected by the **-i** option with no "**r**" modifier. In batch disassembler mode, **windiss** disassembles ELF object files and executables and writes the assembly code to standard output. The **-i** stands for instructions. **windiss** can disassemble programs compiled either:

- For the **windiss** environment, without hardware floating point support. See *31.2.1 Compiling for the WindISS Simulator*, p.433.
- For other environments, if there are no floating point instructions.

The modifiers **o**, **e**, and **l** are appended to the **-i** without an additional hyphen and with no spaces allowed. Modifiers may be used together in any order. To disassemble code use:

- **-i** alone to disassemble the whole file.
- [**e**] **-R1** *start-address* [**-R2** *end-address*] to specify code addresses. Use 0x for hex numbers. If part of a function is specified by a **-R1 and -R2** options, the entire function is disassembled unless the "**e**" option is used to request exact

addresses. A space is required between either the **-R1** or **-R2** option and the address.

- **-R3** *section* to specify a section index in the object file. If the specified section has zero length, the option is ignored.
- **o** to also show machine code.
- **l** *label* to specify the name of a function to be disassembled.

31.4 Interactive Disassembler Mode

31.4.1 Syntax (Interactive Disassembler Mode)

windiss -ir[**o**] *filename*

31.4.2 Description

In interactive disassembler mode, **windiss** prints the disassembled ELF object code and executables interactively. The **-i** stands for instructions; the **r** modifier selects interactive mode; the **o** modifier shows hex machine code in addition to assembly language. **windiss** can disassemble programs compiled either:

- For the **windiss** environment, without hardware floating point support. See *31.2.1 Compiling for the WindISS Simulator*, p.433.
- For other environments, if there are no floating point instructions.

To disassemble code in interactive mode:

d[**isasm**] *label* | [**-e**] *start-address* [*end-address*]

If part of a function is specified, the entire function will be disassembled unless the **-e** option is given. The **-e** option requests that exact addresses be disassembled, without other code.

To quit interactive mode:

q[**uit**]

31.5 Examples

Example 31-1 **Simulate Using All Defaults**

Run **windiss** in simulator mode. The program output is 17.

```
windiss a.out

17
windiss: task finished, exit code: 83521, Instructions executed: 2118
windiss: interrupts were never enabled
```

Example 31-2 **Simulate with Specified Memory Sizes**

Run **windiss** in simulator mode, specifying memory size as 20,000 bytes, and then 1 megabyte:

```
windiss -m 20000 a.out

windiss: loading outside of memory, EA=0x4c00 (increase by using -m
<size>)

windiss -m 1M a.out

17
windiss: task finished, exit code: 83521, Instructions executed: 2118
windiss: interrupts were never enabled
```

Example 31-3 **Simulate Showing POSIX Calls**

Run **windiss** in simulator mode, and use the debug option with a mask to show POSIX calls.

```
windiss -d 8 a.out
%% posix call 120: isatty(1), ret=1, errno=0
%% posix call 4: write(1, 0x6bfc, 4)

17
windiss: task finished, exit code: 83521, Instructions executed: 2118
windiss: interrupts were never enabled
```

Example 31-4 **Batch Disassemble Entire File**

Disassemble **a.out**:

```
windiss -i a.out
```

Example 31-5 **Batch Disassemble One Function in File**

Disassemble **main** in **a.out**:

```
windiss -il main a.out
```

Example 31-6 **Batch Disassemble Functions in Address Range**

Disassemble all code in function which includes addressees from 0x9c to 0x4e:

```
windiss -i -R1 0x9c -R2 0x4e a.out
```

Disassemble only code from 0x9c to 0x4e:

```
windiss -ie -R1 0x9c -R2 0x4e a.out
```

Example 31-7 **Interactive Disassembly**

Disassemble **a.out** in interactive mode, examine **main** and addresses 0xa0 to 0xa4:

```
windiss -ir a.out                    Command line
d main                               Interactive command
d -e 0xa0 0xa4                       Exact address range

q                                    Quit
```

PART VI

C Library

32

Library Structure, Rebuilding

32.1 Introduction

These chapters describe the C libraries provided with Wind River compiler.

The libraries are compliant with the following standards and definitions:

- ANSI X3.159-1989
- ISO/IEC 9945-1:1990
- POSIX IEEE Std 1003.1
- SVID Issue 2

For C++ specific headers, see *13.1 Header Files*, p.219.

32.2 Library Structure

NOTE: Libraries are usually selected automatically by the **-t** option to the linker, or by default as set by **dctrl -t**. This section is provided for user customization of the process and can be skipped for standard use.

The Wind River library structure supports a wide range of processors, types of floating point support, and execution environments. This section describes that structure and the mechanism used by the linker to select particular libraries.

This section should be read in conjunction with the following:

- *2. Configuration and Directory Structure*.
- *4. Selecting a Target and Its Components*.

These sections describe the location of the components of the tools and the configuration variables (and their equivalents — environment variables and command-line options) used to control their operation. That knowledge is assumed here.

32.2.1 Libraries Supplied

The next table shows the libraries distributed with the tools. This does not include **libc.a**, which is not an archive library, but is instead a text file which includes other libraries (see *32.2.3 libc.a*, p.449). These libraries are distributed in various subdirectories of *version_path* as described following the table.

`libcfp.a`
Floating point functions called by user code, including, for example, the **printf** and **scanf** formatting functions (but not the actual device input/output code). The version selected depends on the type of floating point selected: hardware, software, or none as described below.

Typically included automatically by **libc.a** (see *32.2.3 libc.a*, p.449).

`libchar.a`
Basic operating system functions using simple character input/output for **stdin** and **stdout** only (**stderr** and named files are not supported). This is an alternative to **libram.a**.

Sometimes included automatically by **libc.a**, see *32.2.3 libc.a*, p.449.

libcomplex.a
C++ complex math class library for use with older compiler releases. See *Older Versions of the Compiler*, p.214.

Not automatic; include with **-l** complex option.

libd.a
Additional standard library and support functions delivered with C++ only (**libc.a** is also required).

Included automatically in the link command generated by **dplus**. If the linker is invoked directly (command **dld**), then must be included by the user with the **-ld** option.

libdold.a
Additional standard library and support functions delivered with C++ only (**libc.a** is also required) for use with older compiler releases. See *Older Versions of the Compiler*, p.214.

Included automatically in the link command generated by **dplus** when the **-Xc++-old** option is used. If the linker is invoked directly (command **dld**), then must be included by the user with the **-ldold** option.

libi.a
General library containing all standard ANSI C functions except those in **libcfp.a**, **libchar.a**, and **libram.a**.

Typically included automatically by **libc.a** (see *32.2.3 libc.a*, p.449).

libimpfp.a
Conversions between floating point and other types. There are three versions: one for use with hardware floating point, one for software floating point, and an empty file when “none” is selected for floating point.

libimpl.a
Utility functions called by compiler-generated or runtime code for constructs not implemented in hardware, e.g. low-level software floating point (except conversions), 64-bit integer support, and register save/restore when absent in the hardware.

Typically included automatically by **libc.a** (see *32.2.3 libc.a*, p.449).

libios.a
C++ **iostream** class library for use with older compiler releases. See *Older Versions of the Compiler*, p.214.

Not automatic; include with **-lios** option.

libm.a
Advanced math function library.

Not automatic; include with an **-lm** option.

libstl.a
Alias for **libstlstd.a**.

Not automatic; include with **-lstl** (or **-lstlstd**) option.

libstlabr.a
Abridged standard C++ library. Does not provide exception-handling functions or the **type_info** class for RTTI support. For more information, see *13.2 C++ Standard Libraries*, p.220.

Not automatic; include with **-lstlabr** option.

libstlstd.a
C++ **iostream** and complex math class libraries.

Not automatic; include with **-lstlstd** (or **-lstl**) option.

libwindiss.a
Support library required by the **windiss** core instruction-set simulator. This library is included automatically whenever a **-t** option ending in "**:windiss**" is used, for example, **-tARMES:windiss**. See *31. WindISS Simulator and Disassembler* for details.

libpthread.a
Unsupported implementation of POSIX threads for use with the example programs. Text file which includes sub-libraries **libdk*.a**.

libram.a
Basic operating system functions using Ram-disk file input/output—an alternative to **libchar.a**.

Sometimes included automatically by **libc.a** (see *32.2.3 libc.a*, p.449).

The tools accommodate requirements for different floating point and target operating system and input/output support using two mechanisms:

- **libc.a** is a text file which includes a number of the libraries listed above. Several **libc.a** files which include different combinations are delivered for each target.
- The configuration information held in the configuration variables **DTARGET**, **DOBJECT**, **DFP**, and **DENVIRON** causes **dcc** or **dplus** to generate a particular set of paths used by the linker to search for libraries. By setting these configuration variables appropriately, the user can control the search and consequently the

particular **libc.a** or other libraries used by the linker to resolve unsatisfied externals.

As described in *4. Selecting a Target and Its Components*, these four configuration variables are normally set indirectly using the **-t***tof*:*environ* option on the command line invoking the compiler, assembler, or linker or by default with the **dctrl** program.

- The **DENVIRON** configuration variable (set from the *environ* part of **-t***tof*:*environ***)** designates the "target operating system" environment. The tools use two standard values: **simple** and **cross**, which as shown below, help define the library search paths.

  In addition, the tools may be supplied with directories and files to support other *environ* operating-system values. See the release notes and other relevant documentation for details on any particular operating system supported.

The remainder of this section describes these mechanisms in more detail.

32.2.2 Library Directory Structure

For ARM microprocessors:

- The library directories all begin with "ARM" as shown in Table 32-1.
- The object module format specifier — the *o* part of the **-t***tof*:*environ* option or its equivalent, is "**L**" for ELF.
- The tools have been installed in the *version_path* directory as described in Table 2-1.

Given the above assumptions, and following the pattern described in *4. Selecting a Target and Its Components*, the libraries above (*32.2.1 Libraries Supplied*, p.444) will be arranged as follows:

Table 32-1 **Library Directory Locations**

Directory / file	Contents
ARME/	Directories and files for ELF components (final "**E**" in **ARME**).
libc.a	Text file which includes other ELF libraries as described below — no input/output support.

Table 32-1 **Library Directory Locations** (cont'd)

Directory / file	Contents
libchar.a	ELF basic operating system functions using character input/output for **stdin** and **stdout** only (**stderr** and named files are not supported).
libi.a	ELF standard ANSI C functions.
libimpl.a	ELF functions called by compiler-generated or runtime code.
libd.a	ELF additional C++ standard and support functions.
libram.a	ELF basic operating system functions using RAM-disk input/output.
cross/libc.a	ELF **libc.a** which includes the RAM-disk input/output library **libram.a**.
simple/libc.a	ELF **libc.a** which includes the basic character input/output library **libchar.a**.
windiss/libwindiss.a	Support library for WindISS instruction-set simulator when supplied. Note: implicitly also uses **cross/libc.a**.
ARMEN/	ELF floating point stubs for floating point support of "None".
libcfp.a	Stubs to avoid undefined externals.
libimpfp.a	Empty file required by different versions of **libc.a.**
ARMES/	ELF software floating point libraries:
libcfp.a	Floating point functions called by user code.
libcomplex.a	Old C++ complex math class library.
libimpfp.a	Conversions between floating point and other types.
libios.a	Old C++ **iostream** class library.
libm.a	Math library.
libpthread.a	Unsupported implementation of POSIX threads for use with the example programs. Text file which includes sub-libraries **libdk*.a**.

Table 32-1 **Library Directory Locations** (cont'd)

Directory / file	Contents
libstlstd.a	C++ **iostream** and complex math class libraries.

32.2.3 libc.a

There are three **libc.a** files in the table above. Each of these is a short text file which contains **-l** option lines, each line naming a library. The **-l** option is the standard command-line option to specify a library for the linker to search. When the linker finds that **libc.a** is a text file, it reads the **-l** lines in the **libc.a** and then searches the named libraries for unsatisfied externals. (As with any **-l** option, only the portion of the name following "lib" is given; thus, **-li** identifies library **libi.a**.)

This approach allows the functions in **libc.a** to be factored into groups for different floating point and input/output requirements. Three of the **libc.a** files delivered with the tools are:

Table 32-2 **libc.a Files Delivered With the Tools**

liba.c files	Contents	Use

ARME/libc.a	`-li`
	`-lcfp`
	`-limpl`
	`-limpfp`
		Standard C runtime but with no input/output support; if input/output calls are made they will be undefined.

ARME/simple/libc.a	`-li`
	`-lcfp`
	`-lchar`
	`-limpl`
	`-limpfp`
		Supports character input/output by adding **libchar.a** for **stdin** and **stdout** only (**stderr** and named files are not supported).

ARME/cross/libc.a	`-li`
	`-lcfp`
	`-lram`
	`-limpl`
	`-limpfp`
		Supports RAM-disk input/output by adding **libram.a**. Used automatically by **windiss**.

Notes:

- Only one of the **simple** or **cross** (or similar) libraries should be used.
- **windiss** is a pseudo-value for *environ*: it selects the **windiss/libwindiss.a** library silently and in addition selects the **cross/libc.a** library.

- The order of the lines in each **liba.c** file determines the order in which the linker will search for unsatisfied externals.

The particular **libc.a** found, as well as the directories for the libraries listed in each **libc.a**, are determined by the search path given to the linker as described in the next section.

32.2.4 Library Search Paths

When **dcc** or **dplus** is invoked, it invokes the compiler, assembler, and linker in turn. The generated linker command line includes:

- an **-lc** option to cause the linker to search for **libc.a**
- for C++, an **-ld** option to cause the linker to search for **libd.a**
- a **-Y P** option which specifies the directories to be searched for these libraries and also for the libraries named in the selected **libc.a** (and any others specified by the user with **-l** *libname* **options)**

The **-Y P** option generated for each target is a function of the **-t***tof***:***environ* option or its equivalent environment variables, and is defined in *4.2 Selected Startup Module and Libraries*, p.24.

Following the pattern there, the assumptions made here will generate a **-Y P** option listing the following directories *in the order given* for each setting of the floating point *f* part of the **-t:***tof* option or its equivalent, and where *environ* is either **simple** or **cross**:

Table 32-3 **Directories Searched for Libraries**

'f'	**Directories**	**Environment**	**Floating point support**
N	*version_path*/**ARMEN**/*environ*	specific	None
	version_path/**ARMEN**	generic	None
	version_path/**ARME**/*environ*	specific	not applicable
	version_path/**ARME**	generic	not applicable
S	*version_path*/**ARMES**/*environ*	specific	Software
	version_path/**ARMES**	generic	Software
	version_path/**ARME**/*environ*	specific	not applicable
	version_path/**ARME**	generic	not applicable

Notes:

- There is no error if a directory given with the **-Y P** option does not exist.
- The difference between "None" floating point support and "not applicable" is that the directories for the "not applicable" cases do not contain any floating point code, only integer, while the "None" cases will use the **ARMEN/libcfp.a** and **ARMEN/libimpfp.a** libraries. **ARMEN/libcfp.a** provides stubs functions that call **printf** with an error message for floating point externals used by compiler-generated or runtime code so that these externals will not be undefined; **ARMEN/libimpfp** is an empty file needed because each **libc.a** is common to all types of floating point support.

The following table gives examples of the libraries found given the above directory search order. Note that the search for the libraries included by a **libc.a** is independent of the search for **libc.a**. That is, regardless of which directory supplies **libc.a**, the search for the libraries it names begins anew with the first directory in the selected row of Table 32-3 above. In all cases, a library is taken from the first directory in which it is found.

Table 32-4 **Examples of Libraries Found for Different -t Options**

-t option	Libraries Found	Notes

`-tARMEN:simple`	`ARME/simple/libc.a`
	
	`ARME/libi.a`
	`ARMEN/libcfp.a`
	`ARME/libchar.a`
	`ARME/libimpl.a`
	`ARMEN/libimpfp.a`
		libc.a is specific to the environment, but never to the floating point support. It is found in the third directory searched. It names four libraries:
		▪ **libi.a** and **libimpl.a** are common to all **ARME** systems and are found in the fourth directory **ARM**.
		▪ The floating point support is independent of the environment and comes from the second directory **ARMEN**.
		▪ The character input/output support is independent of the floating point support, and while it has been selected because of the **simple** environment setting, it resides in the generic fourth directory **ARM**.

Table 32-4 **Examples of Libraries Found for Different -t Options** (cont'd)

-t option	Libraries Found	Notes
-tARMES:cross	**ARME/cross/libc.a** **ARME/libi.a** **ARMES/libcfp.a** **ARME/libram.a** **ARME/libimpl.a** **ARMES/libimpfp.a**	Again, **libc.a** is specific to the environment but not the floating point support, and is found in the third directory **ARME/cross**. It again names four libraries: ▪ **libi.a** and **libimpl.a** are in the fourth directory **ARME** as before. ▪ The software floating point library **libcfp.a** is from the second directory, now **ARMES**. ▪ This time **libram.a** has been selected by **ARME/cross/libc.a** instead of **libchar.a** (but still from the fourth directory **ARME** as before).
-tARMES:windiss	In addition to the libraries found for **-tARMES:cross**, searches **windiss/libwindiss.a** before searching for **ARME/cross/libc.a**.	

Table 32-4 **Examples of Libraries Found for Different -t Options** (cont'd)

-t option	Libraries Found	Notes

-tARMES:cust	**ARME/cust/libc.a**
	
	ARME/libi.a
	ARMES/libcfp.a
	ARME/cust/libchar.a
	ARME/libimpl.a
	ARMES/libimpfp.a
		The customer has defined a new **libc.a** in a new **ARME/cust** directory for a C++ project using software floating point. This **libc.a** text file consists of the following five lines:
		
		`-li`
		`-lcfp`
		`-lchar`
		`-limpl`
		`-limpfp`
		
		Thus, based on the search order implied by the **-tARMES:cust** option, the standard libraries **ARME/libi.a, ARME/libimpl.a**, **ARMES/libcfp.a**, and **ARMES/libimpfp.a** will be searched.
		
		In addition, the library **ARME/cust/libchar.a**, a special character I/O package for the customer's **ARME -t** environment, will also be searched. Because directory **ARMES/cust** is searched before **ARME**, the linker will find the customer's **libchar.a** library rather than the standard **ARME/libchar.a**.

32.3 Library Sources, Rebuilding the Libraries

32.3.1 Sources

This section describes how to re-build the libraries from source.

The libraries and makefiles are contained in three subdirectories of *version_path*/**libraries**:

build/*
There are subdirectories for each of **ARME**, **ARMEH**, **ARMEN**, etc. Each subdirectory contains a main **Makefile** and supporting makefiles.

Only the **ARME/Makefile** is to be used directly by the user. It in turn invokes the makefiles in **ARMEH**, **ARMEN**, etc. These latter makefiles are self-documenting and begin with comments that should be read before re-building the libraries.

include/
include.cxx/*
include.unx/*
Include files used by for the C++ (but not C), and C libraries respectively.

src/*
Source for all generally distributed library files.

32.3.2 Rebuilding the Libraries

The following steps rebuild the libraries:

1. If you do not want to run make against all of the libraries, edit the **Makefile** at both the **ARME** level and the **ARMEH, ARMEN**, etc., levels to remove any unwanted libraries.
2. Change directory to *version_path***/libraries/build/ARME**.
3. Enter the command:

   ```
   dmake -vd
   ```

   Note: to change the arguments that the libraries build with, change the **CFLAGS** macro defined in *version_path***/libraries/build/defs.mk**.
4. Each library will be built in its corresponding **build** directory, that is, *version_path***/libraries/build/ARME**, *version_path***/libraries/build/ARMEH**, etc.
5. Move the successfully built libraries to the *version_path***/ARME**, *version_path***/ARMEH**, etc. corresponding directories, replace each existing file with the newly built file.

   Alternatively, leave the libraries where they are, or move them to some other location, and provide **-Y P** options as described in the first part of this chapter.

NOTE: The Dinkum C++ libraries are built with the GNU make utility (**gmake**), not with **dmake**.

32.3.3 C++ Libraries

The Wind River tools include two versions of the standard C++ library: the complete version (**libstlstd.a**) and the abridged version (**libstlabr.a**). For information about these libraries, see *13.2 C++ Standard Libraries*, p.220. By default, **libstlstd.a** is compiled with the full library sources and exception-handling enabled, while **libstlabr.a** is compiled with the abridged library sources and exception-handling disabled. You can compile these libraries in a different configuration by redefining either or both of the macros **__CONFIGURE_EMBEDDED** and **__CONFIGURE_EXCEPTIONS**. These macros are defined in **dtools.conf** and automatically reset by compiler flags such as **-Xc++abr**; hence their definitions must be overridden on the command line if you wish to change them. Setting **__CONFIGURE_EMBEDDED** to 1 uses the abridged library sources and setting **__CONFIGURE_EXCEPTIONS** to 1 enables exception-handling. For example, to compile the **libstlstd.a** without exception-handling, add **__CONFIGURE_EXCEPTIONS=0** to the command line.

33 Header Files

33.1 Files

The following list is a subset of the header files provided. Each is enclosed in angle brackets, < >, whenever used in text to emphasize their inclusion in the standard C library.

All header files are found in *version_path*/**include**. See *2. Configuration and Directory Structure* for additional information.

NOTE: In this manual, some paths are given using UNIX format, that is, using a "/" separator. For Windows, substitute a"\" separator.

33.1.1 Standard Header Files

<ar.h>
Archive header.

<assert.h>
assert( ) macro.

<ctype.h>
Character handling macros.

<dcc.h>
Prototypes not found elsewhere.

<errno.h>
error macros and **errno** variable.

<fcntl.h>
creat( ), **fcntl( )**, and **open( )** definitions.

<float.h>
Floating point limits.

<limits.h>
Limits of processor and operating system.

<locale.h>
Locale definitions.

<malloc.h>
Old **malloc( )** definitions. Use <**stdlib.h**>.

<math.h>
Defines the constant **HUGE_VAL** and declares math functions.

<mathf.h>
Single precision versions of <**math.h**> functions.

<memory.h>
Old declarations of **mem*( )**. Use <**string.h**>.

<mon.h>
monitor( ) definitions.

<netdb.h>
Berkeley socket standard header file.

<netinet/in.h>
Berkeley socket standard header file.

<netinet/tcp.h>
Berkeley socket standard header file.

<regexp.h>
Regular expression handling.

<search.h>
Search routine declarations.

<setjmp.h>
setjmp( ) and **longjmp( )** definitions.

<signal.h>
Signal handling.

<stdarg.h>
ANSI variable arguments handling.

<stddef.h>
ANSI definitions.

<stdio.h>
stdio library definitions.

<stdlib.h>
ANSI definitions.

<string.h>
str*( ) and **mem*( )** declarations.

<sys/socket.h>
Berkeley socket standard header file.

<sys/types.h>
Type definitions.

<time.h>
Time handling definitions.

<unistd.h>
Prototypes for UNIX system calls.

<values.h>
Old limits definitions. Use <**limits.h**> and <**float.h**>.

<varargs.h>
Old variable arguments handling. Use <**stdarg.h**>.

NOTE: If the macro **__lint** is set (**#define __lint**), the header files will not use any C language extensions. This is useful for checking code before running it with a third party lint facility.

33.2 Defined Variables, Types, and Constants

The following list is a subset of the variables, types, and constants defined in the header files in the C libraries.

errno.h

Declares the variable **errno** holding error codes. Defines error codes; all starting with **E**. See the file for more information.

fcntl.h

Defines the following constants used by **open( )** and **fcntl( )**:

O_RDONLY
Open for reading only.

O_WRONLY
Open for writing only.

O_RDWR
Open for reading and writing.

O_NDELAY
No blocking.

O_APPEND
Append all writes at the end of the file.

float.h

Defines constants handling the precision and range of floating point values. See the ANSI C standard for reference.

limits.h

Defines constants defining the range of integers and operating system limits. See the ANSI C and POSIX 1003.1 standards for reference.

math.h

Defines the value **HUGE_VAL** that is set to IEEE double precision infinity.

mathf.h

Defines the value **HUGE_VAL_F** that is set to IEEE single precision infinity.

setjmp.h

Defines the type **jmpbuf**, used by **setjmp( )** and **longjmp( )**.
Defines the type **sigjmpbuf**, used by **sigsetjmp( )** and **siglongjmp( )**.

signal.h

Defines the signal macros starting with SIG.
Defines the volatile type **sig_atomic_t** that can be used by signal handlers.
Defines the type **sigset_t**, used by POSIX signal routines.

stdarg.h

Defines the type **va_list** used by the macros **va_start**, **va_arg**, and **va_end**.

stddef.h

Defines **ptrdiff_t** which is the result type of subtracting two pointers.
Defines **size_t** which is the result type of the **sizeof** operator.
Defines **NULL** which is the null pointer constant.

stdio.h

Defines **size_t** which is the result type of the **sizeof** operator.
Defines **fpos_t** which is the type used for file positioning.
Defines **FILE** which is the type used by stream and file input and output.
Defines the **BUFSIZ** constant which is the size used by **setbuf( )**.
Defines the **EOF** constant which indicates end-of-file.
Defines **NULL** which is the null pointer constant.
Declares **stdin** as a pointer to the **FILE** associated with standard input.
Declares **stdout** as a pointer to the **FILE** associated with standard output.
Declares **stderr** as a pointer to the **FILE** associated with standard error.

stdlib.h

Defines **size_t** which is the result type of the **sizeof** operator.
Defines **div_t** and **ldiv_t** which are the types returned by **div( )** and **ldiv( )**.
Defines **NULL** which is the null pointer constant.
Defines the **EXIT_FAILURE** and **EXIT_SUCCESS** constants returned by **exit( )**.

string.h

Defines **NULL** which is the null pointer constant.
Defines **size_t** which is the result type of the **sizeof** operator.

time.h

Defines **CLOCKS_PER_SEC** constant which is the number of clock ticks per second.

34 C Library Functions

34.1 Format of Descriptions

This chapter briefly describes the functions and function-like macros provided in the Wind River C libraries. For more detailed descriptions, and for information about the C++ libraries, see the references cited in *Additional Documentation*, p.8.

NOTE: The standard C libraries documented here are *not* the ones used for VxWorks applications. If you specify the **:rtp** or **:vxworks***x.x* execution environment, the tools will automatically link a different set of C libraries. See the documentation that accompanied your VxWorks development tools for more information.

Each function description is formatted as follows:

name
header files
prototype definition
brief description

OS calls: optional; see below
Reference: see below

34.1.1 Operating System Calls

Some of the functions described in this chapter make calls on operating system functions that are standard in UNIX environments. In embedded environments, such functions cannot be used unless the embedded environment includes a real-time operating system providing these operating system functions.

The functions which call operating system functions, directly or indirectly, have all the required operating system functions listed. The non-UNIX user can employ this list to see what system functions need to be provided in order to use a particular function.

Some functions refer to standard input, output, and error — the standard input/output streams found in UNIX and Windows environments. For embedded environments, see *15.8.1 Character I/O*, p.264 and *15.8.2 File I/O*, p.265 for suggestions for file system support.

34.1.2 References

The function descriptions refer to the following standards and definitions:

ANSI
: The function/macro is defined in ANSI X3.159-1989.

ANSI 754
: The function is define in ANSI/IEEE Std 754-1985.

DCC
: The function/macro is added to Wind River C.

POSIX
: The function/macro is defined in IEEE Std 1003.1-1990.

SVID
: The function/macro is defined in System V Interface Definition 2.

UNIX
: The function/macro is provided to be compatible with Unix V.3.

Other references:

MATH
: The math libraries must be specified at link time with the **-lm** option.

SYS
: The function must be provided by the operating system or emulated in a stand-alone system.

REENT
: The function is reentrant. It does not use any static or global data.

REERR
: The function might modify **errno** and is reentrant only if all processes ignore that variable. But see *34.2 Reentrant Versions*, p.465 below.

Most functions in the libraries have a synonym to conform to various standards. For example, the function **read( )** has the synonym **_read( )**. In ANSI C, **read( )** is not defined, which means that the user is free to define **read( )** as a new function. To avoid conflicts with such user-defined functions, library functions, e.g. **fread( )**, call the synonym defined with the leading underscore, e.g. **_read( )**.

34.2 Reentrant Versions

In some cases, non-reentrant standard functions are supplied in special reentrant versions. These reentrant versions are not separately documented, but they are easy to find because their names end in *_r*. For example, **localtime( )** (in **gmtime.c**) has a reentrant counterpart called **localtime_r( )** (in **gmtime_r.c**).

All functions that modify the **errno** variable call the wrapper function **__errno_fn( )**, defined in **cerror.c**. When a function is marked as REERR in the listing below, you can make it completely reentrant by modifying **__errno_fn( )** to preserve the value of **errno**.

For information about **malloc( )** and **free( )**, see *15.10 Reentrant and "Thread-Safe" Library Functions*, p.270.

34.3 Function Listing

This section lists all functions in the library in alphabetic order. Leading underscores "_" are ignored with respect to the alphabetic ordering.

a64l( )

```
#include <stdlib.h>
long a64l(const char *s);
```

Converts the base-64 number, pointed to by *s, to a long value.

Reference: SVID, REENT.

abort( )

```
#include <stdlib.h>
int abort(void);
```

Same as **exit( )**, but also causes the signal **SIGABRT** to be sent to the calling process. If **SIGABRT** is neither caught nor ignored, all streams are flushed prior to the signal being sent and a core dump results.

OS calls: **close**, **getpid**, **kill**, **sbrk**, **write**.

Reference: ANSI.

abs( )

```
#include <stdlib.h>
int abs(int i);
```

Returns the absolute value of its integer operand.

Reference: ANSI, REENT.

access( )

```
#include <unistd.h>
int access(char *path, int amode);
```

Determines accessibility of a file.

The C libraries provide an interface to this operating system call. Please see your OS manual for a complete definition.

Reference: POSIX, SYS.

acos( )

```
#include <math.h>
double acos(double x);
```


Returns the arc cosine of *x* in the range [0, π]. *x* must be in the range [-1, 1]. Otherwise zero is returned, **errno** is set to **EDOM**, and a message indicating a domain error is printed on the standard error output.

OS calls: **write**.

Reference: ANSI, MATH, REERR.

acosf( )

```
#include <mathf.h>
float acosf(float x);
```

Returns the arc cosine of *x* in the range [0, π]. *x* must be in the range [-1, 1]. Otherwise zero is returned, **errno** is set to **EDOM**, and a message indicating a domain error is printed on the standard error output. This is the single precision version of **acos( )**.

OS calls: **write**.

Reference: DCC, MATH, REERR.

advance( )

```
#include <regexp.h>
int advance(char *string, char *expbuf);
```

Does pattern matching given the string *string* and a compiled regular expression in *expbuf*. See SVID for more details.

Reference: SVID.

asctime( )

```
#include <time.h>
char *asctime(const struct tm *timeptr);
```

Converts time in *timeptr* into a string in the form exemplified by

```
    "Sun Sep 16 01:03:52 1973\n".
```

Reference: ANSI.

asin( )

```
#include <math.h>
double asin(double x);
```

Returns the arc sine of *x* in the range [-π/2, π/2]. x must be in the range [-1, 1]. Otherwise zero is returned, **errno** is set to **EDOM** and a message indicating a domain error is printed on the standard error output.

OS calls: **write**.

Reference: ANSI, MATH, REERR.

asinf( )

```
#include <mathf.h>
float asinf(float x);
```

Returns the arc sine of *x* in the range [-π/2, π/2]. *x* must be in the range [-1, 1]. Otherwise zero is returned, **errno** is set to **EDOM** and a message indicating a domain error is printed on the standard error output. This is the single precision version of **asin( )**.

OS calls: **write**.

Reference: DCC, MATH, REERR.

assert( )

```
#include <assert.h>
void assert(int expression);
```

Puts diagnostics into programs. If *expression* is false, **assert( )** writes information about the particular call that failed (including the text of the argument, the name of the source file, and the source line number — the latter are respectively the

values of the preprocessing macros __**FILE**__ and __**LINE**__) on the standard error file. It then calls the **abort( )** function. **assert( )** is implemented as a macro. If the preprocessor macro **NDEBUG** is defined at compile time, the **assert( )** macro will not generate any code.

OS calls: **close**, **getpid**, **kill**, **sbrk, write**.

Reference: ANSI.

atan( )

```
#include <math.h>
double atan(double x);
```

Returns the arc tangent of *x* in the range [-π/2, π/2].

OS calls: **write**.

Reference: ANSI, MATH, REERR.

atanf( )

```
#include <mathf.h>
float atan(float x);
```

Returns the arc tangent of *x* in the range [-π/2, π/2]. This is the single precision version of **atan( )**.

OS calls: **write**.

Reference: DCC, MATH, REERR.

atan2( )

```
#include <math.h>
double atan2(double x, double y);
```

Returns the arc tangent of *y/x* in the range [-π, π], using the signs of both arguments to determine the quadrant of the return value. If both arguments are zero, then zero is returned, **errno** is set to **EDOM** and a message indicating a domain error is printed on the standard error output.

OS calls: **write**.

Reference: ANSI, MATH, REERR.

atan2f( )

```
#include <mathf.h>
float atan2(float x, float y);
```

Returns the arc tangent of *y/x* in the range [-π, π], using the signs of both arguments to determine the quadrant of the return value. If both arguments are zero, then zero is returned, **errno** is set to **EDOM** and a message indicating a domain error is printed on the standard error output. This is the single precision version of **atan2( )**.

OS calls: **write**.

Reference: DCC, MATH, REERR.

atexit( )

```
#include <stdlib.h>
void atexit(void (*func) (void));
```

Registers the function whose address is *func* to be called by **exit( )**.

Reference: ANSI.

atof( )

```
#include <stdlib.h>
double atof(const char *nptr);
```

Converts an ASCII number string *nptr* into a **double**.

Reference: ANSI, REERR.

atoi( )

```
#include <stdlib.h>
int atoi(const char *nptr);
```

Converts an ASCII decimal number string *nptr* into an **int**.

Reference: ANSI, REENT.

atol( )

```
#include <stdlib.h>
long atol(const char *nptr);
```

Converts an ASCII decimal number string *nptr* into a **long**.

Reference: ANSI, REENT.

bsearch( )

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base, size_t nel, size_t size,
int (*compar)( ));
```

Binary search routine which returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order. *key* points to a datum instance to search for in the table, *base* points to the element at the base of the table, *nel* is the number of elements in the table. *compar* is a pointer to the comparison function, which is called with two arguments that point to the elements being compared.

Reference: ANSI, REENT.

calloc( )

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

Allocates space for an array of *nmemb* objects of the size *size*. Returns a pointer to the start (lowest byte address) of the object. The array is initialized to zero. See **malloc( )** for more information.

OS calls: **sbrk**, **write**.

Reference: ANSI.

ceil( )

```
#include <math.h>
double ceil(double x);
```

Returns the smallest integer not less than *x*.

OS calls: **write**.

Reference: ANSI, MATH, REENT.

ceilf( )

```
#include <mathf.h>
float ceilf(float x);
```

Returns the smallest integer not less than *x*. This is the single precision version of **ceil( )**.

OS calls: **write**.

Reference: DCC, MATH, REENT.

_chgsign( )

```
#include <math.h>
double _chgsign(double x);
```

Returns *x* copies with its sign reversed, not 0 - *x*. The distinction is germane when *x* is +0 or -0 or NaN. Consequently, it is a mistake to use the sign bit to distinguish signaling NaNs from quiet NaNs.

Reference: ANSI 754, MATH, REENT.

clearerr( )

```
#include <stdio.h>
void clearerr (FILE *stream);
```

Resets the error and EOF indicators to zero on the named *stream*.

Reference: ANSI.

clock( )

```
#include <time.h>
clock_t clock(void);
```

Returns the number of clock ticks of elapsed processor time, counting from a time related to program start-up. The constant **CLOCKS_PER_SEC** is the number of ticks per second.

OS calls: **times**.

Reference: ANSI.

close( )

```
#include <unistd.h>
int close(int fildes);
```

Closes the file descriptor *fildes*.

The C libraries provide an interface to this operating system call. Please see your OS manual for a complete definition.

Reference: POSIX, SYS.

compile( )

```
#include <regexp.h>
int compile(char *instring, char *expbuf, char *endbuf, int eof);
```

Compiles the regular expression in *instring* and produces a compiled expression that can be used by **advance( )** and **step( )** for pattern matching.

Reference: SVID.

_copysign( )

```
#include <math.h>
double _copysign(double x, double y);
```

Returns *x* with the sign of *y*. Hence, **abs(x)** = **_copysign(x, 1.0)** even if *x* is NaN.

Reference: ANSI 754, MATH, REENT.

cos( )

```
#include <math.h>
double cos(double x);
```

Returns the cosine of *x* measured in radians. Accuracy is reduced with large argument values.

OS calls: **write**.

Reference: ANSI, MATH, REERR.

cosf( )

```
#include <mathf.h>
float cosf(float x);
```

Returns the cosine of *x* measured in radians. Accuracy is reduced with large argument values. This is the single precision version of **cos( )**.

OS calls: **write**.

Reference: DCC, MATH, REERR.

cosh( )

```
#include <math.h>
double cosh(double x);
```

Returns the hyperbolic cosine of *x* measured in radians. Accuracy is reduced with large argument values.

OS calls: **write**.

Reference: ANSI, MATH, REERR.

coshf( )

```
#include <mathf.h>
float coshf(float x);
```

Returns the hyperbolic cosine of *x* measured in radians. Accuracy is reduced with a large argument values. This is the single precision version of **cosh( )**.

OS calls: **write**.

Reference: DCC, MATH, REERR.

creat( )

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat(char *path, mode_t mode);
```

Creates the new file *path*.

The C libraries provide an interface to this operating system call. Please see your OS manual for a complete definition.

Reference: POSIX, SYS.

ctime( )

```
#include <time.h>
char *ctime(const time_t *timer);
```

Equivalent to calling **asctime(localtime(***timer***))**.

Reference: ANSI.

difftime( )

```
#include <time.h>
double difftime(time_t t1, time_t t0);
```

Returns the difference in seconds between the calendar time *t0* and the calendar time *t1*.

Reference: ANSI, REENT.

div( )

```
#include <stdlib.h>
div_t div(int numer, int denom);
```

Divides *numer* by *denom* and returns the quotient and the remainder as a **div_t** structure.

Reference: ANSI, REENT.

drand48( )

```
#include <stdlib.h>
double drand48(void);
```

Generates pseudo-random, non-negative, double-precision floating point numbers uniformly distributed over the half-open interval [0.0, 1.0[(i.e. excluding 1.0), using the linear congruential algorithm and 48-bit integer arithmetic. It must be initialized using the **srand48( )**, **seed48( )**, or **lcong48( )** functions.

Reference: SVID.

dup( )

```
#include <unistd.h>
int dup(int fildes);
```

Duplicates the open file descriptor *fildes*.

The C libraries provide an interface to this operating system call. Please see your OS manual for a complete definition.

Reference: POSIX, SYS.

ecvt( )

```
#include <dcc.h>
char *ecvt(double value, int ndigit, int *decpt, int *sign);
```

Converts *value* to a null-terminated string of *ndigit* digits and returns a pointer to it. The high-order digit is non-0 unless *value* is zero. The low-order digit is rounded to the nearest value (5 is rounded up). The position of the decimal point relative the beginning of the string is stored through *decpt* (negative means to the left of the returned digits). If the sign of the result is negative, the integer pointed to by *sign* is set to one, otherwise it is set to zero.

Reference: DCC.

erf( )

```
#include <math.h>
double erf(double x);
```

Returns the error function of *x*.

Reference: SVID, MATH, REENT.

erff( )

```
#include <mathf.h>
float erff(float x);
```

Returns the error function of *x*. This is the single precision version of **erf( )**.

Reference: DCC, MATH, REENT.

erfc( )

```
#include <math.h>
double erfc(double x);
```

Complementary error function = 1.0 - **erf(*x*)**. Provided because of the extreme loss of relative accuracy if **erf(*x*)** is called for large *x* and the result subtracted from 1.0.

Reference: SVID, MATH, REENT.

erfcf( )

```
#include <mathf.h>
float erfcf(float x);
```

Complementary error function = 1.0 - **erff**(*x*). Provided because of the extreme loss of relative accuracy if **erff(*x*)** is called for large *x* and the result subtracted from 1.0. This is the single precision version of **erfc( )**.

Reference: DCC, MATH, REENT.

exit( )

```
#include <stdlib.h>
void exit(int status);
```

Normal program termination. Flushes all open files. Executes all functions submitted by the **atexit( )** function. Does not return to its caller. The following *status* constants are provided:

EXIT_FAILURE	unsuccessful termination
EXIT_SUCCESS	successful termination

OS calls: **_exit**, **close**, **sbrk, write**.

Reference: ANSI.

_exit( )

```
#include <unistd.h>
void _exit(int status);
```

Program termination. All files are closed. Does not return to its caller.

The C libraries provide an interface to this operating system call. Please see your OS manual for a complete definition.

Reference: POSIX, SYS.

exp( )

```
#include <math.h>
double exp(double x);
```

Returns the exponential function of *x*. Returns **HUGE_VAL** when the correct value would overflow or 0 when the correct value would underflow, and sets **errno** to **ERANGE**.

OS calls: **write**.

Reference: ANSI, MATH, REERR.

expf( )

```
#include <mathf.h>
float expf(float x);
```

Returns the exponential function of *x*. Returns **HUGE_VAL** when the correct value would overflow or 0 when the correct value would underflow and sets **errno** to **ERANGE**. This is the single precision version of **exp( )**.

OS calls: **write**.

Reference: DCC, MATH, REERR.

fabs( )

```
#include <math.h>
double fabs(double x);
```

Returns the absolute value of *x*.

Reference: ANSI, MATH, REENT.

fabsf( )

```
#include <mathf.h>
float fabsf(float x);
```

Returns the absolute value of *x*. This is the single precision version of **fabs( )**.

Reference: DCC, MATH, REENT.

fclose( )

```
#include <stdio.h>
int fclose(FILE *stream);
```

Causes any buffered data for the named *stream* to be written out, and the stream to be closed.

OS calls: **close**, **sbrk, write**.

Reference: ANSI.

fcntl( )

```
#include <fcntl.h>
int fcntl(int fildes, int cmd, ...);
```

Controls the open file *fildes*.

The C libraries provide an interface to this operating system call. Please see your OS manual for a complete definition.

Reference: POSIX, SYS.

fcvt( )

```
#include <dcc.h>
char *fcvt(double value, int ndigit, int *decpt, int *sign);
```

Rounds the correct digit for **printf** format "**%f**" (FORTRAN F-format) output according to the number of digits specified. See **ecvt( )**.

Reference: DCC.

fdopen( )

```
#include <stdio.h>
FILE *fdopen(int fildes, const char *type);
```

See **fopen( )**. **fdopen( )** associates a stream with a file descriptor, obtained from **open( )**, **dup( )**, **creat( )**, or **pipe( )**. The *type* of stream must agree with the mode of the open file.

OS calls: **fcntl**, **lseek**.

Reference: POSIX.

feof( )

```
#include <stdio.h>
int feof (FILE *stream);
```

Returns non-zero when end-of-file has previously been detected reading the named input *stream*.

Reference: ANSI.

ferror( )

```
#include <stdio.h>
int ferror (FILE *stream);
```

Returns non-zero when an input/output error has occurred while reading from or writing to the named *stream*.

Reference: ANSI.

fflush( )

```
#include <stdio.h>
int fflush(FILE *stream);
```

Causes any buffered data for the named *stream* to be written to the file, and the *stream* remains open.

OS calls: **write**.

Reference: ANSI.

fgetc( )

```
#include <stdio.h>
int fgetc(FILE *stream);
```

Behaves like the macro **getc( )**, but is a function. Runs more slowly than **getc( )**, takes less space, and can be passed as an argument to a function.

OS calls: **isatty**, **read**, **sbrk**, **write**.

Reference: ANSI.

fgetpos( )


```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

Stores the file position indicator for *stream* in **pos*. If unsuccessful, it stores a positive value in **errno** and returns a nonzero value.

OS calls: **lseek**.

Reference: ANSI.

fgets( )

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

Reads characters from *stream* into the array pointed to by *s*, until *n*-1 characters are read, or a new-line character is read and transferred to *s*, or an EOF is encountered. The string is terminated with a null character.

OS calls: **isatty**, **read**, **sbrk**, **write**.

Reference: ANSI.

fileno( )

```
#include <stdio.h>
int fileno (FILE *stream);
```

Returns the integer file descriptor associated with the named *stream*; see **open( )**.

Reference: POSIX.

_finite( )

```
#include <math.h>
double _finite(double x);
```

Returns a non-zero value if -∞< *x* < +∞, and returns 0 otherwise.

Reference: ANSI 754, MATH, REENT

floor( )

```
#include <math.h>
double floor(double x);
```

Returns the largest integer (as a double-precision number) not greater than *x*.

Reference: ANSI, MATH, REENT.

floorf( )

```
#include <mathf.h>
float floorf(float x);
```

Returns the largest integer (as a single-precision number) not greater than *x*. This is the single precision version of **floor( )**.

Reference: DCC, MATH, REENT.

fmod( )

```
#include <math.h>
double fmod(double x, double y);
```

Returns the floating point remainder of the division of *x* by *y*, zero if *y* is zero or if *x*/*y* would overflow. Otherwise the number is f with the same sign as *x*, such that *x*=i*y*+f for some integer i, and absolute value of f is less than absolute value of *y*.

Reference: ANSI, MATH, REENT.

fmodf( )

```
#include <mathf.h>
float fmodf(float x, float y);
```

Returns the floating point remainder of the division of *x* by *y*, zero if *y* is zero or if *x*/*y* would overflow. Otherwise the number is f with the same sign as x, such that *x*=i*y*+f for some integer i, and absolute value of f is less than absolute value of *y*. This is the single precision version of **fmod( )**.

Reference: DCC, MATH, REENT.

fopen( )

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *type);
```

Opens the file named by *filename* and associates a stream with it. Returns a pointer to the **FILE** structure associated with the stream. *type* is a character string having one of the following values:

"**r**"	open for reading
"**w**"	truncate or create for writing
"**a**"	append; open for writing at EOF, or create for writing
"**r+**"	open for update (read and write)
"**w+**"	truncate or create for update
"**a+**"	append; open or create for update at EOF

A "**b**" can also be specified as the second or third character in the above list, to indicate a binary file on systems where there is a difference between text files and binary files. Examples: "**rb**", "**wb+**", and "**a+b**".

OS calls: **lseek**, **open**.

Reference: ANSI.

fprintf( )

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format, ... );
```

Places output argument on named output stream. See **printf( )**.

NOTE: By default in most environments, **fprintf** buffers its output until a newline is output. To cause output character-by-character without waiting for a newline, call *setbuf( )*, p.520, with a NULL buffer pointer after opening but before writing to the stream:

```
setbuf(*stream, 0);
```

OS calls: **isatty**, **sbrk**, **write**.

Reference: ANSI.

fputc( )

```
#include <stdio.h>
int fputc(int c, FILE *stream)
```

Behaves like the macro **putc( )**, but is a function. Therefore, it runs more slowly, takes up less space, and can be passed as an argument to a function.

OS calls: **isatty**, **sbrk**, **write**.

Reference: ANSI.

fputs( )

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

Writes the null-terminated string pointed to by s to the named output *stream*.

OS calls: **isatty**, **sbrk**, **write**.

Reference: ANSI.

fread( )

```
#include <stdio.h>
#include <sys/types.h>
int fread(void *ptr, size_t size, int nitems, FILE *stream);
```

Copies *nitems* items of data from the named input *stream* into an array pointed to by *ptr*, where an item of data is a sequence of bytes of length *size*. It leaves the file pointer in *stream* pointing to the byte following the last byte read.

OS calls: **isatty**, **read**, **sbrk**, **write**.

Reference: ANSI.

free( )

```
#include <stdlib.h>
void free(void *ptr);
extern int __no_malloc_warning;
```

Object pointed to by *ptr* is made available for further allocation. *ptr* must previously have been assigned a value from **malloc( )**, **calloc( )**, or **realloc( )**.

If the pointer *ptr* was freed or not allocated by **malloc( )**, a warning is printed on the **stderr** stream. The warning can be suppressed by assigning a non-zero value to the integer **__no_malloc_warning**. See **malloc( )** for more information.

OS calls: **sbrk**, **write**.

Reference: ANSI.

freopen( )


```
#include <stdio.h>
FILE *freopen(const char *filenam, const char *type, FILE *stream);
```

See **fopen( )**. **freopen( )** opens the named file in place of the open *stream*. The original stream is closed, and a pointer to the **FILE** structure for the new stream is returned.

OS calls: **close**, **lseek**, **open**, **sbrk**, **write**.

Reference: ANSI.

frexp( )

```
#include <math.h>
double frexp(double value, int *eptr);
```

Given that every non-zero number can be expressed as $x*(2^n)$, where $0.5 <= |x| < 1.0$ and n is an integer, this function returns x for a *value* and stores n in the location pointed to by *eptr*.

Reference: ANSI, REENT.

frexpf( )

```
#include <mathf.h>
float frexpf(float value, int *eptr);
```

Given that every non-zero number can be expressed as $x*(2^n)$, where $0.5 <= |x| < 1.0$ and n is an integer, this function returns x for a *value* and stores n in the location pointed to by *eptr*. This is the single precision version of **frexp( )**.

Reference: DCC, MATH, REENT.

fscanf( )

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...);
```

Reads formatted data from the named input *stream* and optionally assigns converted data to variables specified by the *format* string. Returns the number of successful conversions (or **EOF** if input is exhausted). See **scanf( )**.

OS calls: **isatty**, **read**, **sbrk**, **write**.

Reference: ANSI.

fseek( )

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
```

Sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, from the current position, or from the end of the file, according to *whence*. The next operation on a file opened for update may be either input or output. whence has one of the following values:

SEEK_SET	offset is absolute position from beginning of file.
SEEK_CUR	offset is relative distance from current position.
SEEK_END	offset is relative distance from the end of the file.

OS calls: **lseek**, **write**.

Reference: ANSI.

fsetpos( )

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

Sets the file position indicator for *stream* to **pos* and clears the EOF indicator for *stream*. If unsuccessful, stores a positive value in **errno** and returns a nonzero value.

OS calls: **lseek**, **write**.

Reference: ANSI.

fstat( )

```
#include <sys/types.h>
#include <sys/stat.h>
int fstat(int fildes, struct stat *buf);
```

Gets file status for the file descriptor *fildes*.

The C libraries provide an interface to this operating system call. Please see your OS manual for a complete definition.

Reference: POSIX, SYS.

ftell( )

```
#include <stdio.h>
long ftell(FILE *stream);
```

See **fseek( )**. Returns the offset of the current byte relative to the beginning of the file associated with the named *stream*.

OS calls: **lseek**.

Reference: ANSI.

fwrite( )

```
#include <stdio.h>
#include <sys/types.h>
int fwrite(const void *ptr, size_t size, int nitems, FILE *stream);
```

Appends at most *nitems* items of data from the array pointed to by *ptr* to the named output *stream*. See **fread( )**.

OS calls: **isatty**, **sbrk**, **write**.

Reference: ANSI.

gamma( )

```
#include <math.h>
double gamma(double x);
extern int signgam;
```

Returns the natural logarithm of the absolute value of the gamma function of *x*. The argument *x* must be a positive integer. The sign of the gamma function is returned as -1 or 1 in *signgam*.

OS calls: **write**.

Reference: UNIX, MATH, REERR.

gammaf( )

```
#include <mathf.h>
float gammaf(float x);
extern int signgamf;
```

Returns the natural logarithm of the absolute value of the gamma function of *x*. The argument *x* must be a positive integer. The sign of the gamma function is returned as -1 or 1 in *signgamf*. This is the single precision version of **gamma( )**.

OS calls: **write**.

Reference: DCC, MATH, REERR.

gcvt( )

```
#include <dcc.h>
char *gcvt(double value, int ndigit, char *buf);
```

See **ecvt( )**. Converts *value* to a null-terminated string in the array pointed to by *buf* and returns *buf*. Produces *ndigit* significant digits in FORTRAN F-format if possible, otherwise E-format. Any minus sign or decimal point will be included as part of the string. Trailing zeros are suppressed.

Reference: DCC.

getc( )

```
#include <stdio.h>
int getc(FILE *stream);
```

Returns the next character (i.e. byte) from the named input *stream*. Moves the file pointer, if defined, ahead one character in *stream*.

OS calls: **isatty**, **read**, **sbrk**, **write**.

Reference: ANSI.

getchar( )

```
#include <stdio.h>
int getchar(void);
```

Same as **getc**, but defined as **getc(stdin)**.

OS calls: **isatty**, **read**, **sbrk**, **write**.

Reference: ANSI.

getenv( )

```
#include <stdlib.h>
char getenv(char *name);
```

Searches the environment list for a string of the form *name=value*, and returns a pointer to value if present, otherwise a null pointer.

Reference: ANSI, REENT.

getopt( )

```
#include <stdio.h>
int getopt(int argc, char *const *argv, const char *optstring);
    extern char *optarg;
    extern int optind, opterr;
```

Returns the next option letter in *argv* that matches a letter in *optstring*, and supports all the rules of the command syntax standard. *optarg* is set to point to the start of the option-argument on return from **getopt( )**. **getopt( )** places the *argv* index of the next argument to be processed in *optind*. Error message output may be disabled by setting *opterr* to 0.

OS calls: **write**.

Reference: SVID.

getpid( )

```
#include <unistd.h>
pid_t getpid(void);
```

Gets process ID.

The C libraries provide an interface to this operating system call. Please see your OS manual for a complete definition.

Reference: POSIX, SYS.

gets( )

```
#include <stdio.h>
char *gets(char *s);
```

Reads characters from **stdin** into the array pointed to by *s*, until a new-line character is read or an **EOF** is encountered. The new-line character is discarded and the string is terminated with a null character. The user is responsible for allocating enough space for the array *s*.

OS calls: **isatty**, **read**, **sbrk**, **write**.

Reference: ANSI.

getw( )

```
#include <stdio.h>
int getw(FILE *stream);
```

Returns the next word (i.e., the next integer) from the named input *stream*, and increments the file pointer, if defined, to point to the next word.

OS calls: **isatty**, **read**, **sbrk**, **write**.

Reference: SVID.

gmtime( )

```
#include <time.h>
struct tm *gmtime(const time_t *timer);
```

Breaks down the calendar time *timer* into sections, expressed as Coordinated Universal Time.

Reference: ANSI.

hcreate( )

```
#include <search.h>
int hcreate(unsigned nel);
```

Allocates sufficient space for a hash table. See **hsearch( )**. The hash table must be allocated before **hsearch( )** is used. *nel* is an estimate of the maximum number of entries the table will contain.

OS calls: **sbrk**.

Reference: SVID.

hdestroy( )

```
#include <search.h>
void hdestroy(void);
```

Destroys the hash table, and may be followed by another call to **hcreate( )**. See **hsearch( )**.

OS calls: **sbrk**, **write**.

Reference: SVID.

hsearch( )

```
#include <search.h>
ENTRY *hsearch(ENTRY item, ACTION action);
```

Hash table search routine which returns a pointer into the hash table, indicating the location where an entry can be found. *item.key* points to a comparison key, and *item.data* points to any other data for that key. *action* is either **ENTER** or **FIND** and indicates the disposition of the entry if it cannot be found in the table. **ENTER** means that *item* should be inserted into the table and **FIND** indicates that no entry should be made.

OS calls: **sbrk**.

Reference: SVID.

hypot( )

```
#include <math.h>
double hypot(double x, double y);
```

Returns sqrt($x * x + y * y$), taking precautions against unwarranted overflows.

Reference: UNIX, MATH, REERR.

hypotf( )

```
#include <mathf.h>
float hypotf(float x, float y);
```

Returns sqrt($x * x + y * y$), taking precautions against unwarranted overflows. This is the single precision version of **hypot( )**.

Reference: DCC, MATH, REERR.

irand48( )

```
#include <stdlib.h>
long irand48(unsigned short n);
```

Generates pseudo-random non-negative long integers uniformly distributed over the interval [0, n-1], using the linear congruential algorithm and 48-bit integer arithmetic. Must be initialized using **srand48( )**, **seed48( )**, or **lcong48( )** functions.

Reference: UNIX.

isalnum( )

```
#include <ctype.h>
int isalnum(int c);
```

Tests for any letter or digit. Returns non-zero if test is true.

Reference: ANSI, REENT.

isalpha( )

```
#include <ctype.h>
int isalpha(int c);
```

Tests for any letter. Returns non-zero if test is true.

Reference: ANSI, REENT.

isascii( )

```
#include <ctype.h>
int isascii(int c);
```

Tests for ASCII character, code between 0 and 0x7f. Returns non-zero if test is true.

Reference: SVID, REENT.

isatty( )

```
#include <unistd.h>
int isatty(int fildes);
```

Tests for a terminal device. Returns non-zero if *fildes* is associated with a terminal device.

Although not a system call in the UNIX environment, it needs to be implemented as such in an embedded environment using the **stdio** functions.

Reference: POSIX.

iscntrl( )

```
#include <ctype.h>
int iscntrl(int c);
```

Tests for control character (0x7f or less than 0x20). Returns non-zero if test is true.

Reference: ANSI, REENT.

isdigit( )

```
#include <ctype.h>
int isdigit(int c);
```

Tests for digit [0-9]. Returns non-zero if test is true.

Reference: ANSI, REENT.

isgraph( )

```
#include <ctype.h>
int isgraph(int c);
```

Tests for printable character not including space. Returns non-zero if test is true.

Reference: ANSI, REENT.

islower( )

```
#include <ctype.h>
int islower(int c);
```

Tests for lower case letter. Returns non-zero if test is true.

Reference: ANSI, REENT.

_isnan( )

```
#include <math.h>
double _isnan(double x);
```

Returns a non-zero value if *x* is a NaN, and returns 0 otherwise.

Reference: ANSI 754, MATH, REENT

isprint( )

```
#include <ctype.h>
int isprint(int c);
```

Tests for printable character (including space). Returns non-zero if test is true.

Reference: ANSI, REENT.

ispunct( )

```
#include <ctype.h>
int ispunct(int c);
```

Tests for printable punctuation character. Returns non-zero if test is true.

Reference: ANSI, REENT.

isspace( )

```
#include <ctype.h>
int isspace(int c);
```

Tests for space, tab, carriage return, new-line, vertical tab, or form-feed. Returns non-zero if test is true.

Reference: ANSI, REENT.

isupper( )

```
#include <ctype.h>
int isupper(int c);
```

Tests for upper-case letters. Returns non-zero if test is true.

Reference: ANSI, REENT.

isxdigit( )

```
#include <ctype.h>
int isxdigit(int c);
```

Tests for hexadecimal digit (0-9, a-f, A-F). Returns non-zero if test is true.

Reference: ANSI, REENT.

j0( )

```
#include <math.h>
double j0(double x);
```

Returns the Bessel function of *x* of the first kind of order 0.

OS calls: **write**.

Reference: UNIX, MATH, REERR.

j0f( )

```
#include <mathf.h>
float j0f(float x);
```

Returns the Bessel function of *x* of the first kind of order 0. This is the single precision version of **j0( )**.

OS calls: **write**.

Reference: DCC, MATH, REERR.

j1( )

```
#include <math.h>
double j1(double x);
```

Returns the Bessel function of *x* of the first kind of order 1.

OS calls: **write**.

Reference: UNIX, MATH, REERR.

j1f( )

```
#include <mathf.h>
float j1f(float x);
```

Returns the Bessel function of *x* of the first kind of order 1. This is the single precision version of **j1( )**.

OS calls: **write**.

Reference: DCC, MATH, REERR.

jn( )

```
#include <math.h>
double jn(double n, double x);
```

Returns the Bessel function of *x* of the first kind of order *n*.

OS calls: **write**.

Reference: UNIX, MATH, REERR.

jnf( )

```
#include <mathf.h>
float jnf(float n, float x);
```

Returns the Bessel function of *x* of the first kind of order *n*. This is the single precision version of **jn( )**.

OS calls: **write**.

Reference: DCC, MATH, REERR.

jrand48( )

```
#include <stdlib.h>
long jrand48(unsigned short xsubi[3]);
```

Generates pseudo-random non-negative long integers uniformly distributed over the interval [-2^{31}, 2^{31}-1], using the linear congruential algorithm and 48-bit integer arithmetic. The calling program must place the initial value Xi into the *xsubi* array and pass it as an argument.

Reference: SVID.

kill( )

```
#include <signal.h>
int kill(int pid, int sig);
```

Sends the signal *sig* to the process *pid*.

The C libraries provide an interface to this operating system call. Please see your OS manual for a complete definition.

Reference: POSIX, SYS.

krand48( )

```
#include <stdlib.h>
long krand48(unsigned short xsubi[3], unsigned short n);
```

Generates pseudo-random non-negative long integers uniformly distributed over the interval [0, n-1], using the linear congruential algorithm and 48-bit integer arithmetic.

Reference: UNIX.

l3tol( )

```
#include <dcc.h>
void l3tol(long *lp, char *cp, int n);
```

Converts the list of *n* three-byte integers packed into the character string pointed to by *cp* into a list of long integers pointed to by **lp*.

Reference: UNIX, REENT.

l64a( )

```
#include <stdlib.h>
char *l64a(long l);
```

Converts the long integer *l* to a base-64 character string.

Reference: SVID.

labs( )

```
#include <stdlib.h>
long labs(long i);
```

Returns the absolute value of *i*.

Reference: ANSI, REENT.

lcong48( )

```
#include <stdlib.h>
void lcong48(unsigned short param[7]);
```

Initialization entry point for **drand48( )**, **lrand48( )**, and **mrand48( )**. Allows the user to specify parameters in the random equation: **Xi** is *param*[0-2], multiplier a is *param*[3-5], and addend c is *param*[6].

Reference: UNIX.

ldexp( )

```
#include <math.h>
double ldexp(double value, int exp);
```

Returns the quantity: *value* * (2^{exp}). See also **frexp( )**.

Reference: UNIX, REERR.

ldexpf( )

```
#include <mathf.h>
float ldexpf(float value, int exp);
```

Returns the quantity: *value* * (2^{exp}). See also **frexpf( )**. This is the single precision version of **ldexp( )**.

Reference: DCC, MATH, REERR.

ldiv( )

```
#include <stdlib.h>
ldiv_t ldiv(long int numer, long int denom);
```

Similar to **div( )**, except that arguments and returned items all have the type **long int**.

Reference: ANSI, REENT.

_lessgreater( )

```
#include <math.h>
double _lessgreater(double x, double y);
```

The value of $x <> y$ is non-zero only when $x < y$ or $x > y$, and is distinct from NOT($x = y$) per Table 4 of the ANSI 754 standard.

Reference: ANSI 754, MATH, REENT.

lfind( )

```
#include <stdio.h>
#include <search.h>
void *lfind(const void *key, const void *base, unsigned *nelp, int size,
    int (*compar)( ));
```

Same as **lsearch( )** except that if datum is not found, it is not added to the table. Instead, a null pointer is returned.

Reference: UNIX, REENT.

link( )

```
#include <unistd.h>
int link(const char *path1, const char *path2);
```

Creates a new link *path2* to the existing file *path1*.

The C libraries provide an interface to this operating system call. Please see your OS manual for a complete definition.

Reference: SYS.

localeconv( )

```
#include <locale.h>
struct lconv *localeconv(void);
```

Loads the components of an object of the type **struct lconv** with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale. See also **setlocale( )**.

Reference: ANSI.

localtime( )

```
#include <time.h>
struct tm *localtime(const time_t *timer);
```

Breaks down the calendar time *timer* into sections, expressed as local time.

Reference: ANSI.

log( )

```
#include <math.h>
double log(double x);
```

Returns the natural logarithm of a positive *x*.

OS calls: **write**.

Reference: ANSI, MATH, REERR.

_logb( )

```
#include <math.h>
double _logb(double x);
```

Returns the unbiased exponent of *x*, a signed integer in the format of *x*, except that **logb(NaN)** is NaN, **logb(infinity)** is +∞, and **logb(0)** is -∞and signals the division by zero exception. When *x* is positive and finite the expression **scalb(***x***, -logb(***x***))** lies strictly between 0 and 2; it is less than 1 only when *x* is denormalized.

Reference: ANSI 754, MATH, REENT.

logf( )

```
#include <mathf.h>
float logf(float x);
```

Returns the natural logarithm of a positive *x*. This is the single precision version of **log( )**.

OS calls: **write**.

Reference: DCC, MATH, REERR.

log10( )

```
#include <math.h>
double log10(double x);
```

Returns the logarithm with base ten of a positive *x*.

OS calls: **write**.

Reference: ANSI, MATH, REERR.

log10f( )

```
#include <mathf.h>
float log10f(float x);
```

Returns the logarithm with base ten of a positive *x*. This is the single precision version of **log10( )**.

OS calls: **write**.

Reference: DCC, MATH, REERR.

longjmp( )

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

Restores the environment saved in *env* by a corresponding **setjmp( )** function call. Execution will continue as if the **setjmp( )** had just returned with the value *val*. If *val* is 0 it will be set to 1 to avoid conflict with the return value from **setjmp( )**.

Reference: ANSI, REENT.

lrand48( )

```
#include <stdlib.h>
long lrand48(void);
```

Generates pseudo-random non-negative long integers uniformly distributed over the interval [0, 2^{31}-1], using the linear congruential algorithm and 48-bit integer arithmetic. Must be initialized using **srand48( )**, **seed48( )**, or **lcong48( )** functions.

Reference: SVID.

lsearch( )

```
#include <stdio.h>
#include <search.h>
void *lsearch(const void *key, const void *base, unsigned *nelp, int size,
    int (*compar)( ));
```

Linear search routine which returns a pointer into a table indicating where a datum may be found. If the datum is not found, it is added to the end of the table. *base* points to the first element in the table. *nelp* points to an integer containing the number of elements in the table. *compar* is a pointer to the comparison function which the user must supply (for example, **strcmp( )**).

Reference: SVID, REENT.

lseek( )

```
#include <unistd.h>
off_t lseek(int fildes, off_t offset, int whence);
```

Moves the file pointer for the file *fildes* to the file offset *offset*. *whence* has one of the following values:

SEEK_SET	offset is absolute position from beginning of file
SEEK_CUR	offset is relative distance from current position
SEEK_END	offset is relative distance from the end of the file

The C libraries provide an interface to this operating system call. Please see your OS manual for a complete definition.

Reference: SYS.

ltol3( )

```
#include <dcc.h>
void ltol3(char *cp, long *lp, int n);
```

Converts a list of long integers to three-byte integers. It is the inverse of **l3tol( )**.

Reference: UNIX, REENT.

mallinfo( )

```
#include <malloc.h>
struct mallinfo mallinfo(void)
```

Used to determine the best setting of **malloc( )** parameters for an application. Must not be called until after **malloc( )** has been called.

Reference: SVID.

malloc( )

```
#include <stdlib.h>
void *malloc(size_t size);
```

Allocates space for an object of size *size*. Returns a pointer to the start (lowest byte address) of the object. Returns a null pointer if no more memory can be obtained by the OS.

The first time **malloc( )** is called, it checks the following environment variables:

DMALLOC_INIT=*n*
: If set, **malloc( )** initializes allocated memory with the byte value *n*. This is useful when debugging programs that may depend on **malloc( )** areas always being set to zero.

DMALLOC_CHECK
: If set, **malloc( )** and **free( )** check the free-list every time they are called. This is useful when debugging programs that may trash the free-list.

NOTE: malloc( ) and related functions must be initialized by the function **__init( )** in **crtlibso.c**. See the note at the end of *15.4.3 Notes for crtlibso.c and ctordtor.c*, p.256 for details. See also *15.10 Reentrant and "Thread-Safe" Library Functions*, p.270.

OS calls: **sbrk**.

Reference: ANSI.

__malloc_set_block_size( )

```
#include <malloc.h>
size_t __malloc_set_block_size(size_t blocksz);
```

To avoid excess execution overhead, **malloc( )** acquires heap space in 8KB master blocks and sub-allocates within each block as required, re-using space within each 8KB block when individual allocations are freed. The default 8KB master block size may be too large on systems with small RAM. To change this, call this **__malloc_set_block_size** function. The argument must be a power of two.

mallopt( )

```
#include <malloc.h>
int mallopt(int cmd, int value);
```

Used to allocate small blocks of memory quickly by allocating a large group of small blocks at one time. This function exists in order to be compatible to SVID, but its use is not recommended, since the **malloc( )** function is already optimized to be fast.

Reference: SVID.

matherr( )

```
#include <math.h>
int matherr(struct exception *x);
```

Invoked by math library routines when errors are detected. Users may define their own procedure for handling errors, by including a function named **matherr( )** in their programs. The function **matherr( )** must be of the form described above. When an error occurs, a pointer to the exception structure *x* will be passed to the user-supplied **matherr( )** function. This structure, which is defined by the <**math.h**> header file, includes the following members:

```
int      type;
char     *name;
double   arg1, arg2, retval;
```

The member **type** is an integer describing the type of error that has occurred from the following list defined by the <**math.h**> header file:

DOMAIN	argument domain error
SING	argument singularity
OVERFLOW	overflow range error
UNDERFLOW	underflow range error
TLOSS	total loss of significance
PLOSS	partial loss of significance

The member **name** points to a string containing the name of the routine that incurred the error. The members **arg1** and **arg2** are the first and second arguments with which the routine was invoked.

The member **retval** is set to the default value that will be returned by the routine unless the user's **matherr( )** function sets it to a different value.

If the user's **matherr( )** function returns non-zero, no error message will be printed, and **errno** will not be set.

If the function **matherr( )** is not supplied by the user, the default error-handling procedures, described with the math library routines involved, will be invoked upon error. **errno** is set to **EDOM** or **ERANGE** and the program continues.

Reference: SVID, MATH.

matherrf( )

```
#include <mathf.h>
int matherrf(struct exceptionf *x);
```

This is the single precision version of **matherr( )**.

Reference: DCC, MATH.

mblen( )

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

If s is not a null pointer, the function returns the number of bytes in the string s that constitute the next multi-byte character, or -1 if the next *n* (or the remaining bytes) do not compromise a valid multi-byte character. A terminating null character is not included in the character count. If *s* is a null pointer and the multi-byte characters have a state-dependent encoding in current locale, the function returns nonzero; otherwise, it returns zero.

Reference: ANSI, REENT.

mbstowcs( )

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwc, const char *s, size_t n);
```

Stores a wide character string in the array whose first element has the address *pwc*, by converting the multi-byte characters in the string *s*. It converts as if by calling **mbtowc( )**. It stores at most *n* wide characters, stopping after it stores a null wide character. It returns the number of wide characters stored, not counting the null character.

Reference: ANSI, REENT.

mbtowc( )

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

If *s* is not a null pointer, the function returns the number of bytes in the string *s* that constitute the next multi-byte character. (The number of bytes cannot be greater than **MB_CUR_MAX**). If *pwc* is not a null pointer, the next multi-byte character is converted to the corresponding wide character value and stored in **pwc*. The function returns -1 if the next *n* or the remaining bytes do not constitute a valid multi-byte character. If *s* is a null pointer and multi-byte characters have a state-dependent encoding in current locale, the function stores an initial shift state in its internal static duration data object and returns nonzero; otherwise it returns zero.

Reference: ANSI, REENT.

memccpy( )

```
#include <string.h>
void *memccpy(void *s1, const void *s2, int c, size_t n);
```

Copies characters from *s2* into *s1*, stopping after the first occurrence of character *c* has been copied, or after *n* characters, whichever comes first.

Reference: SVID, REENT.

memchr( )

```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
```

Locates the first occurrence of *c* (converted to unsigned char) in the initial *n* characters of the object pointed to by *s*. Returns a null pointer if *c* is not found.

Reference: ANSI, REENT.

memcmp( )

```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```


Compares the first *n* character of *s1* to the first *n* characters of *s2*. Returns an integer greater than, equal to, or less than zero according to the relationship between *s1* and *s2*.

Reference: ANSI, REENT.

memcpy( )

```
#include <string.h>
void *memcpy(void *s1, const void *s2, size_t n);
```

Copies *n* character from the object pointed to by *s2* into the object pointed to by *s1*. The behavior is undefined if the objects overlap. Returns the value of *s1*.

Reference: ANSI, REENT.

memmove( )

```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

Copies *n* characters from the object pointed by *s2* into the object pointed to by *s1*. It can handle overlapping while copying takes place as if the *n* characters were first copied to a temporary array, then copied into *s1*. Returns the value of *s1*.

Reference: ANSI, REENT.

memset( )

```
#include <string.h>
void *memset(void *s, int c, size_t n);
```

Copies the value of *c* into each of the first *n* characters of the object pointed to by *s*. Returns the value of *s*.

Reference: ANSI, REENT.

mktemp( )

```
#include <stdio.h>
char *mktemp (char *template);
```

Replaces the contents of the string pointed to by *template* with a unique filename, and returns the address of *template*. The *template* string should look like a filename with six trailing **X**s, which will be replaced with a letter and the current process ID.

OS calls: **access**, **getpid**.

Reference: SVID.

mktime( )

```
#include <time.h>
time_t mktime(struct tm *timeptr);
```

Converts the local time stored in *timeptr* into a calendar time with the same encoding as values returned by the **time( )** function, but with all values within their normal ranges. It sets the structure members **tm_mday**, **tm_wday**, **tm_yday**.

Reference: ANSI, REENT.

modf( )

```
#include <math.h>
double modf(double value, double *iptr);
```

Returns the fractional part of *value* and stores the integral part in the location pointed to by *iptr*. Both the fractional and integer parts have the same sign as *value*. See also **frexp( )**.

Reference: ANSI, REENT.

modff( )

```
#include <mathf.h>
float modff(float value, float *iptr);
```

Returns the fractional part of *value* and stores the integral part in the location pointed to by *iptr*. Both the fractional and integer parts have the same sign as *value*. See also **frexpf( )**. This is the single precision version of **modf( )**.

Reference: DCC, MATH, REENT.

mrand48( )


```
#include <stdlib.h>
long mrand48(void);
```

Generates pseudo-random non-negative long integers uniformly distributed over the interval [-2^{31}, 2^{31}-1], using the linear congruential algorithm and 48-bit integer arithmetic. Must be initialized using **srand48( )**, **seed48( )**, or **lcong48( )** functions.

Reference: SVID.

_nextafter( )

```
#include <math.h>
double _nextafter(double x, double y);
```

Returns the next representable neighbor of *x* in the direction toward *y*. The following special cases arise: if *x* = *y*, then the result is *x* without any exception being signaled; otherwise, if either *x* or *y* is a quiet NaN, then the result is one or the other of the input NaNs. Overflow is signaled when *x* is finite but **_nextafter(***x*, *y***)** lies strictly between $+2^{Emin}$ and -2^{Emin}. In both cases, inexact is signaled.

Reference: ANSI 754, MATH, REENT.

nrand48( )

```
#include <stdlib.h>
long nrand48(unsigned short xsubi[3]);
```

Generates pseudo-random non-negative long integers uniformly distributed over the interval [0, 2^{31}-1], using the linear congruential algorithm and 48-bit integer arithmetic.

Reference: SVID.

offsetof( )

```
#include <stddef.h>
size_t offsetof(type, member);
```

Returns the offset of the member *member* in the structure *type*. Implemented as a macro.

Reference: ANSI, REENT.

open( )

```
#include <fcntl.h>
int open(const char *path, int oflag, int mode);
```

Opens the file *path* for reading or writing according to *oflag*. Usual values of *oflag* are:

O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing

The C libraries provide an interface to this operating system call. Please see your OS manual for a complete definition.

Reference: POSIX, SYS.

perror( )

```
#include <stdio.h>
void perror(const char *s);

extern int errno;
extern char *sys_errlist[];
extern int sys_nerr;
```

Produces a message on the standard error output describing the last error encountered during a call to a system or library function. The array of message strings **sys_errlist[]** may be indexed by **errno** to access the message string directly without the new-line. **sys_nerr** is the number of messages in the table. See **strerror( )**.

OS calls: **write**.

Reference: ANSI.

pow( )

```
#include <math.h>
double pow(double x, double y);
```

Returns the value of x^y. If x is zero, y must be positive. If x is negative, y must be an integer.

OS calls: **write**.

Reference: ANSI, MATH, REERR.

powf( )

```
#include <mathf.h>
float powf(float x, float y);
```

Returns the value of x^y. If x is zero, y must be positive. If x is negative, y must be an integer. This is the single precision version of **pow( )**.

OS calls: **write**.

Reference: DCC, MATH, REERR.

printf( )

```
#include <stdio.h>
int printf(const char *format, ... );
```

Places output arguments on **stdout**, controlled by *format*. Returns the number of characters transmitted or a negative value if there was an error. A summary of the **printf( )** conversion specifiers is shown below. Each conversion specification is introduced by the character **%**. Conversion specifications within brackets are optional.

% {*flags*} {*field_width*} {**.***precision*} {*length_modifier*} *conversion*

flags

Single characters which modify the operation of the format as follows:

-
left adjusted field

+
signed values will always begin with plus or minus sign

space
values will always begin with minus or space

#
Alternate form. Has the following effect: For **o** (octal) conversion, the first digit will always be a zero. **G**, **g**, **E**, **e** and **f** conversions will always print a decimal point. **G** and **g** conversions will also keep trailing zeros. **X**, **x** (hex) and **p** conversions will prepend non-zero values with **0x** (or **0X**)

0
zero padding to field width (for **d**, **i**, **ll**, **o**, **q**, **u**, **x**, **X**, **e**, **E**, **f**, **g**, and **G** conversions)

field_width
Number of characters to be printed in the field. Field width will be padded with space if needed. If given as "*", the next argument should be an integer holding the field width.

.precision
Minimum number of digits to print for integers (**d**, **i**, **ll**, **o**, **q**, **u**, **x**, and **X**). Number of decimals printed for floating point values (**e**, **E**, and **f**). Maximum number of significant digits for **g** and **G** conversions. Maximum number of characters for **s** conversion. If given as "*" the next argument should be an integer holding the precision.

length_modifier
The following length modifiers are used:

h
Used before **d**, **i**, **o**, **n**, **u**, **x**, or **X** conversions to denote a **short int** or **unsigned short int** value.

l
Used before **d**, **i**, **o**, **n**, **u**, **x**, or **X** conversions to denote a **long int** or **unsigned long int** value.

L
Used before **e**, **E**, **f**, **g**, or **G** conversions to denote a **long double** value. Used before **d**, **i**, **o**, **u**, **x**, or **X** conversions to denote a **long long** value.

conversion
The following conversion specifiers are used:

d
Write signed decimal integer value.

i
Write signed decimal integer value.

ll
Write signed **long long** decimal integer value.

o
Write unsigned octal integer value.

q
Write signed **long long** decimal integer value.

u
Write unsigned decimal integer value.

x
Write unsigned hexadecimal (0-9, abc...) integer value.

X
Write unsigned hexadecimal (0-9, ABC...) integer value.

e
Write floating point value: [-]d.ddde+dd .

E
Write floating point value: [-]d.dddE+dd .

f
Write floating point value: [-]ddd.ddd .

g
Write floating point value in **f** or **e** notation depending on the size of the value (“best” fit conversion).

G
Write floating point value in **f** or **E** notation depending on the size of the value (“best” fit conversion).

c
Write a single character.

s
Write a string.

p
Write a pointer value (address).

n
Store current number of characters written so far. The argument should be a pointer to integer.

%
Write a percentage character.

The floating point values Infinity and Not-A-Number are printed as **inf**, **INF**, **nan**, and **NAN** when using the **e**, **E**, **f**, **g**, or **G** conversions.

NOTE: By default in most environments, **printf** buffers its output until a newline is output. To cause output character-by-character without waiting for a newline, call *setbuf( )*, p.520, with a NULL buffer pointer after opening but before writing to the stream:

```
setbuf(*stream, 0);
```

OS calls: **isatty**, **sbrk**, **write**.

Reference: ANSI.

putc( )

```
#include <stdio.h>
int putc(int c, FILE *stream)
```

Writes the character *c* onto the output *stream* at the position where the file pointer, if defined, is pointing.

OS calls: **isatty**, **sbrk**, **write**.

Reference: ANSI.

putchar( )

```
#include <stdio.h>
int putchar(int c)
```

Similar to **putc( )** but writes to **stdout**.

OS calls: **isatty**, **sbrk**, **write**.

Reference: ANSI.

putenv( )

```
#include <stdlib.h>
int putenv(char *string);
```

string points to a string of the form *name=value*, and **putenv( )** makes the value of the environmental variable *name* equal to *value*. The string pointed to by *string* becomes part of the environment, so altering *string* alters the environment.

OS calls: **sbrk**, **write**.

Reference: SVID.

puts( )

```
#include <stdio.h>
int puts(const char *s);
```

Writes the null-terminated string pointed to by *s*, followed by a new-line character, to **stdout**.

OS calls: **isatty**, **sbrk**, **write**.

Reference: ANSI.

putw( )

```
#include <stdio.h>
int putw(int w, FILE *stream)
```

Writes the word (i.e., integer) *w* to the output *stream* at the position at which the file pointer, if defined, is pointing.

OS calls: **isatty**, **sbrk**, **write**.

Reference: SVID.

qsort( )

```
#include <stdlib.h>
void qsort(void *base, size_t nel, size_t size, int (*compar)( ));
```

Sorts a table in place using the quick-sort algorithm. *base* points to the element at the base of the table, *nel* is the number of elements. *size* is the size of each element. *compar* is a pointer to the user supplied comparison function, which is called with two arguments that point to the elements being compared.

Reference: ANSI, REENT.

raise( )

```
#include <signal.h>
int raise(int sig);
```

Sends the signal *sig* to the executing program.

OS calls: **getpid**, **kill**.

Reference: ANSI.

rand( )

```
#include <stdlib.h>
int rand(void);
```

Returns a pseudo random number in the interval [0, **RAND_MAX**].

Reference: ANSI.

read( )

```
#include <unistd.h>
int read(int fildes, void *buf, unsigned nbyte);
```

Reads max *nbyte* bytes from the file associated with the file descriptor *fildes* to the buffer pointed to by *buf*.

The C libraries provide an interface to this operating system call. Please see your OS manual for a complete definition.

Reference: SYS.

realloc( )

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
extern int __no_malloc_warning;
```

Changes the size of the object pointed to by *ptr* to the size *size*. *ptr* must have received its value from **malloc( )**, **calloc( )**, or **realloc( )**. Returns a pointer to the

start address of the possibly moved object, or a null pointer if no more memory can be obtained from the OS.

If the pointer *ptr* was freed or not allocated by **malloc( )**, a warning is printed on the **stderr** stream. The warning can be suppressed by assigning a non-zero value to the integer variable **__no_malloc_warning**. See **malloc( )** for more information.

OS calls: **sbrk**, **write**.

Reference: ANSI.

remove( )

```
#include <stdio.h>
int remove(const char *filename);
```

Removes the file *filename*. Once removed, the file cannot be opened as an existing file.

OS calls: **unlink**.

Reference: ANSI.

rename( )

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

Renames the file *old* to the file *new*. Once renamed, the file *old* cannot be opened again.

OS calls: **link**, **unlink**.

Reference: ANSI.

rewind( )

```
#include <stdio.h>
void rewind(FILE *stream);
```

Same as **fseek(***stream***, 0L, 0)**, except that no value is returned.

OS calls: **isatty**, **read**, **sbrk**, **write**.

Reference: ANSI.

sbrk( )

```
#include <unistd.h>
void *sbrk(int incr);
```

Gets *incr* bytes of memory from the operating system.

The C libraries provide an interface to this operating system call. Please see your OS manual for a complete definition.

Reference: UNIX, SYS.

_scalb( )

```
#include <math.h>
double _scalb(double x, int N);
```

Returns $y * 2^N$ for integeral values N without computing 2^N.

Reference: ANSI 754, MATH, REENT.

scanf( )

```
#include <stdio.h>
int scanf(const char *format, ...);
```

Reads formatted data from **stdin** and optionally assigns converted data to variables specified by the *format* string. Returns the number of successful conversions (or **EOF** if input is exhausted).

If the format string contains white-space characters, input is scanned until a non-white-space character is found.

A conversion specification is introduced by the character **%**.

If the format string neither contains a white-space nor a %, the format string and the input characters must match exactly.

A summary of the **scanf( )** conversion specifiers is shown below. Conversion specifications within braces are optional.

% {*} {*field_width*} {*length_modifier*} *conversion*

*****
: No assignment should be done (just scan the field).

field_width
: Maximum field to be scanned (default is until no match occurs).

length_modifier

The following length modifiers are used:

l

Used before **d**, **i**, or **n** to indicate **long int** or before **o**, **u**, **x** to denote the presence of an **unsigned long int**. For **e**, **E**, **g**, **G**, and **f** conversions the l character implies a **double** operand.

h

Used before **d**, **i**, or **n** to indicate **short int** or before **o**, **u**, or **x** to denote the presence of an **unsigned short int**.

L

For **e**, **E**, **g**, **G**, and **f** conversions the **L** character implies a **long double** operand. For **d**, **i**, **o**, **u**, **x**, and **X** conversions the **L** character implies a **long long** operand.

conversion

The following conversions are available:

d

Read an optionally signed decimal integer value.

i

Read an optionally signed integer value in standard C notation. Default is decimal notation, but octal (0n) and hex (0xn, 0Xn) notations are also recognized.

ll

Read an optionally signed **long long** decimal integer value.

o

Read an optionally signed octal integer.

q

Read an optionally signed **long long** decimal integer value.

u

Read an unsigned decimal integer.

x, **X**

Read an optionally signed hexadecimal integer.

f, **e**, **E**, **g**, **G**

Read a floating point constant.

s

Read a character string.

c
Read *field_width* number of characters (1 is default).

n
Store the number of characters read so far. The argument should be a pointer to an integer.

p
Read a pointer value (address).

[
Read characters as long as they match any of the characters that are within the terminating]. If the first character after [is a ^, the matching condition is reversed. If the [is immediately followed by] or ^], the] is assumed to belong to the matching sequence, and there must be another terminating character. A range of characters may be represented by first-last, thus [a-f] equals [abcdef].

%
Read a **%** character.

Notes: Except for the **[**, **c**, or **n** specifiers leading white-space characters are skipped. Variables must always be expressed as addresses in order to be assignable by **scanf**.

OS calls: **isatty**, **read**, **sbrk**, **write**.

Reference: ANSI.

seed48( )

```
#include <stdlib.h>
unsigned short *seed48(unsigned short seed16v[3]);
```

Initialization entry point for **drand48( )**, **lrand48( )**, and **mrand48( )**.

Reference: SVID.

setbuf( )

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
```

May be used after the *stream* has been opened but before reading or writing to it. It causes the array pointed to by *buf* to be used instead of an automatically allocated

buffer. If *buf* is the null pointer, then input/output will be unbuffered. The constant **BUFSIZ** in <**stdio.h**> defines the required size of *buf*.

OS calls: **isatty**, **sbrk**, **write**.

Reference: ANSI.

setjmp( )


```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Saves the current execution environment in *env* for use by the **longjmp( )** function. Returns 0 when invoked by **setjmp( )** and a non-zero value when returning from a **longjmp( )** call.

Reference: ANSI, REENT.

setlocale( )

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

Selects the appropriate portion of the program's locale as specified by the *category* and *locale* arguments. Can be used to change or query the program's entire locale with the category **LC_ALL**; the other values for *category* name only portions of the program's locale. **LC_COLLATE** affects the behavior of the **strcoll( )** and **strxfrm( )** functions. **LC_CTYPE** affects the behavior of the character handling functions and the multi-byte functions. **LC_MONETARY** affects the monetary formatting information returned by the **localeconv( )** function. **LC_NUMERIC** affects the decimal-point character for the formatted input/output functions and the string conversion functions, as well as the non-monetary formatting information returned by the **localeconv( )** function. **LC_TIME** affects the behavior of the **strftime( )** function.

A value of "**C**" for *locale* specifies the minimal environment for C translation; a value of **""** for *locale* specifies the implementation-defined native environment. Other implementation-defined strings may be passed as the second argument to **setlocale( )**.

At program start-up, the equivalent of **setlocale(LC_ALL, "C")** is executed.

The compiler currently supports only the "**C**" locale.

Reference: ANSI.

setvbuf( )

```
#include <stdio.h>
void setvbuf(FILE *stream, char *buf, int type, size_t size);
```

See **setbuf( )**. *type* determines how the *stream* will be buffered:

_IOFBF	causes stream to be fully buffered
_IOLBF	causes stream to be line buffered
_IONBF	causes stream to be unbuffered

size specifies the size of the buffer to be used; **BUFSIZ** in <**stdio.h**> is the suggested size.

OS calls: **sbrk**, **write**.

Reference: ANSI.

signal( )

```
#include <signal.h>
void (*signal(int sig, void (*func)( )))(void);
```

Specifies the action on delivery of a signal. When the signal *sig* is delivered, a signal handler specified by *func* is called.

The C libraries provide an interface to this operating system call. Please see your OS manual for a complete definition.

Reference: ANSI, SYS.

sin( )

```
#include <math.h>
double sin(double x);
```

Returns the sine of *x* measured in radians. It loses accuracy with a large argument value.

OS calls: **write**.

Reference: ANSI, MATH, REERR.

sinf( )

```
#include <mathf.h>
float sinf(float x);
```

Returns the sine of *x* measured in radians. It loses accuracy with a large argument value. This is the single precision version of **sin( )**.

OS calls: **write**.

Reference: DCC, MATH, REERR.

sinh( )

```
#include <math.h>
 double sinh(double x);
```

Returns the hyperbolic sine of *x* measured in radians. It loses accuracy with a large argument value.

Reference: ANSI, MATH, REERR.

sinhf( )

```
#include <mathf.h>
float sinhf(float x);
```

Returns the hyperbolic sine of *x* measured in radians. It loses accuracy with a large argument value. This is the single precision version of **sinh( )**.

Reference: DCC, MATH, REERR.

sprintf( )

```
#include <stdio.h>
int sprintf(char *s, const char *format , ...);
```

Places output arguments followed by the null character in consecutive bytes starting at ***s**; the user must ensure that enough storage is available. See **printf( )**.

Reference: ANSI, REENT.

sqrt( )

```
#include <math.h>
double sqrt(double x);
```

Returns the non-negative square root of *x*. The argument must be non-negative.

OS calls: **write**.

Reference: ANSI, MATH, REERR.

sqrtf( )

```
#include <mathf.h>
float sqrtf(float x);
```

Returns the non-negative square root of *x*. The argument must be non-negative. This is the single precision version of **sqrt( )**.

OS calls: **write**.

Reference: DCC, MATH, REERR.

srand( )

```
#include <stdlib.h>
void srand(unsigned seed);
```

Resets the random-number generator to a random starting point. See **rand( )**.

Reference: ANSI.

srand48( )

```
#include <stdlib.h>
void srand48(long seedval);
```

Initialization entry point for **drand48( )**, **lrand48( )**, and **mrand48( )**.

Reference: SVID.

sscanf( )

```
#include <stdio.h>
int sscanf(const char *s, const char *format, ...);
```

Reads formatted data from the character string *s*, optionally assigning converted data to variables specified by the *format* string. It returns the number of successful conversions (or **EOF** if input is exhausted). See **scanf( )**.

Reference: ANSI, REENT.

step( )


```
#include <regexp.h>
int step(char *string, char *expbuf);
```

Does pattern matching given the string *string* and a compiled regular expression *expbuf*. See SVID for more details.

Reference: SVID.

strcat( )

```
#include <string.h>
char *strcat(char *s1, const char *s2);
```

Appends a copy of the string pointed to by *s2* (including a null character) to the end of the string pointed to by *s1*. The initial character of *s2* overwrites the null character at the end of *s1*. The behavior is undefined if the objects overlap.

Reference: ANSI, REENT.

strchr( )

```
#include <string.h>
char *strchr(const char *s, int c);
```

Locates the first occurrence of *c* in the string pointed to by *s*.

Reference: ANSI, REENT.

strcmp( )

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

Compares *s1* to *s2*. Returns an integer greater than, equal to, or less than zero according to the relationship between *s1* and *s2*.

Reference: ANSI, REENT.

strcoll( )

```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

Compares *s1* to *s2*, both interpreted as appropriate to the **LC_COLLATE** category of the current locale. Returns an integer greater than, equal to, or less than zero according to the relationship between *s1* and *s2*.

Reference: ANSI, REENT.

strcpy( )

```
#include <string.h>
char *strcpy(char *s1, const char *s2);
```

Copies the string pointed to by *s2* (including a terminating null character) into the array pointed to by *s1*. The behavior is undefined if the objects overlap.

Reference: ANSI, REENT.

strcspn( )

```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
```

Computes the length of the maximum initial segment of *s1* which consists entirely of characters not from *s2*.

Reference: ANSI, REENT.

strdup( )

```
#include <string.h>
char *strdup(const char *s1);
```

Returns a pointer to a new string which is a duplicate of *s1*.

OS calls: **sbrk**.

Reference: SVID.

strerror( )

```
#include <string.h>
char *strerror(int errnum);
```

Maps the error number in *errnum* to an error message string.

Reference: ANSI, REENT.

strftime( )

```
#include <time.h>
size_t strftime(char *s, size_t maxsize, const char *format,
                const struct tm *timeptr);
```

Uses the format *format* and values in the structure *timeptr* to generate formatted text. Generated characters are stored in successive locations in the array pointed to by *s*. It stores a null character in the next location in the array. Each non-% character is stored in the array. For each % followed by a character, a replacement character sequence is stored as shown below. Examples are in parenthesis.

%a	abbreviated weekday name (Mon)
%A	full weekday name (Monday)
%b	abbreviated month name (Jan)
%B	full month name (January)
%c	date and time (Jan 03 07:22:43 1990)
%d	day of the month (04)
%H	hour of the 24-hour day (13)
%I	hour of the 12-hour day (9)
%j	day of the year, Jan 1 = 001 (322)
%m	month of the year (11)
%M	minutes after the hour (43)
%p	AM/PM indicator (PM)
%S	seconds after the minute (37)
%U	Sunday week of the year, from 00 (34)
%w	weekday number, Sunday = 0 (3)

%W Monday week of the year, from 00 (23)

%x date (Jan 23 1990)

%X time (23:33:45)

%y year of the century (90)

%Y year (1990)

%Z time zone name (PST)

%% percent character (%)

Reference: ANSI, REENT.

strlen( )

```
#include <string.h>
size_t strlen(const char *s);
```

Computes the length of the string *s*.

Reference: ANSI, REENT.

strncat( )

```
#include <string.h>
char *strncat(char *s1, const char *s2, size_t n);
```

Appends not more than *n* characters from the string pointed to by *s2* to the end of the string pointed to by *s1*. The initial character of *s2* overwrites the null character at the end of *s1*. The behavior is undefined if the objects overlap. A terminating null character is always appended to the result.

Reference: ANSI, REENT.

strncmp( )

```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n);
```

Compares not more than *n* characters (characters after a null character are ignored) in *s1* to *s2*. Returns an integer greater than, equal to, or less than zero according to the relationship between *s1* and *s2*.

Reference: ANSI, REENT.

strncpy( )

```
#include <string.h>
char *strncpy(char *s1, const char *s2, size_t n);
```

Copies not more than *n* characters from the string pointed to by *s2* (including a terminating null character) into the array pointed to by *s1*. The behavior is undefined if the objects overlap. If *s2* is shorter than *n*, null characters are appended.

Reference: ANSI, REENT.

strpbrk( )

```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
```

Locates the first occurrence of any character from the string pointed to by *s2* within the string pointed to by *s1*.

Reference: ANSI, REENT.

strrchr( )

```
#include <string.h>
char *strrchr(const char *s, int c);
```

Locates the last occurrence of *c* within the string pointed to by *s*.

Reference: ANSI, REENT.

strspn( )

```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

Computes the length of the maximum initial segment of *s1* which consists entirely of characters from *s2*.

Reference: ANSI, REENT.

strstr( )

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

Locates the first occurrence of the sequence of characters (not including a null character) in the string pointed to by *s2* within the string pointed to by *s1*.

Reference: ANSI, REENT.

strtod( )

```
#include <stdlib.h>
double strtod(const char *str, char **endptr);
```

Returns as a double-precision floating point number the value represented by the character string pointed to by *str*. The string is scanned to the first unrecognized character. Recognized characters include optional white-space character(s), optional sign, a string of digits optionally containing a decimal point, optional **e** or **E** followed by an optional sign or space, followed by an integer. At return, the pointer at **endptr* is set to the first unrecognized character.

Reference: ANSI, REERR.

strtok( )

```
#include <string.h>
char *strtok(char *s1, const char *s2);
```

Searches string *s1* for address of the first element that equals none of the elements in string *s2*. If the search does not find an element, it stores the address of the terminating null character in the internal static duration data object and returns a null pointer. Otherwise, searches from found address to address of the first element that equals any one of the elements in string *s2*. If it does not find element, it stores address of the terminating null character in the internal static duration data object. Otherwise, it stores a null character in the element whose address was found in second search. Then it stores address of the next element after end in the internal duration data object (so next search starts at that address) and returns address found in initial search.

Reference: ANSI.

strtol( )

```
#include <stdlib.h>
long strtol(const char *str, char **endptr, int base);
```

Returns as a long integer the value represented by the character string pointed to by *str*. The string is scanned to the first character inconsistent with the base. Leading white-space characters are ignored. At return, the pointer at **endptr* is set to the first unrecognized character.

If *base* is positive and less then 37, it is used as the base for conversion. After an optional sign, leading zeros are ignored, and "**0x**" or "**0X**" is ignored if *base* is 16.

If *base* is zero, the string itself determines the base: after an optional leading sign a leading zero indicates octal, a leading "**0x**" or "**0X**" indicates hexadecimal, else decimal conversion is used.

Reference: ANSI, REERR.

strtoul( )

```
#include <stdlib.h>
long strtoul(const char *, char **endptr, int base);
```

Returns as an unsigned long integer the value represented by the character string pointed to by *s*. The string is scanned to the first character inconsistent with the base. Leading white-space characters are ignored. This is the same as **strtol( )**, except that it reports a range error only if the value is too large to be represented as the type **unsigned long**.

Reference: ANSI, REERR.

strxfrm( )

```
#include <string.h>
size_t strxfrm(char *s1, char *s2, size_t n);
```

Transforms *s2* and places the result in *s1*. No more than *n* characters are put in *s1*, including the terminating null character. The transformation is such that if **strcmp( )** is applied to the two strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the **strcoll( )** function applied to the same two original strings. Copying between objects that overlap causes undefined results.

Reference: ANSI, REENT.

swab( )

```
#include <dcc.h>
void swab(const char *from, char *to, int nbytes)
```

Copies *nbytes* bytes pointed to by *from* to the array pointed to by *to*. *nbytes* must be even and non-negative. Adjacent even and odd bytes are exchanged.

Reference: SVID, REENT.

tan( )

```
#include <math.h>
double tan(double x);
```

Returns the tangent of *x* measured in radians.

OS calls: **write**.

Reference: ANSI, MATH, REERR.

tanf( )

```
#include <mathf.h>
float tanf(float x);
```

Returns the tangent of *x* measured in radians. This is the single precision version of **tan( )**.

OS calls: **write**.

Reference: DCC, MATH, REERR.

tanh( )

```
#include <math.h>
double tanh(double x);
```

Returns the hyperbolic tangent of *x* measured in radians.

Reference: ANSI, MATH, REENT.

tanhf( )

```
#include <mathf.h>
float tanhf(float x);
```

Returns the hyperbolic tangent of *x* measured in radians. This is the single precision version of **tanh( )**.

Reference: DCC, MATH, REENT.

tdelete( )

```
#include <search.h>
 void *tdelete(const void *key, void **rootp, int (*compar)( ));
```

The **tdelete( )** function deletes a node from a binary search tree. The value for *rootp* will be changed if the deleted node was the root of the tree. Returns a pointer to the parent of the deleted node. See **tsearch( )**.

Reference: SVID.

tell( )

```
#include <dcc.h>
 long tell(int fildes);
```

Returns the current location in the file descriptor *fildes*. This is the same as **lseek(***fildes***,0L,1)**.

OS calls: **lseek**.

Reference: DCC.

tempnam( )

```
#include <stdio.h>
char *tempnam(const char *dir, const char *pfx);
```

Creates a unique filename, allowing control of the choice of directory. If the **TMPDIR** variable is specified in the user's environment, it is used as the temporary file directory. Otherwise, the argument *dir* points to the name of the directory in which the file is to be created. If *dir* is invalid, the path-prefix **P_tmpdir** (<**stdio.h**>) is used. If **P_tmpdir** is invalid, **/tmp** is used. See **tmpnam( )**.

Reference: SVID.

tfind( )

```
#include <search.h>
void *tfind(void *key, void *const *rootp, int (*compar)( ));
```

tfind( ) will search for a datum in a binary tree, and return a pointer to it if found, otherwise it returns a null pointer. See **tsearch( )**.

Reference: SVID, REENT.

time( )

```
#include <time.h>
time_t time(time_t *timer);
```

Returns the system time. If *timer* is not a null pointer, the time value is stored in **timer*.

The C libraries provide an interface to this operating system call. Please see your OS manual for a complete definition.

Reference: ANSI, SYS.

tmpfile( )

```
#include <stdio.h>
FILE *tmpfile(void);
```

Creates a temporary file using a name generated by **tmpnam( )** and returns the corresponding **FILE** pointer. File is opened for update ("**w+**"), and is automatically deleted when the process using it terminates.

OS calls: **lseek**, **open**, **unlink**.

Reference: ANSI.

tmpnam( )

```
#include <stdio.h>
char *tmpnam(char *s);
```

Creates a unique filename using the path-prefix defined as **P_tmpdir** in <**stdio.h**>. If *s* is a null pointer, **tmpnam( )** leaves the result in an internal static area and returns a pointer to that area. At the next call to **tmpnam( )**, it will destroy the contents of the area. If s is not a null pointer, it is assumed to be the address of an

array of at least **L_tmpnam** bytes (defined in <**stdio.h**>); **tmpnam( )** places the result in that array and returns *s*.

OS calls: **access**, **getpid**.

Reference: ANSI.

toascii( )


```
#include <ctype.h>
int toascii(int c);
```

Turns off all bits in the argument *c* that are not part of a standard ASCII character; for compatibility with other systems.

Reference: SVID, REENT.

tolower( )

```
#include <ctype.h>
int tolower(int c);
```

Converts an upper-case letter to the corresponding lower-case letter. The argument range is -1 through 255, any other argument is unchanged.

Reference: ANSI, REENT.

_tolower( )

```
#include <ctype.h>
int _tolower(int c);
```

Converts an upper-case letter to the corresponding lower-case letter. Arguments outside lower-case letters return undefined results. The speed is somewhat faster than **tolower( )**.

Reference: SVID, REENT.

toupper( )

```
#include <ctype.h>
int toupper(int c);
```

Converts a lower-case letter to the corresponding upper-case letter. The argument range is -1 through 255, any other argument is unchanged.

Reference: ANSI, REENT.

_toupper( )

```
#include <ctype.h>
int _toupper(int c);
```

Converts a lower-case letter to the corresponding upper-case letter. Arguments outside lower-case letters return undefined results. The speed is somewhat faster than **toupper( )**.

Reference: SVID, REENT.

tsearch( )

```
#include <search.h>
void *tsearch(const void *key, void ** rootp, int (*compar)( ));
```

Used to build and access a binary tree. The user supplies the routine *compar* to perform comparisons. *key* is a pointer to a datum to be accessed or stored. If a datum equal to **key* is in the tree, a pointer to that datum is returned. Otherwise, **key* is inserted, and a pointer to it is returned. *rootp* points to a variable that points to the root of the tree.

Reference: SVID.

twalk( )

```
#include <search.h>
void twalk(void *root, void (*action)( ));
```

twalk( ) traverses a binary tree. *root* is the root of the tree to be traversed, and any node may be the root for a walk below that node. *action* is the name of the user supplied routine to be invoked at each node, and is called with three arguments. The first argument is the address of the node being visited. The second argument is a value from the enumeration data type **typedef enum {preorder, postorder, endorder, leaf} VISIT** (see <**search.h**>), depending on whether this is the first, second, or third time the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root as level zero. See **tsearch( )**.

Reference: SVID, REENT.

tzset( )

```
#include <sys/types.h>
#include <time.h>
void tzset(void);
```

tzset( ) uses the contents of the environment variable **TZ** to override the value of the different external variables for the time zone. It scans the contents of **TZ** and assigns the different fields to the respective variable. **tzset( )** is called by **asctime( )** and may be called explicitly by the user.

Reference: POSIX.

ungetc( )

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Inserts character *c* into the buffer associated with input *stream*. The argument *c* will be returned at the next **getc( )** call on that stream. **ungetc( )** returns *c* and leaves the file associated with *stream* unchanged. If *c* equals **EOF**, **ungetc( )** does nothing to the buffer and returns **EOF**. Only one character of push-back is guaranteed.

Reference: ANSI.

unlink( )

```
#include <unistd.h>
int unlink(const char *path);
```

Removes the directory entry *path*.

The C libraries provide an interface to this operating system call. Please see your OS manual for a complete definition.

Reference: POSIX, SYS.

_unordered( )

```
#include <math.h>
double _unordered(double x, double y);
```

Returns a non-zero value if *x* is unordered with *y*, and returns zero otherwise. See Table 4 of the ANSI 754 standard for the meaning of *unordered*.

Reference: ANSI 754, MATH, REENT.

vfprintf( )

```
#include <stdarg.h>
#include <stdio.h>
int vfprintf(FILE *stream, const char *format, va_list arg);
```

This is equivalent to **fprintf( )**, but with the argument list replaced by *arg*, which must have been initialized with the **va_start** macro.

NOTE: By default in most environments, **vfprintf** buffers its output until a newline is output. To cause output character-by-character without waiting for a newline, call *setbuf( )*, p.520, with a NULL buffer pointer before after opening but before writing to the stream:

```
setbuf(*stream, 0);
```

OS calls: **isatty**, **sbrk**, **write**.

Reference: ANSI.

vfscanf( )

```
#include <stdarg.h>
#include <stdio.h>
int vfscanf(FILE *stream, const char *format, va_list arg);
```

This is equivalent to **fscanf( )**, but with the argument list replaced by *arg*, which must have been initialized with the **va_start** macro.

OS calls: **isatty**, **read**, **sbrk**, **write**.

Reference: DCC.

vprintf( )

```
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char *format, va_list arg);
```

This is equivalent to **printf( )**, but with the argument list replaced by *arg*, which must have been initialized with the **va_start** macro.

NOTE: By default in most environments, **vprintf** buffers its output until a newline is output. To cause output character-by-character without waiting for a newline, call *setbuf( )*, p.520, with a NULL buffer pointer before after opening but before writing to the stream:

```
setbuf(*stream, 0);
```


OS calls: **isatty**, **sbrk**, **write**.

Reference: ANSI.

vscanf( )

```
#include <stdarg.h>
#include <stdio.h>
int vscanf(const char *format, va_list arg);
```

This is equivalent to **scanf( )**, but with the argument list replaced by *arg*, which must have been initialized with the **va_start** macro.

OS calls: **isatty**, **read**, **sbrk**, **write**.

Reference: DCC.

vsprintf( )

```
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char *s, const char *format, va_list arg);
```

This is equivalent to **sprintf( )**, but with the argument list replaced by *arg*, which must have been initialized with the **va_start** macro.

OS calls: **isatty**, **sbrk**, **write**.

Reference: ANSI, REENT.

vsscanf( )

```
#include <stdarg.h>
#include <stdio.h>
int vsscanf(const char *s, const char *format, va_list arg);
```

This is equivalent to **sscanf( )**, but with the argument list replaced by *arg*, which must have been initialized with the **va_start** macro.

OS calls: **isatty**, **read**, **sbrk**, **write**.

Reference: DCC, REENT.

wcstombs( )

```
#include <stdlib.h>
size_t wcstombs(char *s, const wchar_t *wcs, size_t n);
```

Stores a multi-byte character string in the array whose first element has the address s by converting each of the characters in the string *wcs*. It converts as if calling **wctomb( )**. It stores no more than *n* characters, stopping after it stores a null character. It returns the number of characters stored, not counting the null character; unless there is an error, in which case it returns -1.

Reference: ANSI.

wctomb( )

```
#include <stdlib.h>
int wctomb(char *s, wchar_t wchar);
```

If s is not a null pointer, the function determines the number of bytes needed to represent the multi-byte character corresponding to the wide character *wchar*. It converts *wchar* to the corresponding multi-byte character and stores it in the array whose first element has the address *s*. It returns the number of bytes required, not counting the terminating null character; unless there is an error, in which case it returns -1.

Reference: ANSI.

write( )

```
#include <unistd.h>
int write(int fildes, const void *buf, unsigned nbyte);
```

Writes *nbyte* bytes from the buffer *buf* to the file *fildes*.

The C libraries provide an interface to this operating system call. Please see your OS manual for a complete definition.

Reference: POSIX, SYS.

y0( )

```
#include <math.h>
double y0(double x);
```

Returns the Bessel function of positive *x* of the second kind of order 0.

OS calls: **write**.

Reference: UNIX, MATH, REERR.

y0f( )

```
#include <mathf.h>
float y0f(float x);
```

Returns the Bessel function of positive *x* of the second kind of order 0. This is the single precision version of **y0( )**.

OS calls: **write**.

Reference: DCC, MATH, REERR.

y1( )

```
#include <math.h>
double y1(double x);
```

Returns the Bessel function of positive *x* of the second kind of order 1.

OS calls: **write**.

Reference: UNIX, MATH, REERR.

y1f( )

```
#include <mathf.h>
float y1f(float x);
```

Returns the Bessel function of positive *x* of the second kind of order 1. This is the single precision version of **y1( )**.

OS calls: **write**.

Reference: DCC, MATH, REERR.

yn( )

```
#include <math.h>
double yn(double n, double x);
```

Returns the Bessel function of positive *x* of the second kind of order *n*.

OS calls: **write**.

Reference: UNIX, MATH, REERR.

ynf( )

```
#include <mathf.h>
float ynf(float n, float x);
```

Returns the Bessel function of positive *x* of the second kind of order *n*. This is the single precision version of **yn( )**.

OS calls: **write**.

Reference: DCC, MATH, REERR.

PART VII

Appendices

A Configuration Files

A.1 Configuration Files

The compiler drivers and other tools are controlled by options from two sources: the command line, and standard *configuration files* installed automatically as part of the compiler suites.

Configuration files permit options to be constructed from string constants and variables using assignment, **if**, **switch**, **include**, and other statements.

For the most part, configuration files are used internally by the compiler suites to support multiple target processors. The current default target configuration is stored in the *version_path*/**conf/default.conf** configuration file (see *4.3 Alternatives for Selecting a Target Configuration*, p.25).

This appendix explains configuration file processing and the configuration language. It will be useful to those wishing to create configuration files, or to understand or modify the standard configuration files normally used by the tools.

A.2 How Commands, Environment Variables, and Configuration Files Relate

If a tool is executed with no options on the command line, no configuration file, and no environment variables set, then all options will have their default values as described here.

In practice, each tool is usually executed with some options on the command line, perhaps some options set with environment variables, and a number of site-dependent defaults set in configuration files, with remaining options having default values.

NOTE: Configuration files are used when the **dcc**, **dplus**, **das**, or **dld** programs are executed explicitly, e.g., from the command line or in a makefile. In this chapter, the term *tool* refers to any of these programs when executed explicitly.

When the **dcc** or **dplus** command automatically invoke the **das** or **dld** commands, configuration file processing is done for the **dcc** or **dplus** command and not again for the implicit **das** or **dld** command.

A.2.1 Configuration Variables and Precedence

Variables may be set in three places:

- In the operating system environment (see *2.3 Environment Variables*, p.15).
- On the command line using the **-WD** option for any variable, the **-WC** option for configuration variable **DCONFIG**, and the **-t** option to implicitly set configuration variables **DTARGET**, **DOBJECT**, **DFP**, and **DENVIRON**.
- 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 and 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.)

A.2.2 Startup

Here is how each tool processes the command line and configuration files at startup.

NOTE: Order is important. If a variable is given a value, or an option appears more than once, the final instance is taken unless noted otherwise.

1. The tool scans the command line for an **-@** option followed by the name of either an environment variable or a file, and replaces the option with the contents of the variable or file.
2. The tool scans the command line for each **-WD** *variable*=*value* option. If a variable matches an existing environment variable, the new value effectively replaces the existing value for the duration of the command (the operating system environment is not changed).

   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 override any **DCONFIG** environment variable.
3. The tool finds the main configuration file by checking first for a value of variable **DCONFIG**, and then if that is not set, looking in the standard location as given in Table A-1. The tool parses each statement in the configuration file as described in the following subsections.
4. After parsing the configuration file, the tool processes each of the input files on the command line using the options set by command-line and configuration-file processing.

Figure A-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 A-1 **Example of Command-Line and Configuration-File Processing**

Situation

An engineer works on Project 1 and normally uses *target1* with standard optimization (**-O** option). Now the engineer has a *target2* prototype and wants to use extended optimization (**-XO**).

Environment variables (set using operating system commands not shown)

Figure A-1 **Example of Command-Line and Configuration-File Processing** (cont'd)

`DFLAGS: -O`	As described in *2.3.1 Environment Variables Recognized by the Compiler*, p.15, **DFLAGS** is a convenient way to give options with an environment variable.
Command line `dcc -t`*target2* `-XO test1.c`	The command line is used to select the special processor *target2* and extended optimization.
Excerpts from configuration file dtools.conf `if (!$DTARGET) DTARGET=`*target1* `...`	If the target had not been set on the command line or elsewhere, it would default to *target1*.
`$DFLAGS` `$*`	**$DFLAGS** evaluates to **-O**. **$*** is a special variable evaluating to all of the command-line arguments. The **-XO** option from the command line overrides the related **-O** option from the **DFLAGS** environment variable.

A.3 Standard Configuration Files

Wind River recommends the use of three configuration files in a hierarchy. Standard versions of two of the files, **dtools.conf** and **default.conf** are shipped with the tools.

The tool identifies the main configuration file using the **DCONFIG** variable as described in *A.2.2 Startup*, p.547. If **DCONFIG** is not set, it then 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).

Table A-1 **Main Configuration File: Standard Name and Location**

System	Path and Name
UNIX	**/usr/lib/diab/***version***/conf/dtools.conf**
Windows 95, 98, and NT	**c:\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 A-2 on the next page **dtools** shows how the compiler combines the various environment variables and command-line options. **dtools** also serves as an example of how to write the configuration language.

Avoid altering **dtools**. Instead, set defaults and specific options by using the **-t** option on the command line to set **DTARGET**, **DOBJECT**, **DFP**, and **DENVIRON** (see *4.1 Selecting a Target*, p.21), or otherwise modifying **default.conf**, and/or by providing your own **user.conf**.

As shown in Figure 4-b, the **dtools** configuration file includes **default.conf** and then **user.conf** near the beginning. 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** statement names a non-existent file; therefore, both files are optional.

A.3.1 DENVIRON Configuration Variable

Configuration variable **DENVIRON** is set in **default.conf** and may be overridden by setting an environment variable of the same name or by providing a **-t***tof*:*environ* option on the command line executing **dcc**, **dplus**, **das**, or **dld**.

As shown in Figure A-2, if a file named **$DENVIRON.conf** exists in the **conf** subdirectory, it will be included by **dtools.conf**. The tools are delivered with several such "environment" **.conf** files. These are used to set options as required for several different target operating systems support by Wind River.

The **DENVIRON** configuration variable also controls the default search path use by the linker to find libraries. See the *environ* entry in the Table 4-1 and the section *4.2 Selected Startup Module and Libraries*, p.24 for details.

Figure A-2 **Standard dtools.conf Configuration File - Simplified Structure**

1. Variables and assignments used to customize selection and operation of the tools.

2. `include default.conf` — Read the second of the two configuration files included with the tools. This file records the target configuration in variables **DTARGET**, **DOBJECT**, and **DFP**, and **DENVIRON**, and is updated automatically during installation or by **dctrl -t**.

3. `include user.conf` — ASCII file to be created by the user to set, for example, default **-X** options and optimizations, additional default include files and libraries, default preprocessor macros, etc.

4. Switch and other statements using **DTARGET**, **DOBJECT**, and **DFP** to set options and flags, especially with respect to different targets. Also selection of tools if not customized above.

5. `include $DENVIRON.conf` — This optional file sets options for a specific target operating system. See *A.3.1 DENVIRON Configuration Variable*, p.549.

6. **dcc**, **dplus** section

   `$UFLAGS1` — Standard options to be used unless overridden by **$UFLAGS2**. To be set by the user in the **user.conf** configuration file.

   `$DFLAGS` — As described in *2.3.1 Environment Variables Recognized by the Compiler*, p.15, **$DFLAGS** is a convenient way to set an environment variable for widely used options. Because it follows **$UFLAGS1**, an option in **$DFLAGS** will override the same option in **$UFLAGS1**.

   `$*` — All arguments from the command line (**-t**, **-WD**, and **-WC** options are not re-processed). Options here will override the same options in both **$UFLAGS1** and **$DFLAGS**.

   `$UFLAGS2` — Overrides for **$UFLAGS1**, **$DFLAGS**, and the command line. To be set by the user in the **user.conf** configuration file.

7. **das** section

   ```
   $UAFLAGS1
   $*
   $UAFLAGS2
   ```

   $UAFLAGS1 and **$UAFLAGS2** can be set in **user.conf** to provide options for the assembler when it is executed explicitly, with **$UFLAGS1** options processed before command-line options and **$UFLAGS2** options processed after.

8. **dld** section

   ```
   $ULFLAGS1
   $*
   $ULFLAGS2
   ```

   And similarly, **$ULFLAGS1** and **$ULFLAGS2** can be set in **user.conf** to set options for the linker when it is executed explicitly.

A.3.2 UFLAGS1, UFLAGS2, DFLAGS Configuration Variables

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

- **UFLAGS1** and **UFLAGS2** are intended for compiler 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 A-2 above, **UFLAGS1** is expanded before command-line options and files, and **UFLAGS2** after command-line options.

  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. (Note: as
# assignment, quotes are required with embedded spaces.)
UFLAGS1="-Xlint -Xforce-prototypes"
```

NOTE: Variables are referenced with a "**$**", e.g., **$UFLAGS1** as shown in Figure A-2, but are written without a "**$**" when being set by an assignment statement.

  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.

  DFLAGS applies only to explicit execution of **dcc** and **dplus**, not to explicit execution of **das** or **dld**. However, some options are passed by **dcc** and **dplus** to the assembler or linker, e.g., the **-L** or **-Y P** options to specify a library search directory for the linker, or the **-Wa**,*arguments* **or -Wl**,*arguments* options to pass arguments to the assembler or linker. If **DFLAGS** includes such options, they will be passed along as usual.

- Options for a specific compilation are given on the command line. These override any options set with **UFLAGS1**, **DFLAGS**, but not **UFLAGS2** since **UFLAGS2** occurs after **$*** in **dtools.conf**.

NOTE: **UFLAGS1** and **UFLAGS2** (and **UAFLAGS1**, **UAFLAGS2**, **ULFLAGS1**, **ULFLAGS2**) cannot be overridden by environment variables of the same name. This is because they are reset to empty strings at the beginning of **dtools.conf** before being read from **user.conf**. This is in contrast to **DFLAGS** which is not so reset and can therefore be an environment variable.

A.3.3 UAFLAGS1, UAFLAGS2, ULFLAGS1, ULFLAGS2 Configuration Variables

Similar to the way **UFLAGS1** and **UFLAGS2** are intended to provide "permanent" options to be processed before and after command-line options for the compiler, **UAFLAGS1** and **UAFLAGS2** provide before-and-after options for the assembler and **ULFLAGS1** and **ULFLAGS2** provide before-and-after options for the linker.

As with **UFLAGS1** and **UFLAGS2**, it is expected that these options will be assigned values in a user-supplied **user.conf** configuration file, and because they are reset to the empty string at the beginning of **dtools.conf** they cannot be set as environment variables. See Figure A-2 for additional details.

A.4 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 greater 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.

A.4.1 Statements and Options

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 exception 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 readability. Blank lines are ignored.

A line may not 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 text which is not a statement or comment per the above is taken as options. In general, options have 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 allowed by a particular option, and either may include variables introduced by a "**$**" character (see *A.4.4 Variables*, p.554 below). Examples:

```
-O
-XO                     "O" is a name
-o test.out             "test.out" is a value
-Xlocal-data-area=0
-I$HOME/include         "$HOME" is a variable
```

A.4.2 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"
```

A.4.3 String Constants

A string constant is any sequence of characters ending in whitespace (spaces and tabs) or at end-of-line. To include whitespace in a string constant, enclose the entire constant in double quotes. Also, a string may include a variable prefixed with a "**$**" character.

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"
"$XFLAGS -Xanother-X-flag"                # $XFLAGS will be expanded
```

A.4.4 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 set a variable in a configuration file use an assignment statement. (See *A.4.5 Assignment Statement*, p.555).

To evaluate a variable, that is, to use its value, precede it with a "**$**" character. See *A.2.1 Configuration Variables and Precedence*, p.546 for a discussion of how *environment* variables and variables used in configuration files relate 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 **-WC** and **-WD** arguments have already been processed and are effectively ignored.) See examples below and also Figure A-2.

The special variable $-*x*, where *x* is 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 **$/** is replaced by the directory separation character on the host system: "**/**" on UNIX and "****" on Windows. (On any specific system, you can just use the appropriate character. Wind River uses "**$/**" for portability.)

Examples: assume that the environment variable **DFLAGS** is set to "**-XO**", and that the following command is given:

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

The following table shows examples of how variables are set given these assumptions.

Table A-2 **Variable Evaluation in Configuration Files**

Variable	Evaluates To	Comment (see assumptions above)
$DFLAGS	"-XO"	Environment variable.
$DFP	"soft"	Value is as if **-WD** set the **DFP** configuration variable (see *5.3.26 Define Configuration Variable (-W Dname=value)*, p.42).
$-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**.

A.4.5 Assignment Statement

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

variable **=** [*string-constant*]

As noted above, a *string-constant* may include a variable — see the last example.

Examples:

```
DLIBS=                              # Set to empty string.
XLIB=$HOME/lib                      # Variable XLIB is set.
YFLAGS="$XFLAGS -X12"               # Use "" for spaces in a string.
if (...) PF=-p GF=-g                # Two on one line (see if below).
$XFLAGS="$XFLAGS -Xanother-flag" # Inner $XFLAGS will be expanded.
```

A.4.6 Error Statement

The **error** statement terminates configuration file processing with an error. See the **switch** statement for an example.

A.4.7 Exit Statement

The **exit** statement stops configuration file processing. This is useful, for example, in an header file that specifies all compiler options, but does not want the compiler to continue the parsing in **default.conf** and **dtools.conf**.

A.4.8 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:

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*

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

```
if
else if
else if
...
else
```

Examples:

```
if (!$LIB) LIB=/usr/lib      # if LIB s not defined, set it
if ($OPT == yes) -O          # option -O if OPT is "yes"
else -g                      # else option  -g
```

A.4.9 Include Statement

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

include *file*

The contents of file *file* are parsed as if inserted in place of the **include** statement. The file must be located in the same directory as the main configuration file since no path is allowed in **include** statements. (See *A.3 Standard Configuration Files*, p.548.)

If the given file does not exist, the statement is ignored. Example:

```
include user.con
```

A.4.10 Print Statement

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

print *string*

Example:

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

A.4.11 Switch Statement

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

switch (*string***)**
case *pattern1***:**
  ...

  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** statement. 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 CHIP*:                    # any DTARGET beginning withCHIP
        ...
        break
    case *:                        # any other DTARGET
        print Error: DTARGET not set"
        error
endsw
```

B

Compatibility Modes: ANSI, PCC, and K&R C

NOTE: This section relates to C, not C++. Of the options listed in Table B-1, only **-Xdialect-strict-ansi** (equivalent to **-Xstrict-ansi**) affects the C++ compiler.

The Wind River compiler supports various standards, including full ANSI C89, partial ANSI C99, and full ANSI C++. Many existing C programs are coded in accordance with slightly varying standards. To ease porting of these programs, C modules can be compiled in four different modes as selected by an option from the following table:

Table B-1 **Compatibility Mode Options for C Programs**

Mode	Option	Meaning
C89	`-Xdialect-c89`	Conform to the ISO/IEC 9899:1990 standard for C.
C99	`-Xdialect-c99`	Conform to the ISO/IEC 9899:1999 standard for C. NOTE: Only a subset of this standard is supported.
ANSI	`-Xdialect-ansi`	Conform to ANSI X3.159-1989 with some additions as shown in the table below.
Strict ANSI	`-Xdialect-strict-ansi`	Conform strictly to the ANSI X3.159-1989 standard. Equivalent to **-Xstrict-ansi**.

Table B-1 **Compatibility Mode Options for C Programs** (cont'd)

Mode	Option	Meaning
K & R	`-Xdialect-k-and-r`	Conform to the pre-ANSI "standard" defined in *The C Programming Language* by Kernighan and Ritchie, with most ANSI extensions activated.
PCC	`-Xdialect-pcc`	Emulate 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 B-2 **Features of Compatibility Modes for C Programs**

Functionality	K&R	ANSI	Strict ANSI	PCC
long float is same as **double**.	y	n	n	y
The **long long** type is defined; but a warning (w) is generated when **long long** is used.	y	y	w	y
The **asm** keyword is defined.	y	y	n	y
The **volatile**, **const**, and **signed** keywords are defined.	y	y	y	n
"Double underscore" keywords (e.g. **__inline__** and **__attribute__**) are defined.	y	y	n	y
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. The compiler always accepts this and is either silent (s) or gives a warning (w).	s	s	s	w
A scalar type can be cast explicitly to a structure or union type, if the sizes of the types are the same. Such typecasts generate a warning (w).	w	w	n	w

Table B-2 **Features of Compatibility Modes for C Programs** (cont'd)

Functionality	K&R	ANSI	Strict ANSI	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 a bit-field is promoted to a larger integral type, sign is always preserved.	y	y	n	y
When prototypes are used and the arguments do not match an error (e) or warning (w) is generated.	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 an error (e) or warning (w) is generated.	e	e	e	w
When an array is declared with a zero dimension, generates a warning.	n	n	y	n
Incompletely braced structure and array initializers can either be parsed top-down (t) or bottom-up (b). May be controlled by the **-Xbottom-up-init** option (*5.4.14 Parse Initial Values Bottom-up (-Xbottom-up-init)*, p.61).	t	t	t	b
When pointers and integers are mismatched, generates an error (e) or a warning (w). May be controlled by the **-Xmismatch-warning** (*5.4.92 Warn On Type and Argument Mismatch (-Xmismatch-warning)*, p.94).	e	e	e	w
Trigraphs, e.g. "**??**" sequences, are recognized.	y	y	y	n

Table B-2 **Features of Compatibility Modes for C Programs** (cont'd)

Functionality	K&R	ANSI	Strict ANSI	PCC
Illegal structure references generate either an error (e) or a warning (w). If more than one defined structure contains a member, an error is always generated. Example: `int *p; p->m = 1;` **p** is both a pointer to an **int** and a pointer to a structure containing member **m**. This is likely an error.	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");` The "a" in the **printf** string will be replaced only for K&R and PCC.	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
Preprocessor recognizes vararg macros. (Not available with -Xpreprocessor-old option.)	y	y	y	n
__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
Spaces are legal before cpp **#**-directives.	n	y	y	n

Table B-2 **Features of Compatibility Modes for C Programs** (cont'd)

Functionality	K&R	ANSI	Strict ANSI	PCC
Parameters redeclared in the outer most level of a function will generate 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, in which a newly defined type is used to declare a parameter, is legal: `f(i) typedef int i4; i4 i; {}`	n	n	n	y

C
Compiler Limits

The C and C++ compiler limits usually relate to the size of internal data structures. Most internal data structures are dynamically allocated and are therefore limited only by total available virtual memory.

The following shows the minimum limits required by Section 2.2.4.1 of the ANSI X3.159-1989 C standard. The Wind River Compiler meets or exceeds these limits in all cases. When not limited by available memory (effectively unlimited), the C and C++ limit is shown in parentheses. “No limit” is shown in some cases for emphasis.

- 15 nesting levels of compound statements, iteration control, and selection control structures
- 8 nesting levels for **#include** directives (Wind River: 100)
- 8 nesting levels of conditional inclusion
- 12 pointer, array, and function declarators modifying a basic type in a declaration
- 127 expressions nested by parentheses
- 31 initial characters are significant in an internal identifier or a macro name (Wind River: no limit)
- 6 significant initial characters in an external identifier (Wind River: no limit)
- 511 external identifiers in one source file (Wind River: no limit)
- 127 identifiers with block scope in one block
- 1,024 macro identifiers simultaneously defined in one source file
- 31 parameters in one function definition and call

- 31 parameters in one macro definition and invocation
- 509 characters in a logical source line
- 509 characters in a string literal (after concatenation)
- 32,767 bytes in an object
- 255 case labels in a **switch** statement

The length of a symbol output by the compiler is limited to approximately 8,000 characters. In C++ projects with complex hierarchies, it is possible, though unlikely, that mangled names will run up against this limit, resulting in assembler errors, linker errors, or unexpected runtime behavior (when the wrong function or variable is accessed).

Memory is dynamically allocated as required, and is a function of:

- The size of the largest function in the source file. The size is measured in number of expression nodes, where each operand and operator generate one node in addition to several nodes per function. After code generation, the memory used by a function is 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.
- Large initialized arrays.

In addition, the number of KBytes the compiler is allowed to use to delay code generation in order to perform interprocedural optimizations is limited internally. The default value is 3000KB with **-O** and 6MB with **-XO**. It can be changed with option **-Xparse-size** (see *5.4.99 Specify Optimization Buffer Size (-Xparse-size)*, p.97).

The compiler does not generate correct debug information if there are more than 1023 included files.

D

Compiler Implementation Defined Behavior

D.1 Introduction

The ANSI C standard X3.159-1989 leaves certain aspects of a C implementation to the tools vendor. This appendix describes how Wind River has implemented these details. Note that there are differences between C and C++; this appendix addresses C only.

NOTE: 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.

D.2 Translation

Diagnostics

See *H. Messages*.

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.

Each character in the source character set is 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'
```

By default, wide characters are implemented as **long** integers (32 bits). See also *5.4.145 Define Type for wchar (-Xwchar=n)*, p.113.

Unless specified by the use of the **-Xchar-signed** or **-Xchar-unsigned** options (*5.4.20 Specify Sign of Plain Char (-Xchar-signed, -Xchar-unsigned)*, p.63), the treatment of plain **char** as a **signed char** or an **unsigned char** is as defined in Table 8-1.

Integers

Integers are represented in two's-complement binary form. The properties of the different integer types are defined in *8.1 Basic Data Types*, p.163.

Bitwise operations on signed integers treat both operands as if they were unsigned, but treat the result as signed.

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

Right shifting a negative integer divides it by the corresponding power of 2, with an odd integer rounded down. In the binary representation (on all supported processors), the sign bit is propagated to the right as bits are dropped from the right end of the number.

Floating Point

The floating point types use the IEEE 754-1985 floating point format on all supported processors. The properties of the different floating point types are defined in *8.1 Basic Data Types*, p.163.

The default rounding mode is “round to nearest”.

Arrays and Pointers

The maximum number of elements in an array is equal to (**UINT_MAX**–4)/**sizeof**(*element-type*). For **UINT_MAX**, see **limits.h**.

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** (this is sufficient to avoid overflow).

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 pages 163 to 167 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.

Header files are searched for in the order described for the **-I** command-line option (see *Set Header Files Directory (-I path)*, p.280). 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 *6.3 Pragmas*, p.123.

The preprocessor treats a pathname beginning with "/", "\", and a "driver letter" (*c***:**) as an absolute pathname. All other pathnames are taken as relative.

D.3 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[]**, which is a pointer to a null-terminated array of pointers to the parameters. **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[]) {...}
```

D.4 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` *file*`, line-`*number*

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

Table D-1 **ctype Functions**

Function	Decimal ASCII Value and Character
isalnum	65-90 (“A”-”Z”) 97-122 (“a”-”z”) 48-57 (“0”-”9”)
isalpha	65-90 (“A”-”Z”) 97-122 (“a”-”z”)
iscntr	l0-31
isdigit	48-57 (“0”-”9”)
isgraph	33-126
islower	97-122 (“a”-”z”)
isprint	32-126
ispunct	33-47 58-64 91-96 123-126
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 undervalue 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 need not 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.

The libraries support three buffering schemes: unbuffered streams, fully buffered streams, and line buffered streams. See function *setbuf( )*, p.520 and *setvbuf( )*, p.522 for details.

Zero-length files exist.

The rules for composing valid filenames 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 file 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 messages are generated by the **perror** and **strerror** functions may be found in file **errno.h** in the **sys** subdirectory of the **include** subdirectory (see Table 2-2 for the location of **include**).

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(s).

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 *15.11 Target Program Arguments, Environment Variables, and Predefined Files*, p.270. 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.

E

Assembler Coding Notes

This chapter describes the conventions used in the assembler to specify instruction mnemonics and addressing modes.

E.1 Instruction Mnemonics

The assembler supports all ARM and Thumb instructions as described in the *ARM Architecture Reference Manual*, including the simplified mnemonics described there.

The assembler also recognizes the following directives:

```
.code16
.code32
```

All code to follow these is either Thumb or ARM code respectively.

E.2 Operand Addressing Modes

E.2.1 Registers

This section specifies the valid names for registers. See *9.6 Register Use*, p.180 for details on register use.

Registers can be specified in the following ways, in either lower or upper case:

Table E-1 **Register Names**

Register	Use/Description
r0 - r15	General purpose registers; can only be used where a general purpose register is expected.
sp	Same as **r13**.
lr	Same as **r14**.
pc	Same as **r15**.

E.2.2 Expressions

See Chapter *19. Assembler Expressions*, for a complete description of valid expressions. 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.

F

Object and Executable File Format

F.1 Executable and Linking Format (ELF)

This section describes the Executable and Linking Format (ELF). The form *NAME*(*n*) means that the symbolic value *NAME* has the value shown in the parentheses.

F.1.1 Overall Structure

The ELF Object Format is used both for object files (**.o** extension) and executable files. Some of the information is only present in object files, some only in the executable files.

ELF files consist of the following parts. The ELF header must be in the beginning of the file; the other parts can come in any order (the ELF header gives offsets to the other parts).

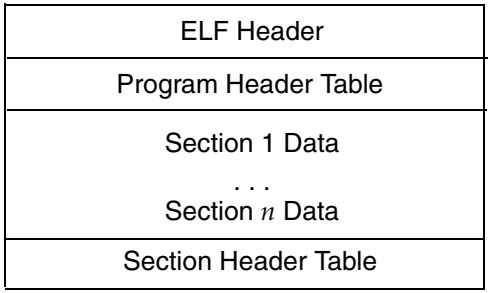
ELF header
: General information; always present.

Program header table
: Information about an executable file; usually only present in executables.

Section data
: The actual data for a section; some sections have special meaning, i.e. the symbol table and the string table.

Section headers
: Information about the different ELF sections; one for each section.

The following figure shows a typical ELF file structure:



F.1.2 ELF Header

The ELF header contains general information about the object file and has the following structure from the file **elf.h** (**Elf32_Half** is two bytes, the other types are four bytes):

```
#define EI_NIDENT 16

typedef struct {
    unsigned char  e_ident[EI_NIDENT];
    Elf32_Half     e_e_type;
    Elf32_Half     e_machine;
    Elf32_Word     e_version;
    Elf32_Addr     e_entry;
    Elf32_Off      e_phoff;
    Elf32_Off      e_shoff;
    Elf32_Word     e_flags;
    Elf32_Half     e_ehsize;
    Elf32_Half     e_phentsize;
    Elf32_Half     e_phnum;
    Elf32_Half     e_shentsize;
    Elf32_Half     e_shnum;
    Elf32_Half     e_shstrndx;
};
```

Table F-1 **ELF Header Fields**

Field	Description
e_ident	Sixteen byte long string with the following content: 4-byte file identification: "\x7FELF" 1-byte class: 1 for 32-bit objects 1-byte data encoding: little-endian: 1, big-endian: 2 1-byte version: 1 for current version 9-byte zero padding
e_type	The file type: relocatable: 1, executable: 2
e_machine	Target architecture: 40 ARM and Thumb
e_version	Object file version: set to 1.
e_entry	Programs entry address.
e_phoff	File offset to the Program Header Table.
e_shoff	File offset to the Section Header Table.
e_flags	Not used.
e_ehsize	Size of the ELF Header.
e_phentsize	Size of each entry in the Program Header Table.
e_phnum	Number of entries in the Program Header Table.
e_shentsize	Size of each entry in the Section Header Table.
e_shnum	Number of entries in the Section Header Table.
e_shstrndx	Section Header index of the entry containing the String Table for the section names.

F.1.3 Program Header

The program header is an array of structures, each describing a loadable segment of an executable file. The following structure from the file **elf.h** describes each entry:

```
typedef struct {
    Elf32_Word      p_type;
    Elf32_Off       p_offset;
    Elf32_Addr      p_vaddr;
    Elf32_Addr      p_paddr;
    Elf32_Word      p_filesz;
    Elf32_Word      p_memsz;
    Elf32_Word      p_flags;
    Elf32_Word      p_align;
} Elf32_Phdr;
```

ELF Program Header Fields

p_type
Type of the segment; only **PT_LOAD(1)** is used by the linker.

p_offset
File offset where the raw data of the segment resides.

p_vaddr
Address where the segment resides when it is loaded in memory.

p_paddr
Not used.

p_filesz
Size of the segment in the file; it may be zero.

p_memsz
Size of the segment in memory; it may be zero.

p_flags
Bit mask containing a combination of the following flags:

PF_X (1) Execute
PF_W (2) Write
PF_R (4) Read

p_align
Alignment of the segment in memory and in the file.

F.1.4 Section Headers

There is incitation header for each section in the ELF file, specified by the **e_shnum** field in the ELF Header. Section headers have the following structure from the file **elf.h**:

```
typedef struct {
    Elf32_Word      sh_name;
    Elf32_Word      sh_type;
    Elf32_Word      sh_flags;
    Elf32_Addr      sh_addr;
    Elf32_Off       sh_offset;
    Elf32_Word      sh_size;
    Elf32_Word      sh_link;
    Elf32_Word      sh_info;
    Elf32_Word      sh_addralign;
    Elf32_Word      sh_entsize;
} Elf32_Shdr;
```

Table F-2 **ELF Section Header Fields**

Field	Description	
sh_name	Specifies the name of the section; it is an index into the section header string table defined below.	
sh_type	Type of the section and one of the below:	
	SHT_NULL (0)	inactive header
	SHT_PROGBITS (1)	code or data defined by the program
	SHT_SYMTAB (2)	symbol table
	SHT_STRTAB (3)	string table
	SHT_RELA (4)	relocation entries
	SHT_NOBITS (8)	uninitialized data
	SHT_COMDAT (12)	like **SHT_PROGBITS** except that the linker removes duplicate **SHT_COMDAT** sections having the same name and removes unreferenced **SHT_COMDAT** sections (used in C++ template instantiation — see *Templates*, p.223).

Table F-2 **ELF Section Header Fields** (cont'd)

Field	Description	
sh_flags	Combination of the following flags:	
	SHF_WRITE (1)	contains writable data
	SHF_ALLOC (2)	contains allocated data
	SHF_EXECINSTR (4)	contains executable instructions
sh_addr	Address of the section if the section is to be loaded into memory.	
sh_offset	File offset to the raw data of the section; note that the **SHT_NOBITS** sections does not have any raw data since it will be initialized by the operating system.	
sh_size	Size of the section; an **SHT_NOBITS** section may have a non-zero size even though it does not occupy any space in the file.	
sh_link	Link to the index of another section header:	
	SHT_COMDAT	section with which this section should be combined
	SHT_RELA	the symbol table
	SHT_NOBITS	section with which this section should be combined
	SHT_PROGBITS	section with which this section should be combined
	SHT_SYMTAB	the string table
sh_info	Contains the following information:	
	SHT_RELA	the section to which the relocation applies
	SHT_SYMTAB	index of the first non-local symbol
sh_addralign	Alignment requirement of the section.	
sh_entsize	Size for each entry in sections that contains fixed-sized entries, such as symbol tables.	

The following table shows the correspondence between the *type-spec* as defined on 388 and the ELF section type and flags assigned to the output section.

Table F-3 **type-spec – ELF Section Type and Flags Correspondence**

Type-spec	Section Type (sh_type)	Section Flags (sh_flags)
BSS	**SHT_NOBITS**	**SHF_ALLOC | SHF_WRITE**
COMMENT	**SHT_PROGBITS**	(none)
CONST	**SHT_PROGBITS**	**SHF_ALLOC**
DATA	**SHT_PROGBITS**	**SHF_ALLOC | SHF_WRITE**
TEXT	**SHT_PROGBITS**	**SHF_ALLOC | SHF_EXECINSTR**

F.1.5 Special Sections

Following are the names of some typical sections and explains their contents:

.text
Machine instructions.

.rodata
Constant data and strings.

.data
Initialized data.

.bss
Uninitialized variables.

.comment
Comments from **#ident** directives in C.

.ctors
Code that is to be executed before the **main( )** function.

.dtors
Code that is to be executed when the program has finished execution.

.debug
Symbolic debug information using the DWARF format.

.line
Line number information for symbolic debugging.

.rela*name*
Relocation information for the section *name*.

.shstrtab
Section names.

.strtab
String Table for symbols in the Symbol Table.

.symtab
Contains the Symbol Table.

F.1.6 ELF Relocation Information

Relocation Information sections contain information about unresolved references. Since compilers and assemblers do not know at what absolute memory address a symbol will be allocated, and since they are unaware of definitions of symbols in other files, every reference to such a symbol will create a relocation entry. The relocation entry will point to the address where the reference is being made, and to the symbol table entry that contains the symbol that is referenced. The linker will use this information to fill in the correct address after it has allocated addresses to all symbols.

When an offset is added to a symbol in the assembly source

```
ldr    r1,=var+16
```

that offset is stored in the **r_addend** field, so that adding the real address of the symbol with the address field will yield a correct reference.

The relocation section does not normally exist in executable files.

A relocation entry has the following structure from the file **elf.h**:

```
typedef struct {
    Elf32_Addr      r_offset;
    Elf32_Word      r_info;
    Elf32_Sword     r_addend;
} Elf32_Rela;
```

ELF Relocation Entry Fields

r_offset
Relative address of the area within the current section to be patched with the correct address.

r_info >> 8
Upper 24 bits of **r_info** is an index into the symbol table pointing to the entry describing the symbol that is referenced at **r_offset**.

r_info & 255
Lower 8 bits is the relocation type that describes what addressing mode is used; it describes whether the mode is absolute or relative, and the size of the addressing mode. See the table below for a description of the various relocation types.

r_addend
A constant to be added to the symbol when computing the value to be stored in the relocatable field.

The relocation types for each supported target are documented in *version_path*/**include/elf**_*target*.**h**.

F.1.7 Line Number Information

The line number information section **.line** contains the mapping from source line numbers to machine instruction addresses used by symbolic debuggers. This information is only available if the **-g** option is specified to the compiler.

F.1.8 Symbol Table

The symbol table section **.symtab** is an array of entries containing information about the symbols referenced in the ELF file. A symbol table entry has the following structure from the file **elf.h**:

```
typedef struct {
    ELF32_Word      st_name;
    ELF32_Addr      st_value;
    ELF32_Word      st_size;
    unsigned char   st_info;
    unsigned char   st_other;
    Elf32_Half      st_shndx;
} Elf32_Sym;
```

ELF Symbol Table Fields

st_name
Index into the symbol string table which holds the name of the symbol.

st_value
Value of the symbol:

The alignment requirement of symbols whose section index is **SHN_COMMON**.

- The offset from the beginning of a section in relocatable files.
- The address of the symbol in executable files.

st_size
Size of an object.

st_info >> 4
Upper four bits define the binding of the symbol:

STB_LOCAL (0) symbol is local to the file
STB_GLOBAL (1) symbol is visible to all object files
STB_WEAK (2) symbol is global with lower precedence

st_info & 15
Lower four bits define the type of the symbol:

STT_NOTYPE (0) symbol has no type
STT_OBJECT (1) symbol is a data object (a variable)
STT_FUNC (2) symbol is a function
STT_SECTION (3) symbol is a section name
STT_FILE (4) symbol is the filename

st_other
Currently not used.

st_shndx
Index of the section where the symbol is defined. Special section numbers include:

SHN_UNDEF (0x0000) undefined section
SHN_ABS (0xfff1) absolute, non-relocatable symbol
SHN_COMMON (0xfff2) unallocated, external variable

F.1.9 String Table

The string table sections, **.strtab** and **.shstrtab**, contain the null terminated names of symbols in the symbol table and section names. Those symbols point into the string table through an offset. The first byte of the string table is always zero and after that all strings are stored sequentially.

G

Compiler -X Options Numeric List

The compiler **-X** options are listed in alphabetic order in *5.4 Compiler -X Options*, p.48 and following, with the internal numeric equivalent shown for each option.

However, when **-Xshow-configuration=1** is combined with **-S** or **-Xkeep-assembly-file** to create an assembly file, the **-X** options are shown in numeric form only.

This appendix lists compiler **-X** options that have numeric equivalents in numeric order.

Each option is shown in the form:

-X*n* -X*name* (*page number*)

H

Messages

H.1 Introduction

This appendix provides additional information on messages generated by the compilers and some of the other tools.

In analyzing messages, remember that a message can be generated for code which is apparently correct. Such a message is often the result of earlier errors. If a message persists after all other errors have been cleared, please report the circumstances to Customer Support.

H.2 Compiler Messages

H.2.1 Compiler Message Format

Compiler messages have the form:

```
"file", line #: severity-level (compiler:error #): message
```

Messages have one of four *severity-level* values as follows. The *severity level* for each message is shown in parentheses in the message description; for example, (w) for a warning message.

NOTE: The *severity level* of a message can be changed with the **-e** command-line option. See *5.3.8 Change Diagnostic Severity Level (-e)*, p.36. See also *option Pragma*, p.128 to put this or other options in, for example, a header file for use in other source files, or *A.3.2 UFLAGS1, UFLAGS2, DFLAGS Configuration Variables*, p.551 to specify options "permanently" in environment variables or configuration files.

Table H-1 **Compiler Message Severity Levels**

Severity Level	Type	Compilation Continues	Object File Produced	Notes
i	Information	Yes	Yes	Usually provides detailed information for an earlier message.
w	Warning	Yes	Yes	
e	Error	Yes	No	
f	Fatal	No	No	

In each message, "*compiler*" identifies the compiler reporting the error: **dcc** for the C compiler or **dplus** for the C++ compiler.

Example:

```
"err1.c", line 2: error (dcc:1525): identifier i not declared
```

H.2.2 Errors in asm Macros and asm Strings

Errors in assembly code embedded in C or C++ using **asm** macros or **asm** string statements are caught by the assembler, not by the compiler.

If the **-S** option is not used, the compiler will generate a temporary assembly file which is discarded after assembly. To preserve the assembly file for use in diagnosing errors reported in **asm** macros or **asm** strings, either:

- Use the **-Xkeep-assembly-file** and **-Xpass-source** command-line options to generate an annotated assembly file along with the object file.
- Use the **-S** option to stop after compilation, along with the **-Xpass-source** option, and then assemble the file explicitly using **das**.

H.2.3 C Compiler Message Detail

Numbered messages are issued by the compiler subprogram. Unnumbered messages are issued by the driver and are listed first.

NOTE: These messages are generated by **ctoa** (the C compiler) and **dtoa** (the older C++ compiler), not by **etoa** (the current C++ compiler). If you are compiling C++ code without the **-Xc++-old** option, a different set of C++ diagnostics is generated (see *H.2.4 C++ Messages*, p.649). When a message is shared by compilers, the same number is used for all instances.

(driver) **can't find program** *program_name*
program_name will be the name of some component of the compiler or other tool. (f)

Possible causes:

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

(driver) **can't fork**
The system cannot start a new process. (f)

(driver) **missing comma in -Y option**
The **-Y***c*,*dir* option must include a comma. (f)

(driver) **illegal output name** *file*
Specific output filenames given with the **-o** option are invalid to avoid common typing mistakes. (f)

```
dplus a.c -o b.c # b.c is an illegal output file name
```

(driver) **invalid option** *unknown*
The driver 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)

(driver) **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 Customer Support. (f)

1000: (general 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 Customer Support. (f)

1001: **illegal argument type**
The operand cannot be used with the operator. (e)

```
if ( i > pointer ) . . .
```

1003: function takes no arguments
Function was defined without arguments, but was called with arguments. (e)

```
int fun (){}
main(){
    fun(1);
}
```

1004: wrong number of arguments
Number of arguments given does not match prototype or function definition, (w) in C modules if **-Xpcc** or **-Xk-and-r** or **-Xmismatch-warning**, (e) otherwise.

```
int fn(int, int); ... fn(1,2,3);
```

1006: *string* in *string*
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 Customer Support. (f)

1007: ambiguous conversion -- cannot cast operand
The compiler cannot find an unambiguous way to convert an item from one type to another. (e)

1010: *Operator*, *type-designator*, **argument must be of pointer or integral type**
An operator that requires an integral or pointer type was applied to a different type.

```
float f;
f = ~f;
```

1012: *operator*, *type-designator*, **argument must be of pointer or arithmetic type**
The operator requires a pointer or arithmetic type operand. (e)

```
struct S {
    int i;
}s;
struct S *p;
*p -> i =3;     //
```

1013: left argument must be of integral type
The left operand must be an integral type. (e)

```
pointer | 3;
```

1015: *type-designator*, *operator*, *type-designator*, **left argument must be of arithmetic type**
The operand to the left of the operator must be of arithmetic type. (e)

```
pointer * 2;
pointer / 2;
```

1017: right argument must be of integral type
The right operand must be an integral type. (e)

```
7 | pointer;
```

1019: *type-designator*, *operator*, *type-designator*, **right argument must be of arithmetic type**
The operand on the right of the operator must be of arithmetic type. (e)

```
2 * pointer;
2 / pointer;
```

1025: division by zero
The compiler has detected a source expression that would result in a division by 0 during target execution. (w)

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

1028: *type-designator* [*type-designator*] **requires a pointer and an int**
A subscripted expression requires a pointer and an integer. (e)

```
main(){
    int x;
    x[3]=4;
}
```

1030: can't take address of main

Special rules for the function **main( )** are violated. (e)

```
int *p;
p = main;
```

1031: can't take address of a cast expression

The address operator requires an **lvalue** for its operand. (e)

```
int i, *p;
float f;
p = &(int)f;
```

1032: (anachronism) address of bound member function

The correct way to refer to the address of a member function is to use the "**::**" operator. The C method, using the dot "**.**" operator, causes the compiler to generate the "anachronism" warning. (w)

```
class C {
public:
    fun();
} c;

main(){
    class C * p;
    p= &c.fun; // Old way to reference a function
}
```

1033: can't take address of expression

Cannot use "**&**" or other means to find the address of the expression. (e)

```
int *pointer;
&pointer++;
```

1034: can't take address of bit-field expression

The address of bit-fields is not available. (e)

```
int *p;
struct {
    int i:3;
}s;
p = &s.i;
```

1041: 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;
```

1042: ?"*type-designator*":" *type-designator,* **bad argument type(s)**

Incompatible types have been used with the conditional operator. (e)

```
int i, *pointer, *p;
p = (2>1) ? i : pointer;
```

1043: trying to decrement object of type bool
A a **boolean** cannot be decremented. (e)

```
bool b;
b--;
```

1044: assignment to constant expression
A constant cannot be assigned a value after the constant is defined. (e)

```
const int i=5;
i=7;
```

1045: assignment to non-lvalue of type *type-designator*
The operand being assigned is not an **lvalue** type. (e)

```
const c = 5;
c = 7;
```

1046: assignment from *type-designator* to *type-designator*
An attempt has been made to assign a type to an incompatible type. (e)

```
int i, j;
i = &j;
```

1047: 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;
```

1050: bad left argument to operator *operator* **not a pointer**
The operator requires a pointer for its left operand. (e)

```
int int1, j;
int1 -> j=3;
```

1051: 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
```

1055: illegal function call
The function call is not valid. (e)

```
int i;
i();
```

1056: illegal function definition
A function definition is invalid. (e)

```
fun(iint i);
```

1057: main( ) may not be called from within a program
Calling **main( )** is not permitted. (e)

```
fun(){
    main();
}
```

1059: (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 Customer Support. (f)

1060: assignment operator "=" found where "==" expected
Encountered a conditional where the left hand side is assigned a constant value: (w)

```
if (i = 0) ... /* should possibly be i == 0) */
```

1061: illegal cast from *type-designator* to *type-designator*
An attempt is made to perform a cast to an invalid type, i.e., a structure or array type. (e)

```
struct a = (struct abc)x;
```

1063: 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)

1074: illegal cast
An attempt is made to perform a cast to an invalid type, i.e., a structure or array type. (e)

1075: friend declaration outside class/struct/union declaration
The keyword **friend** is used in a invalid context (e)

```
friend class foo {
    ...
};
```

1076: 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?
```

1077: typedefs cannot have external linkage
Linkage specification ignored for **typedef**, cannot have "**C**" or "**C++**" linkage. (w)

```
extern "C" typedef int foo;
```

1079: identifier *name* **previously declared** *linkage*
The identifier was already declared with another linkage specification. (e)

```
int foo;
extern "C" int foo;
```

1080: 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
}
```

1081: illegal storage class
External variables cannot be **automatic**. Parameters cannot be **automatic**, **static**, **external**, or **typedef**. (e)

```
int fn(i)
static int i; { ... }
```

1082: illegal storage class
A variable has been declared, but cannot legally be assigned to storage. (e)

```
register int r;           // Outside of any function
```

1083: only functions can be inline
The **inline** keyword was applied to a non-function, for example, a variable. (e)

1084: only non-static member functions can be virtual
For example, operators **new** and **delete** cannot be **virtual**.

```
virtual void *operator new( size_t size){...}
```

1086: redeclaration of *identifier*
It is invalid to redeclare a variable on the same block level. (e)

```
int a; double a;
```

1087: redeclaration of function
A function was already declared. May be caused by mis-typing the names of similar functions. (e)

1088: 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.

1089: illegal initializer
An initializer is not of the proper form for the object being initialized. Often caused by a type mismatch or a missing member in a structure constant. (e)

1090: static/external initializers must be constant expressions
Static initializations can only contain constant expressions. (e)

```
static int i = j+3;
```

1091: string too long
A string initializer is larger than the array it is initializing. (e)

```
char str[3] = "abcd";
```

1092: 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 };
```

1094: illegal type for identifier *identifier*
This can indicate a **template** was instantiated with the wrong arguments. (e)

```
template<class T>
class C{};

C<int, int> WrongArgs;
```

1096: 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)

1097: function-declaration in wrong context
A function may not be declared inside a **struct** or **union** declaration. (e)

```
struct { int f(); };
```

1098: only non-static member functions can be *string*
Only non-static member functions can be **const** or **volatile**.

```
class A {
    static foo() const;
};
```

1099: 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)

1100: member is incomplete
The structure member has an incomplete type, i.e., an empty array or undefined structure. (e)

```
struct { int ar[]; };
```

1101: anonymous union member 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)

1102: anonymous unions can't have member functions

1103: anonymous unions can't have protected or private members

1104: 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
}
```

1105: anonymous unions in file scope must be static
A special rule for an anonymous **unions** is violated. (e)

1106: friends can't be virtual
A **friend** is not a member of the class; it cannot be **virtual**. (e)

1107: conversion functions must be members of a class
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)

1108: member function declared as friend in its own class
Invalid declaration. (e)

```
class A {
    foo(int);
    friend A::foo(int);
}
```

1110: 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)

1111: 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)

1112: 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 { ... }; ... };
```

1113: 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;
};
```

1114: array is incompletely specified

An array cannot be declared with an incomplete type. (e)

```
int a[]; // No array size
```

1115: type ... is incomplete

Attempt to access a member in an incomplete type. (e)

1117: 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; { ... }
```

1120: constant expression expected

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

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

1121: integer constant expression expected

The size of an array must be computable at compile time. (e)

```
int ar[fn()];
```

1123: illegal type of switch-expr

A **switch** expression is of a non-integral type. (e)

1124: duplicate default labels

A **switch** has should not have more than one default label.

1125: int constant expected

A bit-field width must be an integer constant. (e)

1126: case expression should be integral constant

Case expressions must be integral constants. (e)

```
int i,j;
switch (i) {
  case j:
    i = 8;
}
```

1127: duplicate case constants

A **case** constant should not occur more than once in a **switch** statement. (e)

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

1127: duplicate case constants

Duplicate **case** constants were detected. (e)

```
main(){
    int year,j;
    switch (year) {
      case 2000:
        j = 8;
      case 2000:
        j = 9;
    }
}
```

1128: function must return a value

Found a **return** statement with no value in a function. (e)

```
int foo()
{
    return; // Must return a value.
}
```

1129: constructor and destructor may return no value

A constructor or destructor must not return a value. (e)

1130: 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) { ... }
```

1131: multiple initializations

A variable was initialized more than once. (e)

```
static int a = 4;
static int a = 5;
```

1133: extern objects can only be initialized in file scope

An **extern** object cannot be initialized inside a function. (e)

```
main(){
    extern int i=7;
}
```

1133: extern objects can only be initialized in file scope
Attempt to initialize an **extern** object in a function. (e)

```
foo()
{
    extern int one = 1;
}
```

1134: can't initialize arguments
It is not valid to attempt to initialize function parameters. (e)

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

1135: can't init typedefs
A **typedef** declaration cannot have an initializer. (e)

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

1136: initialization of automatic aggregates is an ANSI extension
When the compiler is run in PCC compatibility mode on a C module (**-Xpcc**), it will report initialization of automatic aggregate types. (w)

```
f() { int ar[3] = {1,2,3}; ... }
```

1140: too many parameters for operator ...

Overloaded operator declared with too many parameters. (e)

1141: too few parameters for operator ...
Overloaded operator declared with too few parameters. (e)

1142: 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
};
```

1143: operator->( ) must return class or reference to class

1144: operator ... can only be overloaded for classes
The operators "**,**" and "**=**" and the unary "**&**" can only be overloaded for classes. (e)

1145: operator . . . must be a non-static member function
The operators **( )**, **[]**, and **->** must be non-static member functions. These operators can only be defined for classes. (e)

1146: non-member operator function must take at least one argument of class or enum type or reference to class or enum type
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)

```
Date operator+(int i, j){...}
```

1147: constructors can't be declared *string*
Constructors cannot be declared **static** or **virtual**.

1148: constructors can't have a return type
A constructor declaration is invalid. (e)

1149: constructor is illformed, must have other parameters
A constructor declaration is invalid. (e)

1151: 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();
};
```

1152: destructors must have same name as the class/struct/union
The destructor declaration is invalid. (e)

1153: destructors may have no return type

```
const ~k(){}
```

1154: destructors can't be declared *string*
Destructors cannot be declared **static**.

1155: destructors may take no arguments
The destructor declaration is invalid. (e)

1156: conversion functions may take no arguments
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)

1157: 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)

1159: no type found for *identifier***, can be omitted for member functions only**
The identifier has not been declared. (e)

1160: class already has operator delete with *number of* **argument(s)**
The **delete** operator cannot be overloaded. (e)

1161: member operator functions can't be static
Operator functions in a class cannot be declared **static**. (e)

1162: member of abstract class
A class member cannot be of abstract type. (e)

1163: unions can't have virtual member functions
Union cannot have **virtual** functions as members. (e)

1164: 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)

1165: redeclaration of member *identifier*

A member occurs more than once in **a struct**, **union**, or **class**. (e)

```
struct { int m1; int m1; };
```

1166: member *name* **already declared**
Attempt to re-declare a member. (e)

```
class A {
    int a;
    int a; // Already declared
};
```

1167: static data member 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)

1168: a local class can't have static data members
Only non-**static** members can be used in a local class. (e)

1169: unions can't have static data members
Union cannot have **static** data members. (e)

1170: 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)

1171: illegal storage class for class member
A class member cannot be **auto**, **register**, or **extern**. (e)

1172: parameter has no identifier
When declaring a function, a name as well as a type, must be supplied for each parameter. (e)

```
int fn(int a, int) { ... }
```

1173: compiler out of sync: probably missing ";" or "}

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

1174: ellipsis not allowed as argument to overloaded operator
Cannot declare an overloaded operator with "**...**" as arguments. (e)

1175: 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)

1176: argument *n* **to string must be of type size_t**
For example, operator **delete's** second argument must be of type size_t

```
void operator delete(void *type, int x){
    free(type);
}
```

1177: *string* **must return void ***
For example the operator **new** must return a **void** pointer.

```
int *operator new(size_t size){...}
```

1179: *string* **takes one or two arguments**
For example, operator **delete** takes one or two arguments (e).

```
void operator delete(void *type, size_t size, int x){...}
```

1180: operator delete must have a first argument of type void *
The first argument of **delete** must be of type **void***.

```
void operator delete(int x){
    free(x);
}
```

1181: *string* **must return void**
For example, operator **delete** must return **void**.

```
int operator delete(void *type){...}
```

1182: 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);
```

1183: temporary inserted for non-const reference

The compiler made a temporary copy of a variable used in an assignment to a C++ reference. (w)

```
void getCount(unsigned int& count)
{
    count = 5;
    return;
}
...
signed int x = 100;
getCount(x);
```

In this example, the compiler makes a temporary copy of **x** and passes the copy (cast to **unsigned int**) to **getCount**. Hence it is the copy of **x**, and not **x** itself, that is modified by **getCount**; after the function executes, the value of **x** is still 100, not 5.

1184: temporary inserted for reference return

```
vint& constant1()
{
     return 1;
}
```

1186: const member *identifier* **must have initializer**

A constant member of a class must be initialized. (e)

```
class line{
    const int length;
...
};
```

1188: jump past initializer

An object cannot be accessed before it has been constructed.

```
class C
{
    public:
    int i;
    C(int ii) : i(ii) {}
};

void AllAlarmsOnOff (int function)
{
    switch ( function )
    {
         case 1:
             C c(0);
             break;
         default:
             c.i = 12;  // invalid access
             break;
    }
}
```

1190: this cannot be used outside a class member function body

1192: mismatching parenthesis, bracket or ? : around expression
Mostly likely, a parenthesis or bracket was left out of an expression, or the "**?**" and "**:**" in a conditional expression where interchanged. (e)

```
int i = (5 + 4]; // ] should have been a )
```

1193: missing operand for operator
An operand is missing. (e)

```
i & ;
```

1194: (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 notify Customer Support. (f)

1195: missing operand somewhere before
An operand was left out of an expression. (e)

1196: missing expression inside parenthesis
An expression was expected between the parentheses. (e)

```
i =() ;
```

1197: missing operand for operator ... inside parenthesis
An operand was left out of an expression. (e)

1198: too many operands inside parenthesis
An operator between the operands is missing. (e)

1199: missing expression inside brackets
An expression was expected between the brackets. (e)

```
int x[5];
int i = x[]; // x must be subscripted
```

1200: missing operand for operator ... inside brackets

1201: too many operands inside brackets

1202: missing operator before *string*
An operator is needed before *string*.

```
i = (2>1) 3: 4; // Conditional operator needs '?'
```

1205: operator ? without matching :
Operator "**?**" must be followed by a "**:**" . (e)

```
int i = 4 ? 5; // Missing : part
```

1207: syntax error near *token*
The parser has found an unexpected token. (e)

```
if (a == 1 ( /* missing ')' */
```

1208: expression expected
Could not find an expression where it was expected. (e)

```
if () { // The condition is missing.
    ...
}
```

1209: illegal expression
There was something wrong with the expression. Another error has probably already been reported. (e)

1210 to **1216: (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 notify Customer Support. (f)

For users searching online: 1211, 1212, 1213, 1214, 1215, 1216.

1219: (internal 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 Customer Support. (f)

1221: don't know size of object
The **sizeof** operator is used on an incompletely specified array or undefined structure, or an array of objects of unknown size is declared. (e)

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

1224: type must have default constructor
The class must have a default constructor. (e)

1227: EOF in comment
The source file ended in a comment. (w) if **-Xpcc**, (e) otherwise.

1228: 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 */
```

1229: EOF in character constant
The source file ended at an unexpected place during parsing. (f)

1230: newline in character constant

```
Vchar TAB = '\t;
```

1231: empty character constant

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

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

1232: too many characters in wide character constant

1234: newline in wide character constant

A newline is in a wide character constant.

Example: in the following, the wide character constant is intended to be **L'ab'**, but is broken across two lines.

```
int i = L'a
b';
```

1235: empty wide character constant

Empty wide character constants are not allowed:

```
int i = L'';
```

1236: EOF in string constant

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

1237: 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.
```

1238: 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;
```

1239: too long constant

A numeric constant is longer than 256 characters. (e)

1240: floating point value (...) out of range

A floating point constant exceeds the range of the representation format. (e)

```
double d = 1e10000;
```

1241: floating point overflow

Floating point overflow occurred during constant evaluation. (e)

```
float f=4E200;
```

1242: 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;    // '8' is invalid in an octal constant
```

1243: constant out of range

Constant overflows its type. (e)

```
int i = 4294967299; // Constant bigger than ULONG_MAX
```

1243: 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)

```
int j = 0xfffffffffff;
```

1244: constant out of range (*string***)**

An invalid constant was used. (w)

```
const int x=0xfffffffff;
if ((char)c==257) ...
```

1245: 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 only occur outside of a string constant, character constant, or comment. (e)

```
name$from$PLM = 1;
```

1246: no value associated with token

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 Customer Support. (f)

1247: syntax error after *string***, expecting** *string*

The expression is missing a semicolon or some token. (e)

```
int i
```

1248, 1249: label *identifier* **already exists**

A label can only refer to a single place in a function. (e)

1250: label *identifier* **not defined**

The label used in a **goto** statement is not defined. (e)

1251: label *identifier* **not used**

The label is never used. One possible cause is the misspelling of a label. This message appears if the **-Xlint** option is used. (w)

```
main(){
agian:                  // typo?
    goto again;
}
```

1252: typedef specifier may not be used in a function definition
Bad use of the **typedef** specifier. (e)

```
typedef int foo()
{
}
```

1253: 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
```

1254: redefinition of function
The function is already defined. (e)

```
int foo() {}
int foo() {}
```

1255: unions may not have base classes
Union cannot have base classes. (e)

1256: unions can't be base classes
Union cannot be used as base classes. (e)

1257: inconsistent exception specifications
Two function declarations specify different exceptions. (e)

```
int foo() throw (double);
int foo() throw (int);
```

1258: exception handling disabled
Exception handling has been turned off. Use **-Xexception=1** to enable it. (e)

1259: rtti disabled
RTTI (run-time type information) can be enabled or disabled through the **-Xrtti-...** option. See *5.4.114 Enable Run-time Type Information (-Xrtti, -Xrtti-off)*, p.103.

1260: 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;
```

1261: 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)

1264: main can't be overloaded

Special rules for the function **main( )** are violated. (e)

1265: can't distinguish *function_name1* from *function_name2*

Two overloaded functions cannot be distinguished from each other; they effectively have the same number and types of arguments in the same order. (e)

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

1266: 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);
```

1268: only virtual functions can be pure

Pure specifier found after non-**virtual** function. (e)

```
class foo {
    bar() = 0 // Must be virtual
};
```

1269: *Identifier* **is not a struct/class/union member**

The identifier is not a member of a structure, class, or union. (e)

```
int i;
i.j = 3; // j is not a member of a structure.
```

1272: 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)

1275: *error string*

This error number can indicate a number of different kinds of errors. In some cases, this message gives additional information about an error message displayed above this one. For example, if a function call is ambiguous, this error prints the names of candidate functions.

1276: 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)
{
    ...
}
```

1278: can't restrict access to *identifier*
An access declaration cannot 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)

1279: can't enable access to *identifier*

1281: no function matches call to *string*
The compiler did not find a match for a class method, or a **template** function. This can also indicate that a class does not have a default constructor. (e)

```
class line{
public:
    line(){}
};
line l(5,6);
```

Second example:

```
template< class T> T max(T a, T b) {
    return(a>b) ? a : b;

main(){
    int i;
    char c;
    max(i,c);
}
```

1282: can't resolve function call, possible candidates:
An overloaded function was called, but the function arguments did not match any prototype. (e)

```
fun(int i){}
fun(char c){}

main(){
    float f;
    fun(f);
}
```

1285: ambiguous reference to *identifier***, could be** *candidate1 candidate2 ...*
The identifier could not 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?
}
```

1288: 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)

1292: too many arguments for function style cast to *string*
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);
```

1293: non-type in new expression
A **new** expression requires a type.

```
class list {};
. . .
class list * cp;
cp = new lis;    // Spelled wrong
```

1294: type in new expression is abstract
The type in a **new** expression must not be abstract.

1295: 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];
```

1296: can't create void objects
The type in a **new** expression was void.

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

1297: type in new expression is incompletely specified

1298: object of abstract class
Attempt to declare an object of an abstract class. (e)

1298: can't construct object of abstract type
The type in a **new** expression is of abstract class. (e)

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

1299: 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);
```

1304: already volatile
A variable was declared **volatile** more than once. (w)

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

1305 to 1336: (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 Customer Support. (f)

For users searching online: 1305, 1306, 1307, 1308, 1309, 1310, 1311, 1312, 1313, 1314, 1315, 1316, 1317, 1318, 1319, 1320, 1321, 1322, 1323, 1324, 1325, 1326, 1327, 1328, 1329, 1330, 1331, 1332, 1333, 1334, 1335, and 1336.

1337: EOF in inline function body
The end of the source file was found while parsing an inline function. (f)

1338: arguments do not match template
The actual **template** argument types must match the declaration exactly. (e)

```
template<int size>
class foo {
    // ...
};

foo<7, 7> qux;
```

1339: arguments do not match template *template name*
The arguments do not match the **template**.

```
template<class T>
class C{};

C<int, int> WrongArgs;
```

1340: can't recover from earlier errors
Certain earlier errors have made it impossible for the parser to continue. (f)

1341: 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)

1344: syntax error - unexpected end of file
The parser has found an unexpected token. (e)

1347: identifier *name* **used as template** *name*
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 {
}
```

1354: "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
}
```

1355: 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];
```

1360: base class expected
Base class not found after "**:**" or "**,**" in a class definition. (e)

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

1361: can't initialize ... with a list
An object of a class which has constructors, bases, or non-public members cannot be initialized as an aggregate.

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

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

1362: can't nest function definitions
Functions cannot be defined inside other functions.

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

1367: 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 {};
```

1368: 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 {
    ~;
};
```

1370: 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  */
    ...
};
```

1371: conflicting declaration specifiers: *specifier1 specifier2*
Illegal mixing of **auto**, **static**, **register**, **extern**, **typedef** and/or **friend**. (e)

```
extern static int foo;
```

1372: conflicting type declarations
More than one type specified in a declaration. (e)

```
int double foo;
```

1373: enumerator may not have same name as its class
Only constructors and destructors for a class may have the same name as the class. (e)

1376: function *function name* **is not a member of class** *class name*
A function was not declared, it was misspelled, or the parameters were not used consistently. (e)

```
class line{
    lint(int l); // Misspelled
};
line::line(int l){}
```

1378: function *function name* **is not found**
A function call referred to a function that was not found. (e)

```
static int fun();
main(){
    fun();
}
```

1379, 1380: identifier ... declared as struct or class. Use struct or class specifier identifier ... declared as union. Use union 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
```

1381: identifier *name* **not a nested class nor a base class**

Something that is not a class was used as a base class. (e)

1383: identifier *identifier* **is not a type**

What appeared to be a declaration began with an identifier that is not the name of a type.

```
INT I;
```

1384: identifier *name* **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
}
```

1385: identifier *identifier* **not a static member of class** *class name*

Invalid declaration. (e)

```
struct A {
    int i;
};

int A::i;
```

1386: identifier *identifier* **not declared in** *string*

An identifier is used but not declared. Check the *identifier* for spelling errors. (e)

1388: identifier *identifier* **not declared**

An identifier was used without being declared. (e)

1391: identifier *name* **is not a class**

An identifier that is not a class was used before "**::**".

1394: illegal *expression*

A **break** statement is only allowed inside a **for**, **while**, **do** or **switch** statement. (e)

A **continue** is only allowed inside a **for**, **while** or **do** statement. (e)

A **default** or **case** label is only allowed inside a **switch** statement. (e)

1395: illegal function specifier for argument

A parameter cannot be declared **inline** or **virtual**.

```
void foo(inline int);
```

1397: 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 {
    ...
};
```

1403: main can't be declared string

Special rules for the function **main( )** are violated. (e)

1404: 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
}
```

1405: missing argument declaration

Argument declaration omitted. (e)

```
class bar {
    foo(, int);
};
```

1410: no default arguments for overloaded operators

Overloaded operators cannot have default arguments. (e)

1411: no redefinition of default arguments

An argument can be given a default value only once in a set of overloaded functions. (e)

```
void foo(int = 17);
void foo(int = 4711);
```

1412: 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
}
```

1414: non-extern object *name* **of type** *type-name* **must be initialized**
A **const** object must be initialized unless it is **extern**.

1415: 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)

```
const struct S &structure;
```

1417: only functions can have pascal calling conventions

```
int pascal i;
```

1418: only static constant member of integral type may have initializer
A member that is a static integral type can be initialized; others cannot. (e)

```
struct {
    const int *p =0x3333;
}s;
```

1419: operator ... cannot be overloaded
It is invalid to overload any of the operators "**.**" or "**.***" or "**? :**" .

1420: parenthesized expression-list expected after type *typename*

1423: redeclaration of symbol ...
A symbol in an enumerated type clashes with an earlier declaration. (e)

1427: 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?
}
```

1428: static member ... can't be initialized
A **static** class member cannot be initialized in a member initializer. (e)

```
class A {
    static int si;
    A(int ii) : si(ii) {}
};
```

1429: string literal expected in asm definition
String missing in an **asm** statement.

```
asm();    // the parentheses should contain an instruction
```

1430: subsequent argument without default argument
Only the trailing parameters may have default arguments. (e)

```
void foo(int = 4711, double);
```

1431: syntax error - catch handler expected after try
The parser has found an unexpected token. (e)

1432: syntax error - catch without matching try
The parser has found an unexpected token. (e)

1433: syntax error - class key seen after type. Missing ;?
The parser has found an unexpected token. (e)

1434: syntax error - class name expected after ::
The parser has found an unexpected token. (e)

1435: syntax error - colon expected after access specifier
The parser has found an unexpected token. (e)

1436: syntax error - declarator expected after ...
The parser has found an unexpected token. (e)

1437: syntax error - declarator expected after type
The parser has found an unexpected token. (e)

1438: syntax error - declarator or semicolon expected after class definition
The parser has found an unexpected token. (e)

1439: syntax error - else without matching if
The parser has found an unexpected token. (e)

1441: syntax error - identifier expected after ...
The parser has found an unexpected token. (e)

1442: syntax error - initializer expected after =
The parser has found an unexpected token. (e)

1444: syntax error - keyword operator must be followed by an operator or a type specifier
The parser has found an unexpected token. (e)

1446: syntax error - type tag expected after keyword enum
The parser has found an unexpected token. (e)

1454: type defined in return type (forgotten ";"?)
It is illegal to define a type in the function return type. (e)

```
struct foo {} bar()
{
}
```

1455: type definition in bad context
A type was defined where it was not allowed. (e)

1456: type definition in condition
Types cannot be defined in conditions. (e)

```
if (struct foo { int i } bar) {
    // ...
}
```

1457: type definition not allowed in argument list
Types cannot be defined in argument lists. (e)

```
int foo( struct bar int a; } barptr);
```

1460: type expected after new
A **new** expression requires a type. (e)

```
p = new;
```

1461: type expected for ...
No type found in declaration of a variable. (e)

1462: type expected in template parameter
This could indicate a misspelling of a **template** parameter. (e)

```
template<classT> ...;
```

1463: type expected in arg-declaration-clause
An argument type is missing in a function declaration. (e)

```
class bar {
    foo(imt);
};
```

1464: type expected in cast
Found something that was not a type in a cast expression. (e)

1465: type expected
Found an expression that was not a type where a type was expected. (e)

1466: type in new expression can't be *string*
A type in a **new** expression cannot be **pascal** or **asm**.

1467: type in new expression may not contain class/struct/enum declarations
Cannot 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 };
```

1469: unknown language string in linkage specifier: ...
Only "**C**" and "**C++**" allowed in linkage specifiers. (e)

```
extern "F77 { // Don't know anything about F77 linkage}
```

1477: already const
A variable was declared **const** more than once. (w)

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

1479: comma at end of enumerator list ignored
A superfluous comma at the end of a list of enumerators was ignored. (w)

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

1480: enumerators can't have external linkage
extern cannot be specified for **enum** declarations. (e)

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

1481: function *function-name* **not declared**
If the **-Xforce-declarations** option is used, the compiler will generate this error message when a function is used before it has been declared. (w)

1484: missing declarator in typedef
No declarator was given in a **typedef** statement. (e)

```
typedef class foo {
    // ...
};
```

1485: 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
{
    ...
}
```

1486: initializer that is a brace-enclosed list may contain only constant expressions
A variable was initialized using a brace-enclosed list containing an expression (such as a variable) that cannot be evaluated during compilation.

```
int i = 12;
...
int x[] = { 1, 2, 3 , i };
```

This is allowed in C++ but not in C.

1488: 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; ... }
```

1489: redundant semicolon ignored
Found an extra semicolon among the members of a function. (w)

```
class A {
    int a;
    ;
};
```

1492: virtual specified both before and after access specifier
Syntax error. (w)

```
class A {};
class B : virtual public virtual A {};
```

1493: redeclaration of ...
A function has been redeclared to something else. (e)

```
int i(int);
double i(int);
double i( int i) {...}
```

1494: non-extern object identifier of type *type-designator* **must be initialized**
This message may indicate that a **const** member of a class/structure/union was not initialized. (e)

```
class C {
    const int ci;
} c;
```

1495: non-extern const object *name* **must be initialized**
A **const** object must be initialized unless it is **extern**.

```
const char c;
```

1497: too many declaration levels
An internal stack overflowed. This is unlikely to happen in the absence of other errors. (f)

1498: internal table-overflow
Internal stack overflowed. May occur with extremely complex, deeply nested code. To work-around, simplify or modularize the code. If the problem does not appear to be a consequence of some earlier error, please report it to Customer Support. (f)

1500: function <*function_name*> **has no prototype**
The function *function_name* was used without a preceding prototype declaration. In C,

```
void f();
```

is a declaration but not a prototype declaration—it declares *f* to be a function but says nothing about the number or type of arguments it takes. This warning

is returned when an attempt has been made to use *f* without making a prototype declaration of it first.

This warning is returned only when the command line option **-Xforce-prototypes** is used. (w)

1501: function-pointer has no prototype

A function pointer was used but was declared to have a type that lacks a prototype. In C,

```
void (*f)();
```

declares *f* to be a function pointer but says nothing about the number or type of arguments it takes. This warning is returned when an attempt has been made to use *f* without making a prototype declaration of it first.

This warning is returned only when the command line option **-Xforce-prototypes** is used. (w)

1504: arglist in declaration

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

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

1507: end of memory

Ran out of virtual memory during compilation. The compiler first attempts to skip some optimizations in order to use less memory, however this error can occur for large functions on machines with limited memory. Note: initialized arrays require the compiler to hold all initial data and can contribute to this error. If the problem does not appear to be a consequence of some earlier error, please report it to Customer Support. (f)

1509: expression involving packed member too complicated

This indicates that the processor does not support “compound assignment” for volatile members of packed structures.

```
struct1.a |=3; // May have to use struct1.a = struct1.a|3
```

1511: can’t access short or int bit-fields in packed structures unless the architecture supports atomic unaligned accesses (-Xmin-align=1)

Packed structures cannot contain bit-fields unless the architecture support atomic unaligned access. To see if the architecture supports atomic unaligned access, compile a file with the **-S** option and then examine the **.s** assembly file. Look for the **-X93** option in the header. If **X93**=1, the architecture supports atomic unaligned access. (e)

```
#pragma pack(1)
struct S {
    int j:3;
};
```

1513: byte swapped structures can't contain bit-field

Bit-fields are not allowed in byte-swapped structures. (e)

```
#pragma pack (,,1)  // Byte swap
struct s {
    int j:3;
}
```

1515: 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)

1516: parameter *parameter name* **is never used**

A parameter to a function is not used. This message appears if the **-Xlint** option is used. (w)

```
fun(int i){};
```

1517: function *function name* **is never used**

A function was declared but not used. This message appears if the **-Xlint** option is used. In the example, the file consists of one line. (w)

```
static fun();
```

1518: variable *identifier* **is never used**

A variable is never used. This message appears if the **-Xlint** option is used. (w)

```
fun(){
    int i;
}
```

1519: 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.

1520: large structure is used as argument

The size of a structure passed as an argument to a function equals or exceeds the size specified by **-Xstruct-arg-warning**. (This message is returned only when the command-line option **-Xstruct-arg-warning** is used.) (w)

1521: missing return expression
A function is defined with a return type, but does not return a value. This message appears if the **-Xlint** option is used. (w)

```
float fun(){
    return;
}
```

1522: statement not reached
A statement can never be executed. This message appears if the **-Xlint** option is used. (w)

```
main(){
    int never;
    return 0;
    never=6;
}
```

1523: can't recognize storage mode *unknown*
The storage mode specified in an **asm** macro is unknown. See *7. Embedding Assembly Code* for more details. (e)

1524: too many enhanced asm parameters
There can be a maximum of 20 parameters and labels used in an **asm** macro. See *7. Embedding Assembly Code* for details.

1525: identifier *identifier* **not declared**
An identifier was not declared. (e)

```
fun(){
return i;
}
```

1526: asm macro line too long
A very long line was given in an **asm** macro. See *7. Embedding Assembly Code* for more details. (e)

1527: non-portable mix of old and new function declarations
A function declaration was made in accordance to an older C standard. In K & R C, **char**s and **short**s are promoted to **int**, and **float**s are promoted to **double** just before a call is made to a function. However, in ANSI C, the arguments match the prototype at the call site. (w)

1528: can't initialize variable of type *type_designator*
Some types do not allow initialization. (e)

```
void a = 1;
```

1534: only first array size may be omitted
The size of the first dimension of an array can be omitted; all others must be specified. (e)

```
int x[3][];
```

1535: illegal width of bit-field
A bit-field width is greater than the underlying type used for the bit-field. (e)

Example for a target with 32 bit integers:

```
struct { int i:33; }
```

1536: bit-field must be int or unsigned
The compiler detected an unsupported bit-field type. (e)

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

1541: redeclaration of struct/union/enum ...
A **struct**, **union**, or **enum** tag name was used more than once: (e)

```
struct t1 { ... }; struct t1 { ... };
```

1542: redeclaration of member *variable name*
A member has been declared more than once. (e)

```
struct{
    int i;
    int i;
};
```

1543: negative subscript not allowed
The size of an array cannot be negative. (e)

```
int ar[-10];
```

1544: zero subscript not allowed
An array of zero size cannot be declared when compiling for strict ANSI C (**-X7=2**, or **-Xdialect-strict-ansi**). (w)

```
int x[0];
```

1546: dangerous to take address of member of packed or swapped structure
Using the address of a packed or byte-swapped structure is not recommended. (w)

```
#pragma pack (2,2,1)
. . .
ptr = &(struct1.i);
```

1547: 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);
```

1548: can't do sizeof on bit-field
The **sizeof** function does not work on bit-fields. (e)

```
struct {
    int j:3;
} struct1;
i = sizeof(struct1.j);
```

1549: illegal value
Only certain expressions can be on the left hand side of an assignment. (e)

```
a+b = 1;
(a ? b : c) = 2;    /* not valid in C modules*/
```

1550: 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);
```

1551: argument [*identifier***] 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);
```

1552: initializer type "*type*" **incompatible with object type** "*type*"
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;
```

1553: too many errors, good bye
The compiler has found so many errors that it does not seem worthwhile to continue. (f)

1554: 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. This message may also indicate an attempt has been made to find the sum of two pointers.

```
int *pi, **ppi; ... if (pi == ppi) ...
#illegal types: ptr-to-int '==' ptr-to-ptr-to-int

int *p, *q;
p = p + q;   // Attempt to add pointers
#illegal types: ptr-to-int '+' ptr-to-ptr-to-int
```

1555: 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.

1556: volatile packed member cannot be accessed atomically
For the selected processor, a packed member cannot be accessed atomically if it is **volatile**. (w)

```
#pragma pack(1, 1)

struct {
    volatile int v;
} s;
s.v =3;              /* generates error 1556 */
```

1560: unknown pragma
The **pragma** is not recognized. (w)

```
#pragma tist
```

1561: unknown option -X*unknown*
The compiler was started with an **-X** option that is not recognized. (w)

1562: bad #pragma use_section: section *section name* **not defined**
A **#pragma use_section** command has not been correctly given. (w)

```
#pragma section DATA3  // Correct
#pragma use_section x  // Omitted section class name DATA3
```

1563: 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)

1564: bad #pragma pack
The **#pragma pack** statement is not correct. (w)

```
#pragma pack(1,2,3,4) // Takes up to three arguments
```

1565: illegal constant in #pragma pack
An invalid constant has been used in a **pack pragma**. (w)

```
#pragma pack(7) // Must use powers of 2 for alignment
```

1566 to 1572: obsolete messages
Messages numbered 1566 to 1572 should not appear because they refer to obsolete features.

1573: user's error string
Error number 1573 can be used to display any string the user chooses when

- the compiler compiles this file, by use of **#pragma error** *string*:

  ```
  #pragma error Now compiling test.c; // compilation continues
  ```

- the compiler stops because of an error, by use of **error** *string*:

```
#error        // This terminates the compilation process
```

1574: can't open *file* for input
The given *file* cannot be opened. (f)

1575: can't open *file* for output
The given *file* cannot be opened. (f)

1577: can't open profiling file *file*
The file given with the **-Xfeedback=***file* option cannot be opened. (w)

1578: profile file is of wrong version (*file*)
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)

1579: profile file *file* 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)

1580: 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.

1584: illegal declaration-attribute
A declaration contains an invalid combination of declaration specifiers. (w)

```
unsigned double foo;
```

1585: global register *register name* is already used
The global register has already been reserved. (w)

```
#pragma global_register counter = r14
#pragma global_register kounter = r14
```

1586: cannot use scratch registers for global register variables
Scratch registers cannot be used for global register variables. (w)

```
#pragma global_register counter=scratch-register-name
```

1587: global register *register-name* is invalid
Found an unrecognized register name in a **global_register pragma**. (w)

1588: no .cd file specified!
The target description (**.cd**) file was not specified.

The compiler reads a *target description file* during initialization (see "Targets," Table 2-2). Normally, when the **dcc** command is given, the **.cd** file is

automatically specified. To find out the **.cd** filename for your selected target configuration, run **dcc** with the **-#** option to display all of the commands generated, and look at the **-M** option for the **ctoa** program. (f)

Likely causes:

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

1589: can't open .cd file!
See error 1588 for a description of the **.cd** file and likely causes.

1590: .cd file is of wrong type!
See error 1588 for a description of the **.cd** file and likely causes.

1591: .cd file is of wrong version!
See error 1588 for a description of the **.cd** file and likely causes.

1592: cd file *file* **too small?!**
See error 1588 for a description of the **.cd** file and likely causes.

1593: rite error
Write to output file failed. (f)

1595: illegal arg to *function name*
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 Customer Support. (f)

1596: test version of compiler: File is too big!
This error is generated when certain limits in an evaluation copy of the compiler are exceeded. (f)

1597: test version of compiler: Can't continue!
This error is generated when certain limits in an evaluation copy of the compiler are exceeded. (f)

1598: no matching asm pattern exists
While scanning an **asm** macro, no storage-mode-line matching the given parameters was found. See *7. Embedding Assembly Code* for details.

1599: 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)

1600: no table entry found!
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 Customer Support. (f)

1601: address taken in initializer (PIC)
Position-independent code. 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.

1602: variable ... is incomplete
A variable is defined with a type that is incomplete. (e)

```
struct a;
struct a b;
```

1603: 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 Customer Support. (f)

1604: useless assignment to variable *identifier***. Assigned value not used**
The variable assignment has no effect, since the assigned value is not used. This message appears if the **-Xlint** option is used. (w)

```
fun(){
    int i=1;
}
```

1605: 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. The compiler continues, but produces less than optimal code. (w)

1606: conditional expression or part of it is always true/false
A conditional test is made, but the results will always be the same. This message appears if the **-Xlint** option is used. (w)

```
int main(){
    int i = 3;
    if (i < 6)
        return 4;
}
```

1607: variable *name* **is used before set**
During optimization, the compiler discovers a variable that is used before it is set. (w)

```
func() { int a; if (a == 0) ... }
```

1608: variable *identifier* **might be used before set**
A variable may have been used before it was given a value. (w)

```
fun(){
    int i,j;
    i = j;    // j is used before set
}
```

1609: 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)

1611: argument list not terminated
The end of the source file was found in a macro argument list. (w) if **-Xpcc**, (e) otherwise.

1612: 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 it is frequently caused by an unclosed comment or unclosed string. (w) if **-Xpcc**, (e) otherwise.

1617: syntax error in #if
The expression in an **#if** directive is incorrect, (w) if **-Xpcc**, (e) otherwise.

```
#if a *
```

1618: 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.

1619: include nesting too deep
The preprocessor cannot nest header files deeper than 100 levels, (w) if **-Xpcc**, (e) otherwise.

1621: can't find header file *unknown*
The preprocessor cannot find a file named in an **#include** directive. (w) if **-Xpcc**, (e) otherwise.

1622: found #elif, #else, or endif without #if
Found an **#elif**, **#else**, or **#endif** directive without a matching **#if** or **#ifdef**. (w) if **-Xpcc**, (e) otherwise.

1623: bad **include syntax**
The **#include** directive is not followed by **<** or **"** or the filename is too long. (w) if **-Xpcc**, (e) otherwise.

1624, 1625: illegal macro name
illegal macro definition

Macro names and arguments must start with a letter or underscore, (w) if **-Xpcc**, (e) otherwise.

1626: illegal redefinition of *macro_name*

__LINE__, __FILE__, __DATE__, __TIME__, defined, and **__STDC__** cannot be redefined, (w) if **-Xpcc**, (e) otherwise.

1627: macro *macro name* **redefined**

The macro was previously defined. (w)

```
#define PI 3.14
#define PI 3.1416
```

1629: undefined control

Undefined or unsupported directive found after **#**, (w) if **-Xpcc, (e)** otherwise.

```
#pragmo
```

1630: illegal assert name

An **#assert** name must be an identifier and must be preceded by a "**#**" character, (w) if **-Xpcc**, (e) otherwise.

1631: 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);
```

1632: recursive macro *macro name*

A recursive macro has been detected. The error occurs when the macro substitution occurs, line 4 in this case: (e)

```
#define max(A,B) A>B ? A : max(A,B)
main(){
    int i=1,j=2,k;
    k = max(i,j); // Reports error for this line.
}
```

1633: parse error

The complier was not able to parse the expression. (e)

```
x = multiply(y, ); // Comma, but no second argument
main(}             // Typed } instead of )
```

1635: license error: *error message*

An error occurred when checking the license for the software tools. The error message describes the problem (no server for this feature, etc.). Please refer to your *Getting Started* manual or contact Customer Support. (f)

1638: illegal error level *error level* **in option** *option name*
The **-e***xn* option was used with an invalid error level. The **-e** option is used for increasing the severity of error messages for a particular error. (w)

```
dcc -e99 test.c // 99 is invalid error level
```

1640: illegal error message number *message number*
The **-e***xn* option was used with an invalid error message number. The **-e** option is used for increasing the severity of error messages for a particular error. (w)

```
dcc -ew10000 test.c // There is no message number 10000
```

1641: cannot reduce severity of error message *number* **below** *error level*

```
% dcc -ew1614 test.c
warning (dcc:1641): Cannot reduce severity
of message 1641 below "error"
```

1643: narrowing or signed-to-unsigned type conversion found: *type* to *type*
A type conversion from signed to unsigned, or a narrowing type conversion has been found. This message appears if the **-Xlint** option is used. (w)

```
main(){
    int i;
    char c;
    c = i;
}
```

1647: non-*string* **method invocation** *expression* **on** *string* **object** *expression*
This error indicates a mismatch between an invocation and the declaration of a method.

For example, non-**const** method invocation in **const** object. Methods of **const** objects must be **const**.

```
class C {
    int i;
public:
    f() { i = 12; }
    C() {}
};

const C c;

main() {
    c.f();
}

"x.cpp", line 11: error (1647): non-const method
invocation f() on const object c
```

1650: no profiling information found in database *database name*
This applies to programs compiled and run in the RTA (Run-time Analysis tools). (w)

A program was compiled with the option **-Xprof-feedback**=*database directory,* and the profiling information was not found in the database directory. The normal sequence of events is:

a. A program is compiled with an **-Xprof-***type* option that adds profiling code to the program.

b. The program is run and profiling information is collected using the RTA.

c. The program is compiled with the **-Xprof-feedback** option, and the compiler uses the profiling information to optimize the code.

Possible causes of the error:

- The wrong database directory was specified.
- The database does not contain profiling data.

1651: can't find profiling information for *function* **in database**
A program was compiled with the option **-Xprof-feedback**=*database directory,* and the profiling information was not found for the function. See error 1650, above, for a brief explanation of the situations where this error occurs. (w)

Possible causes of the error:

- The module was not compiled with an **-Xprof-***type* option that would add code for instrumentation.
- The program was not run, and so profiling data was not collected.

1657: initializer *method name* **initializes neither a direct base nor a member**
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
};
```

1663: inline of *function* **does not occur in routine function - try increasing value of -Xinline**
This warning is generated whenever the **inline** keyword is specified but the compiler does not inline the function. Increasing the value of **-Xinline** or **-Xparse-size** can help, but there are other reasons for not inlining a function.

1665: long long bit-fields are not supported
long long cannot be used with bit-fields. (w)

```
struct {
    long long story:3;
}
```

1671: non-portable behavior: operands of *type* **are promoted to unsigned type only in non-ANSI mode**

When a non-ANSI compilation mode is used, for example, **-Xpcc**, this warning appears when the compiler selects an unsigned integral type for an expression which would have been signed under ANSI mode. This message appears if the **-Xlint** option is used. Use **-Xlint**=0x200 to suppress this message. (w)

1672: scope of tag *tag* **is only this declaration/definition**

The tag referred to in a parameter list does not have a prior definition. (w)

```
/* struct bar does not have a definition before this point */
foo(struct bar a);
```

1674: template argument *argument* **should be pointer/reference to object with external linkage**

Arguments for template functions need to be pointers or references to objects with external linkage. (e)

```
template <class T, int& Size>
class Base {
....
};

class A {
...
} ;

static int local_linkage_int;

Base<A, local_linkage_int> ob;
```

1675: sizeof expression assumed to contain type-id type-id (use "*typename***")**

When a **type-id** is used in a **sizeof** expression, the compiler assumes that this is intended; otherwise a typename should be used instead. (w)

```
template <class T, int& Size>
class Base {
...
    void incr()
    {
        Size = Size + sizeof(A);
    }
...
};
```

1676: class *class* **is abstract because it doesn't override pure virtual** *function*

A class that has un-overridden pure virtual functions is an "abstract class" and cannot be instantiated. (i)

1677: executable *executable name* **not found in profiling database** *database name*
This applies to programs compiled and run in the RTA (Run-time Analysis tools). (w)

The specified executable was not found.

1678: snapshot *snapshot name* **not found in profiling database** *database name*
This applies to programs compiled and run in the RTA (Run-time Analysis tools). (w)

The snapshot containing profiling information was not found.

1679: no definition found for inline function *function*
The template member function referred to has no definition. (w)

1680: delete called on incomplete type *type*
The **delete** operator is called on a pointer to a type whose full declaration has been deferred. (w)

1682: "(unsigned) long long" type is not supported by the ANSI standard
The ANSI standard does not support the **long long** type. (w; future error)

```
long long x;
```

1683: non-int bit-fields are not supported by the ANSI standard
The ANSI standard allows bit-fields of integer type only. (w; future error)

```
struct foo {
    char x:2;
};
```

1696: intrinsic *function name* **must have** *n* **argument(s)**
The number of arguments passed to an intrinsic function is incorrect. (e)

```
int a, b;
...
a = __ff1(a, b);
```

1697: invalid types on arguments to intrinsic *function name*
An argument of an invalid type is passed to an intrinsic function. (e)

```
char *ptr;
int a;
...
a = __ff1(ptr);
```

1700: implicit intrinsic *function name* **must have** *n* **argument(s) - when the intrinsic is enabled, optional user prototype must match**
When an enabled intrinsic function is redefined, the number of arguments must be the same. (e)

```
unsigned int __ff1(unsigned int x, unsigned int y)
{
    ...
}
```

1701: invalid types on prototype to intrinsic *function name* **- when the intrinsic is enabled, optional user prototype must match**

When an enabled intrinsic function is redefined, the prototypes must match. (e)

```
unsigned int __ff1(int a)
{
    ...
}
```

1702: prototype return type of intrinsic *function name* **should be** *type* **- when the intrinsic is enabled, optional user prototype must match**

When an enabled intrinsic function is redefined, the return type must match. (e)

```
void __ff1(unsigned int a)
{
    ...
}
```

1703: function name matches intrinsic *function name* **- rename function or disable the intrinsic with -Xintrinsic-mask**

A function with the same name as an intrinsic function has been defined. The function should be renamed or intrinsic functions should be disabled. (w)

```
unsigned int __ff1(unsigned int x)
{
    ...
}
```

1704: structure or union cannot contain a member with an incomplete type

Structures or unions should not contain fields of incomplete type. (w; future error)

```
struct x
{
    void a;
};
```

1707: invalid pointer cast/assignment from/to __X mem/__Y mem

The pointer assignment is invalid because it is between locations in two different memory banks. (e)

1708: cannot take address of an intrinsic function

An intrinsic function, which represents a specific CPU instruction, has no location in memory.

1709: unsupported GNU Extension : inline assembler code
The compiler does not translate extended GNU inline assembler syntax (such as register usage specification). (e)

1710: macro *macroname*: **vararg argument count does not match. expected** *n* **or more but given** *m*
Too few arguments are passed to a **vararg** macro. (w)

```
#define TEST_INFO_1(fmt, val, ...) printf(fmt, val, __VA_ARGS__)
...
TEST_INFO_1("val1 = %d, val2 = %d", 12);
```

1711: undefined identifier *identifier* **used in constant expression**
An undefined macro name occurs in a **#if** preprocessor directive. To disable this warning, use **-Xmacro-undefined-warn**. (w)

```
#if (FooDef1 == FooDef2)
# ...
#endif
```

1712: only vector literals may be used in vector initializations
Vectors can be initialized only with vector constants. (e)

```
vector int a[2] = {1, 2};
```

1713: invalid assert name *name*

1714: invalid macro name *name*

1715: no input file given

1716: memory unavailable

1717: unterminated comment

1718: unterminated character or string constant

1719: duplicate parameter name *param* **in macro** *macro*

1720: implicit include file *"file"* **not found**

1721: missing ">" in '#include <*filename***> syntax"**

1722: junk after "#include <*filename***>"**

1723: junk after "#include "filename**"**

1724: "#include" expects <*filename***> or "filename"**

1725: #if nesting too deep

1726: #include file nesting too deep. possible recursion

1727: unmatched *condition.* **block starts on line** *n*

1728: unmatched *condition*

1729: unbalanced *condition*

1730: undefined control after *expr*

1731: EOF inside #... conditional

1732,

1733: illformed macro parameter list in macro *macro*

1734: invalid macro name *name*

1735: invalid argument to macro

1736: illformed macro invocation

1737: invalid assert name *name*

1738: "##" at start of macro definition

1739: "#" precedes non macro argument name or empty argument

1740: macro *macro*: **argument count does not match. expected** *n* **but given** *m*

1741: redefinition of macro "*macro***". previously defined here**

1742: predefined macro *macro* **redefined**

1743: empty token-sequence in "#assert"

1744: no closing ")" in "#assert"

1745: garbage at the end of "#assert"

1746: invalid number in #line

1747: only a string is allowed after #line <*num***>**

1748: string expected after #error

1749: string expected after #ident

1750: # directive not understood

1751: "defined" without an identifier

1752: no closing ")" in "defined"

1753: bad digit in number

1754: bad number in #if...

1755: floating point number not allowed in #if...

1756: wide character constant value undefined

1757: undefined escape sequence in character constant

1758: empty character constant

1759: multi-character character constant

1760: octal character **constant does not fit in a byte**

1761: hex character constant does not fit in a byte

1762: character constant taken as unsigned

1763: garbage at the end of *condition* **argument**

1764: illegal identifier *identifier* in *condition*

1767: can't find include file *file* **in the include path**

1768: invalid "vector bool" constant, valid values 0, 1 or -1

1769: the called object is not a function

1770: array is too large

There is a physical limitation on the amount of space that can be allocated for an array. (e)

1771: reserved identifiers "__FUNCTION__" and "__PRETTY_FUNCTION__" may only be used inside a function

The special identifiers **__FUNCTION__** and **__PRETTY_FUNCTION__**, which return the name of the current function, can be used only within a function. (e)

1772: possible redundant expression

The compiler has encountered a valid but redundant operation, such as **x&x**. This message appears if the **-Xlint** option is used. (w)

1773: quoted section name cannot be empty, set to: *default name*

Quoted section names cannot be empty (**“”** or **“ ”**). For example,

```
.section " ",4,rx
```

will be changed to:

```
.section "default_section_name",4,rx
```

where the default section name is determined by context. (w)

1774: asm macro must be completed with "}" in the very first position

An **asm** macro must conclude with a right brace ("}") in the first column of a new line. The example below shows a valid **asm** macro. (e)

```
asm void setsr (unsigned short value)
{
%mem   value;
       move.w  value,d0
       move.w  d0,sr
}
```

1775: Deprecated use of constructor/destructor ignored, use attribute keyword

The compiler encountered an initialization or finalization function declared with the obsolete prefix **_STI__***nn*_ or **_STD__***nn*_. Use the **__attribute__** keyword to identify initialization and finalization functions, or specify **-Xinit-section=2** to use old-style initialization and finalization sections. (f)

1776: constructor/destructor priority out of range (*number***)**

The specified priority is out of range. The default range is 0-65535; but if **-Xinit-section=2** is enabled, the range is 0-99. (e)

1777: default constructor/destructor priority out of range, setting to lowest

The priority for default constructors and destructors has been set with **-Xinit-section-default-pri** to a value that is out of range. The default range is 0-65535; but if **-Xinit-section=2** is enabled, the range is 0-99. (w)

1778: option -Xc++-old is deprecated and dtoa will be removed in a future release

-Xc++-old, which invokes an obsolete version of the C++ compiler, will not be supported indefinitely. Legacy projects should be ported to the latest C++ compiler. See *Older Versions of the Compiler*, p.214 for more information. (w)

1779: CODE section without execute access mode: *section-name*

A **CODE** section has been created with a specified access mode that does not include execute permission. For example:

```
#pragma section CODE ".SOME_CODE_SECTION" RW far-code
```

In this example, **RW** (read-write) is not a valid access mode, since a **CODE** section must allow execution. **X** (execute) should be added to the access mode. (e)

1780: non-int bitfields not allowed in packed structures

Bit-fields of type **char** or **short** are nonstandard. Depending on the compilation target, such bit-fields can result in faulty code when they occur in packed structures. For example:

```
struct {
    ...
    unsigned short foo:11;
    ...
} __attribute__((packed)) struct1
```

Replace **unsigned short** with **int**. (e)

1793: conflicting types for section *section***:**

An attempt has been made to mix types of information in a single object-file section; for example, constant data (such as a string constant) into a section reserved for code or variables.

In this example, the compiler assumes from the first statement that the section **.mydata** is intended to be of the **DATA** section class, whereas the second statement assumes that **.mydata** will be a **CONST** section class:

```
__attribute__((section(".mydata"))) int var = 1;
__attribute__((section(".mydata"))) const int const_var = 2;
```

H.2.4 C++ Messages

The C++ compiler generates additional messages and diagnostics numbered 4*xxx* and 5*xxx*. No further documentation is currently available for these messages. If a message if unclear, contact Customer Support.

If you are compiling C++ code with the **-Xc++-old** option, see *H.2.3 C Compiler Message Detail*, p.595 for a list of diagnostics.

The severity of some C++ diagnostics (information, warning, error, or fatal) varies according to the circumstances under which the message is generated.

H.3 Assembler Messages

Assembler messages have the format:

```
"file", line #: severity: message
```

Three kinds of messages are generated. The severity values for each as they appear in messages are as follows.

warning
Warning: a message will be printed, assembly will continue, and an output file will be produced.

error
Error: a message will be printed, assembly will continue, but no output will be generated.

fatal
Fatal: a message will be printed and assembly aborted.

Assembler messages are intended to be clear in the context of the error and are not listed here. Please report unclear assembler error messages to Customer Support.

H.4 Linker Messages

H.4.1 Linker Message Format

Linker messages have the format:

```
DLD.EXE: message
```

Where relevant, the file and line are included in the message.

The severity level for each message is shown in parentheses in the message description. A warning (w) generates a diagnostic message, but linking continues and an output file is produced. An error (e) causes the linker to abort.

H.4.2 Linker Message Detail

"." (0x...) is assigned invalid value: 0x...
Assignment to "." creates a gap in section data. The size of this gap should not be negative and should be less 0x4000000. (e)

Absolute section has invalid name: name
Absolute section name must be ".abs.*hexNumber*". (e)

An unknown or incorrect option has been provided
The linker does not recognize an option flag that has been passed to it. (w)

Archive file *filename* **does not have symbol tabl**e
An archive file must have a symbol table to be usable by the linker. Use **dar** to create the table. (e)

ASSERT failed: *assertion*
(Message may include the **assert** expression.) Contact Customer Support. (e)

Assignment to symbol "*symbol***" in the LECL file is ignored**
The symbol is defined in an input object file
The linker command file cannot redefine a symbol that is already defined in an input object file. (w)

Cannot allocate 0x... bytes of memory for "*name*"
The **MEMORY** directive in the linker command language is used to specify the regions from which the linker can allocate memory. When there is not enough space to contain a group, section, or **NEXT** directive, an error message is generated. (e)

Cannot allocate branch island
The linker cannot calculate the address or size of a branch island. The circular dependencies are too complex. (e)

Cannot calculate address of group
Complex circular dependencies cannot be resolved. Linker command language and implicit linking rules constitute an equation system which can be unsolvable, resulting in this or similar error message. (e)

Cannot calculate address of section *section*
Complex circular dependencies cannot be resolved. Linker command language and implicit linking rules constitute an equation system which can be unsolvable, resulting in this or similar error message. (e)

Cannot calculate OVERFLOW size expression
Complex circular dependencies cannot be resolved. An expression value depends on the address or size of a symbol or section, which in turn depends directly or implicitly on the expression value. Example:

```
X = SIZEOF(Y); Y (DATA) : { . = . + X; }
```

Linker command language and implicit linking rules constitute an equation system which can be unsolvable, resulting in this or similar error message. (e)

Cannot calculate size of group
Complex circular dependencies cannot be resolved. Linker command language and implicit linking rules constitute an equation system which can be unsolvable, resulting in this or similar error message. (e)

Cannot create branch island - section *section* **is too large**
Branch islands are created between input sections. If an input section is too large it might not be possible to create an island for that branch.

Cannot create Branch Island for Arm to Thumb call, function *name*
Contact Customer Support. (e)

Cannot create Branch Island for Thumb to Arm call, function *name*
Contact Customer Support. (e)

Cannot create position independent branch island: __SDA2_BASE_ is undefined
-Xpic-only needs the symbol **__SDA2_BASE_** to be defined. (e)

Cannot evaluate expression
Complex circular dependencies cannot be resolved. Linker command language and implicit linking rules constitute an equation system which can be unsolvable, resulting in this or similar error message. (e)

Cannot evaluate fill value expression
Complex circular dependencies cannot be resolved. Linker command language and implicit linking rules constitute an equation system which can be unsolvable, resulting in this or similar error message. (e)

Cannot evaluate value of symbol *symbol*
Complex circular dependencies cannot be resolved. Linker command language and implicit linking rules constitute an equation system which can be unsolvable, resulting in this or similar error message. (e)

Cannot find matching input sections for "..."
Input section specification does not match any input. (w)

Cannot find overflow output section "*section*"
Invalid section name in **OVERFLOW** statement. No such section defined in linker command file. (e)

Cannot get current directory name
Call to **getcwd( )** failed. (e)

Cannot rename "*filename***", error:** *message*
The host operating system reported an error renaming the file. Check the permissions on the directory where the file resides. This usually means that you are not permitted to write in that directory. (e)

Cannot write relocation table: relocation type 0*x***... is not supported by COFF**
This can occur when input and output have different formats (ELF to COFF) and some relocations cannot be converted. (e)

Cannot allocate memory (NEXT)
The **MEMORY** directive in the linker command language is used to specify the regions from which the linker can allocate memory. When there is not enough space to contain a group, section, or **NEXT** directive, an error message is generated. (e)

Cannot calculate size of section "*section***":** "." **(0x...) is assigned invalid value: 0***x***...**

Can't calculate size of section *section*: **it depends on section address ...**

Can't calculate size of section *section*: **it depends on section address....**
The section might require alignment specification
Complex circular dependencies cannot be resolved. Linker command language and implicit linking rules constitute an equation system which can be unsolvable, resulting in this or similar error message. (e)

Can't create file *name*

Can't create file *name*: ...
The host operating system returned an error when **dld** tried to create a file. The permissions in the current directory probably don't allow your **dld** command to write in the directory. (e)

Can't create tempfile *name*: ...
The host operating system returned an error when **dld** tried to create a file. The permissions in the current directory probably don't allow your **dld** command to write in the directory. (e)

Can't find file: *filename*
The linker cannot locate the specified file. (e)

Can't find library: *libname***.a**
The linker cannot locate the specified library. (e)

Can't find output section *section*
Invalid section name in linker command language expression. (e)

Can't find section *section*
Invalid section name in linker command language expression. (e)

Can't lseek on *name*: ...
Possibly an external task has shortened the file. More likely, this represents an internal error in the **dld** code. Please collect a test case to reproduce the problem and contact Customer Support. (e)

Can't open *filename*: ...
The host operating system returned an error when **dld** tried to read the file. Check the permissions on the file and the full pathname to the file. Perhaps there is a spelling error in the path. (e)

Can't open tempfile *name*: ...
The host operating system returned an error when **dld** tried to read the file. Check the permissions on the file and the full pathname to the file. Perhaps there is a spelling error in the path. (e)

Can't search unused sections, main entry symbol "*symbol***" is undefined**
This warning should not be generated since the current linker deletes such symbols silently. (w)

Can't search unused sections, main entry symbol "*symbol***" has absolute address**
This warning should not be generated since the current linker deletes such symbols silently. (w)

COMMON object is eclipsed by a function definition:
Function name: *name*
File: *filename*

A symbol of type **function** is defined with the same name as a **COMMON** object. (w)

Compression switch function "*function*" **is undefined**

PowerPC compressed code only. When **-Xmixed-compression** is on, symbols **__switch_to_uncompressed** and **__switch_to_compressed** must be defined in an input object files. (e)

Don't know where to allocate input section:
no matching input specification found in linker command file.
Section name: *section*
File: *filename*

Change linker command file to include explicit instructions on how to link this section. If the "section name" referred to in the message is **.ctors** or **.dtors**, you may be using an old linker command file that specifies **.init** and **.fini** instead of **.ctors** and **.dtors**. (w)

Don't know where to put COMMONs! No .bss and no COMMON directive
Found a **COMMON** variable but linker command file has no **.bss** nor **COMMON**. (e)

Don't know where to put small COMMONs! No .sbss and no SCOMMON directive
Found a small **COMMON** variable but linker command file has no **.sbss** nor **SCOMMON**. (e)

End of memory
All internal structures used in the linker are dynamically allocated. When the host operating system cannot provide more memory, the linker aborts with an error message. On UNIX, change the amount of memory your shell allows with the **limit** or **ulimit** command; if that does not work, increase your swap area. On Windows, increase your swap area (virtual memory). (e)

Environment variable "RTAPROJECT" must be set
The variable must be set when **-Xgenerate-vmap** is used. (This option is not intended to be set by the user.) (e)

Failed to read file *name*: ...
The host operating system reported a read error. Perhaps the file's permissions were changed by another task after **dld** opened it successfully. (e)

Failed to read file *name*: **file is empty**
The host operating system reported less data in the input file than **dld** expected. Probably the file is corrupted or was only partially written because the file system filled up before its writes were completed. You should recreate the file and retry your **dld** comm**and. (e)**

Failed to read file name from archive *name*
The host operating system reported a read error. Perhaps the file's permissions were changed by another task after **dld** opened it successfully. (e)

Failed to read from file *name*: ...

Failed to read from file *name***(...)**: ...
The host operating system reported a read error. Perhaps the file's permissions were changed by another task after **dld** opened it successfully. (e)

Failed to read from file *name*: **end of file**
The host operating system reported less data in the input file than **dld** expected. Probably the file is corrupted or was only partially written because the file system filled up before its writes were completed. You should recreate the file and retry your **dld** command. (e)

Failed to write to file *name*: ...
The host operating system reported a write error. Perhaps the file's permissions were changed by another task after **dld** opened it successfully. Perhaps the file partition has filled up, leaving insufficient room for the file. (e)

File *filename* **does not have symbol table section**

File *filename***(...) does not have symbol table section**
Invalid input file: no symbol table. (e)

File *filename* **has invalid relocation section**
File *filename***(...) has invalid relocation section**
Invalid input file: invalid reference to relocation information. (e)

File has wrong byte order, file *filename*
Invalid ELF header: Byte order neither big-endian nor little-endian. (e)

File has wrong class, file *filename*
Invalid or unsupported ELF class in input file header. (e)

File has wrong version, file *filename*
Invalid or unsupported ELF version in input file header. (e)

File is not an ELF file, file *filename*
Linker assumed file to be ELF but it does not have valid ELF header. (e)

File *filename* **is not of known format**
Supported formats are COFF, ELF, archive, and linker command language. (e)

File "*filename***", section "***section***", offset 0***x*...**: Invalid relocation:** ....
Input object file has relocation entry which cannot be processed. (e)

File type is not COFF, file *filename*
Contact Customer Support. (e)

File type is not ELF, file *filename*
Contact Customer Support. (e)

Generation of relocation entries without a symbol table is not possible
Invalid **-s** option. (e)

... has BIND address, "> *area-name*" specification is ignored
Contact Customer Support. (w)

Illegal -B option
-B must be followed by "**=**". (e)

Illegal expression
Contact Customer Support. (e)

Illegal filename *prefix*[**COMMON**]**, only * is allowed**
Input specification must be ***[COMMON]**, not **xyz.o[COMMON]**. (e)

Illegal option *option*
Option is not recognized. (w or e)

Illegal option -X*option*
Option is not recognized. (e)

Illegal usage of HEADERSZ in LECL file
Contact Customer Support. (e)

Illegal -Y option
-Y must be followed by "**,**". (e)

In file "*filename***", Section** *"section*
Section offset 0*x***..,**
Symbol "*symbol***"**
Invalid relocation entry
Input file has broken symbol table or relocation information. (e)

In file *filename***, symbol** *symbol* **has invalid value:**
symbol is undefined (state 0*x***...), but value is not zero - 0***x*...
Invalid input file: The symbol table is defective. (w)

In LECL file "*filename***", line** *number*,
name **is not allocable, "> *name*" specification is ignored**
Section or group is not allocatable; see ELF for section attributes. (w)

Input contains mix of little-endian and big-endian object files:
Aborted...
Linking a mix of little-endian and big-endian object files is not supported. (e)

Input contains mix of PPC COFF and ELF object files:
PPC COFF and ELF object files have incompatible calling conventions
Mixing PowerPC COFF and PowerPC ELF is dangerous. (w)

Input files contain code for mixed processors:
Only one file for each processor type is listed
Mixing code generated for different CPU types is dangerous. (w)

Insufficient memory
All internal structures used in the linker are dynamically allocated. When the host operating system cannot provide more memory, the linker aborts with an error message. On UNIX, change the amount of memory your shell allows with the **limit** or **ulimit** command; if that does not work, increase your swap area. On Windows, increase your swap area (virtual memory). (e)

Internal error: cannot calculate COFF header size
Contact Customer Support. (e)

Internal error: cannot calculate ELF header size
Contact Customer Support. (e)

Internal error: can't ADD symbol to non-hashed table
Contact Customer Support. (e)

Internal error: error counting undefines
Contact Customer Support. (e)

Internal error: illegal output file type
Contact Customer Support. (e)

Internal error: illegal/unsupported output format ...
Contact Customer Support. (e)

Internal error: no output file type set
Contact Customer Support. (e)

Internal error: not relocinfo
Contact Customer Support. (e)

Internal error: output buffer overflow
Contact Customer Support. (e)

Internal error: should not happen
Contact Customer Support. (e)

Invalid archive format, file *filename*
Archive file has invalid format. (e)

Invalid archive symbol table, file: *filename*
Invalid input file: The symbol table is defective. (e)

Invalid file header, file *filename* **in archive** *archive*
Contact Customer Support. (e)

Invalid fill pattern alignment, must be 1, 2, or 4
Invalid fill specification in section definition (**SECTIONS** command). (e)

Invalid fill pattern size, must be 1, 2, or 4
Invalid fill specification in section definition (**SECTIONS** command). (e)

Invalid option format: *option*
Valid format is *-optionName*[=*number*]. (e)

Invalid relocation info:
File "*filename***"**
Section "*section***"**
Section address 0*x***...size 0***x***...**
Relocating reference at address 0*x***...**
Can't relocate
Input object file has broken relocation information. (e)

Invalid section header in file "*filename***", section name "***name***"**
Invalid input file: Invalid **COMDAT** section header. (e)

Invalid value of -Xmax-long-branch= option
The option sets the maximum branch offset which does not need a branch island. Some targets (like the PowerPC) have short and long branch instructions. Valid values are 2..0x7fffffff; using the option without a value is an error. (e)

Invalid value of -Xmax-short-branch= option
Valid values are 2..0x7fffffff. Using the option without a value is an error. (e)

Machine type not supported, file *filename*
Machine type not supported, file *filename***(...)**

Invalid input file: unsupported target CPU. (e)

Memory area "*area-name***" is full**
Memory area specified in "**>** *area-name*" is full. (e)

Memory area "*area-name***" is undefined**
Invalid name in "**>** *area-name*" specification. (e)

Memory block extends over 32 bit address range: ...
memory address + *memory size* >= 0x100000000. (w)

Next alignment with zero!
Invalid argument of **NEXT( )**. (e)

No main entry point defined
Executable output needs an entry point. (e)

No section names in file *filename*
Invalid input file: no section names string table. (e)

No string table in file *filename*
Invalid input file: no string table. (e)

Nothing to link
No object files are given in the command line. (e)

Only one COMMON allowed in LECL file
More than one input specification like ***[COMMON]** is not allowed in the linker command file. (e)

Only one SCOMMON allowed in LECL file
More than one input specification like ***[SCOMMON]** is not allowed in the linker command file. (e)

Out of memory reading archive *archive*
All internal structures used in the linker are dynamically allocated. When the host operating system cannot provide more memory, the linker aborts with an error message. On UNIX, change the amount of memory your shell allows with the **limit** or **ulimit** command; if that does not work, increase your swap area. On Windows, increase your swap area (virtual memory). (e)

Output file format not specified
Contact Customer Support. (e)

Output section "*section***" contains mix of compiled for compression and normal sections: The output section will not be prepared for compression**
Mixing compressed and normal code in one section is illegal. (w)

Output sections: have overlapping load addresses
Incompatible specification of output sections. (e)

Output sections: have overlapping run-time addresses
Incompatible specification of output sections. (e)

Overlapping memory block *block*
Two or more **MEMORY** directives define the same memory area. (w)

Redeclaration of *symbol*
More than one definition of a symbol which is not **COMMON** or weak.

Register number in REGISTER( ) section specification must be in 0..*n* range
Invalid register specification. (e)

Relocation error in file *filename*: section *section* refers to local symbol *symbol* in section *section* and section *section* is not taken to output
Linker failed to remove unused sections properly. file a SPR. Contact Customer Support. (e)

Relocation error in file *filename*:
section *section* refers to local symbol *symbol* at section *section* and section *section* is purged COMDAT section
Linker failed to remove unused **COMDAT** sections. Contact Customer Support. (e)

Relocation info is not properly sorted, file *filename*, section *section*
Relocation info is not properly sorted, file *filename*(...), section *section*

Input file has broken relocation information. (e)

Section .data (DATA) is not defined
COFF output must have a **.data** section. (e)

Section e_shstrndx is not a SHT_STRTAB in file "*filename*"

Section e_shstrndx is not a SHT_STRTAB in file "f*ilename*(...)"
Invalid input file: invalid ELF header. (e)

Section *section* extends over 32-bit address range
section address + *section size* >= 0x100000000. (w)

Section .text (TEXT) is not defined

COFF output must have a **.text** section. (e)

Symbol "*symbol*" can't be declared relative
Symbol is declared as "... @ ... = ...
Section "*section*" is empty - can't be used for relative declaration
A section must have some input section to make relative declaration possible. (w)

Symbol "*symbol*" can't be declared relative
Symbol is declared as "... @ ... = ..."
Symbol "*symbol*" is absolute - can't be used for relative declaration
Base symbol must be declared inside a section. (w)

Symbol definition "*name*" not found
Symbol name is used in linker command file but symbol is undefined. (e)

Symbol definitions missing at index *index* **in** *name*
Contact Customer Support. (e)

Symbol "*symbol***" has unknown binding type**
Contact Customer Support. (e)

Symbol *symbol* **has unknown section index**
Invalid symbol table in input ELF file. (w)

Symbol *symbol* **has unknown symbol type**
Input file has a symbol of an unknown or unsupported type. (e)

Symbol *symbol* **in** *name* **is defined in unknown section**
Invalid section table in input ELF file. (w)

Symbol *symbol* **is declared with more than one size**
Symbol *symbol* **is declared with more than one size (***n* **and** *m***)**

Conflicting definition for a **COMMON** variable. (w)

Symbol *symbol* **is undefined but not used**
This warning should not be generated since the current linker deletes such symbols silently. (w)

Symbol *name* **missing. Must be defined when using shared libraries.**
This message is no longer used. (e)

Symbol or section "*name***" not found**
Invalid name in relative symbol definition in linker command file. (e)

Symbol _SDA_BASE_ is undefined
Symbol _SDA2_BASE_ is undefined
Symbol _SDA3_BASE_ is undefined
The symbol **_SDAx_BASE_** is needed to process SDA (Small Data Area) relocations. (e)

Target architecture is not specified
Unknown target. (e)

Undefined symbol "*symbol***"**
Undefined symbol "*symbol***" in file "***filename***"**
Undefined symbol "*symbol***" in file "***filename***(...)"**
An undefined symbol is referenced. (w)

Undefined symbols found - no output written
The **MEMORY** directive in the linker command language is used to specify the regions from which the linker can allocate memory. When there is not enough

space to contain a group, section, or **NEXT** directive, an error message is generated. (e)

Unknown relocation type in *name*
Contact Customer Support. (e)

Unsupported file format: "*name***"**
Supported formats are COFF, ELF, archive, and linker command language. (e)

Unsupported file type in archive
Supported formats in archives are COFF and ELF. (e)

Unsupported output file format
Selected combination of object-file format and target is not supported. (e)

Unsupported relocation type ...
Unsupported relocation type in file "*filename***"**
Input file has unsupported relocation type. (e)

Unused symbols search failure, symbol: *symbol*
The linker failed while attempting to find and delete unused symbols in object files. This could be caused by a linker bug, or by an object file that is corrupt, invalid, or in an unsupported format. (e)

Use -Xmixed-compression command line option to enable generation of compression switches
PowerPC compressed code only. The switches are codes which change the CPU mode from compressed code to normal code and back. (e)

Value of "." is undefined outside a section or group
Illegal use of "**.**" in linker command file. (e)

-Xstop-on-warning is on, linking aborted
The linker stopped after issuing a warning because the **-Xstop-on-warning** option is enabled. (e)

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z