SUPPORT NETWORK

Wind River Diab Compiler for RH850 User's Guide, 5.9.7

30 January 2020

WHEN IT MATTERS, IT RUNS ON WIND RIVER

Corporate Headquarters

Wind River
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.
Toll free (U.S.A.): +1-800-545-WIND
Telephone: +1-510-748-4100
Facsimile: +1-510-749-2010

For additional contact information, see the Wind River website:

www.windriver.com

For information on how to contact Customer Support, see:

www.windriver.com/support

Wind River Diab Compiler for RH850 User's Guide, 5.9.7

30 January 2020

TABLE OF CONTENTS

1. WIND RIVER DIAB COMPILER

1.1. About the Wind River Diab Compiler

The Wind River Diab Compiler suite provides high-performance C and C++ tools designed for professional programmers.

In addition to the benefits of state-of-the-art optimization, the Diab tools 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.

Standards and Portability

C

For C code, the compiler provides support for the ANSI X3.159-1989 standard (referred to as ANSI C or C89), the ISO/IEC 9899:1999 standard (referred to as C99), and the ISO/IEC 9899:2011 standard (referred to as C11). It includes extensions for compatibility with other compilers to simplify porting of legacy code.

Note: The Diab compiler supports the C11 language, but not its libraries.

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. For information about compiling in different modes, see Compilation Modes and Options for C Language Dialects on page 42.

C++

The Diab C++ compiler implements the ANSI C++ language standard ISO/IEC FDIS 14882:2003, often referred to as "C++03." Exceptions, templates, and run-time type information (RTTI) are fully implemented.

The Diab C++ compiler also supports the C++ language standards ISO/IEC 14882:2011 and ISO/IEC 14882:2014, often referred to as "C++14" (sometimes as "C++11/14"), and a "bare metal" subset of the C++14 library. To enable C++14 usage, use the compiler/linker option -Xdialect-c++14. For more information on this option, and the limitations of the compiler's implementation of the C++14 standard, see Compilation Modes and Options for C++ Language Dialects on page 86 and the *Wind River Diab Compiler Options Reference*.

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.

1.2. C/C++ Language and Target Architecture Documentation

Wind River recommends referring to standard C and C++ texts, and to your architecture reference, in addition to the Wind River Diab Compiler documentation set.

C and C++ Documentation

The following C references are recommended:

- the ANSI C89 standard X3.159-1989
- the ANSI C99 standard ISO/IEC 9899:1999
- *The C Programming Language* by Brian Kernighan and Dennis Ritchie (aka "K&R")

For C++, see:

- the ANSI C++03 standard ISO/IEC FDIS 14882
- the C++11/14 standards ISO/IEC 14882:2011 and ISO/IEC 14882:2014
- *The C++ Programming Language* by Bjarne Stroustrup,
- *The Annotated C++ Reference Manual* by Margaret A. Ellis and Bjarne Stroustrup
- *The C++ Standard Template Library* by P.J. Plauger et al.

Architecture Documentation

See the following for details about microprocessor architecture and instructions:

- *RH850 Compiler ABI Specification*
- *V850E3v5 Architecture Specifications*

2. DRIVERS AND SUBPROGRAMS

2.1. About dcc and dplus Drivers and Subprograms

Both the C and C++ drivers invoke various subprograms.

Depending on the input files and options used on the command line, the **dcc** (for C) or **dplus** (for C++) driver may run up to five subprograms: a preprocessor, a compiler, the assembler, and the linker. The driver may also invoke optimization subcomponents.

The default operation for dcc and dplus is to invoke the preprocessor, compiler, assembler, and linker in sequence. The output from each operation is the input for the next. For example, the output from the compiler is an assembly language file that is processed by the assembler. By default, intermediate files are deleted when they are no longer needed.

The basic syntax for the drivers is as follows:

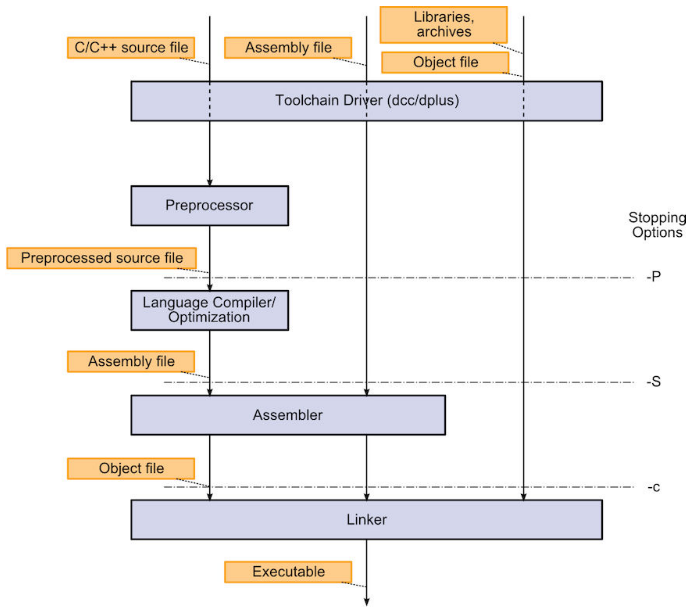
`dcc` [*options*] [*input-files*]

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

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.

2.2. About Subprogram Flow

The subprogram flow illustration shows input files, toolchain elements, stopping options, and processing flow.



2.3. Subprograms, File Name Extensions, and Stopping Options

Each subprogram has default input and output file extensions; and most have stopping options.

Sub-program	Default Input Extension	Stopping Option	Default Output Extension	Function and Stopping Option
`C preprocessor`		`-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 legacy C-to-assembly compiler (which includes a parser, optimizer, and code generator) generates assembly source from preprocessed C source. Not used to compile C++ source. For more information see Compilation Modes and Options for C Language Dialects on page 42.
`etoa`	`.cpp .cxx .cc .C (UNIX)`	`-S`	`.s`	The new C/C++ to assembly compiler (which includes a parser, optimizer, and code generator) generates assembly source from preprocessed C/C++ source. For more information see Compilation Modes and Options for C Language Dialects on page 42.
`das`	`.s`	`-c`	`.o`	The assembler; generates linkable object code from assembly source.

`dld`	`.o .a .dld`
	`.lnk`
			`a.out`	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.

3. DRIVER PROGRAM USAGE

3.1. Compiler Driver Programs

Wind River recommends using the compiler with the **dcc** (for C) or **dplus** (for C++) driver program.

The syntax for **dcc** and **dplus** is 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.

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. See the *Wind River Diab Compiler Options Reference* for details about the options.

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 the *Wind River Diab Compiler Getting Started* 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 About Setting Options on page 14 for details.

Note: The length of the command line is limited by the drivers' 1000-byte internal buffer. To pass longer commands to the tools, see the entry for **-@** and **-@@** in the *Wind River Diab Compiler Options Reference*.

For details about the available options, see the *Wind River Diab Compiler Options Reference*.

About Option Precedence on page 6
Options can come from several sources: the command line, environment variables, configuration files, and so forth.
Option Case-Sensitivity on page 6
Command-line options are case-sensitive.
About Spaces in Command-Line Parameters on page 6
For easier reading, command-line options may be shown with embedded spaces in documentation, although they are not typically written this way in use.

About Value Quoting on page 7
When a command-line parameter can take a string as a value, it does not require quotes.
About Unrecognized Options on page 8
Options beginning with a letter other than "X" and which are not listed in the Wind River Diab Compiler Options Reference are automatically passed by the driver to the linker.
dcc Example on page 8
Various diab options are explained in a sample command line.

3.1.1. About Option Precedence

Options can come from several sources: the command line, environment variables, configuration files, and so forth.

In most cases, if an option appears more than once from whatever source, the final instance is taken. Any exceptions are documented in the individual option descriptions.

See also About Configuration Variable Precedence on page 15.

Parent topic: Compiler Driver Programs on page 5

Related information
Option Case-Sensitivity on page 6
About Spaces in Command-Line Parameters on page 6
About Value Quoting on page 7
About Unrecognized Options on page 8
dcc Example on page 8

3.1.2. Option Case-Sensitivity

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.

Parent topic: Compiler Driver Programs on page 5

Related information
About Option Precedence on page 6
About Spaces in Command-Line Parameters on page 6
About Value Quoting on page 7
About Unrecognized Options on page 8
dcc Example on page 8

3.1.3. About Spaces in Command-Line Parameters

For easier reading, command-line options may be shown with embedded spaces in documentation, although they are not typically written this way in use.

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 using 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 "**=**".

Parent topic: Compiler Driver Programs on page 5

Related information
About Option Precedence on page 6
Option Case-Sensitivity on page 6
About Value Quoting on page 7
About Unrecognized Options on page 8
dcc Example on page 8

3.1.4. About Value Quoting

When a command-line parameter can take a string as a value, it does not require quotes.

For example:

```
-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" );
```

Parent topic: Compiler Driver Programs on page 5

Related information
About Option Precedence on page 6
Option Case-Sensitivity on page 6
About Spaces in Command-Line Parameters on page 6
About Unrecognized Options on page 8
dcc Example on page 8

3.1.5. About Unrecognized Options

Options beginning with a letter other than "X" and which are not listed in the Wind River Diab Compiler Options Reference 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.

Parent topic: Compiler Driver Programs on page 5

Related information
About Option Precedence on page 6
Option Case-Sensitivity on page 6
About Spaces in Command-Line Parameters on page 6
About Value Quoting on page 7
dcc Example on page 8

3.1.6. dcc Example

Various diab options are explained in a sample command line.

The following example is written on several lines for clarity. The individual options shown are documented in the *Wind River Diab Compiler Options Reference*.

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

Options	Descriptions
`-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 About UFLAGS1, UFLAGS2, DFLAGS Configuration Variables on page 21): `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**).

Parent topic: Compiler Driver Programs on page 5

Related information
About Option Precedence on page 6
Option Case-Sensitivity on page 6
About Spaces in Command-Line Parameters on page 6
About Value Quoting on page 7
About Unrecognized Options on page 8

3.2. About Compilation Examples

Examples are provided to illustrate compilation.

In these examples 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);
}
```

3.2.1. Compiling Simple Programs

When compiling small programs, the driver can be invoked to execute all four stages of compilation in one command.

About This Task

This example demonstrates use of the -O option to optimize output and the -o option to change the name of the linked output.

Procedure

1. Execute the following command to build an optimized executable in one step.

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

   Note: When only one file is compiled, assembled, and linked, the intermediate assembly and object files are deleted automatically (for information about changing this default operation see Preserving Assembly Output on page 12).When more than one file is compiled to completion, object files are kept, in this case, **file1.o** and **file2.o**

   Note: To compile for execution on the host system using the WindISS simulator, compile the program with **windiss** specified on the command line, for example:

   ```
   dplus -O -ttof:windiss file1.c file2.cpp -o prog1
   ```

   where *tof* are the target, object file format, and floating point support (as described in the *Wind River Diab Compiler Getting Started*).

   The driver preprocesses, compiles, optimizes, and assembles the two files (one C and one C++), and links them together with the appropriate libraries to create a single executable file. The default output name, **a.out** is replaced by **prog1**. .
2. (Optional step): Use ddump to convert the linked output to **S** records.

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

This produces the file **srec.out** by default. See the *Wind River Diab Compiler Utilities Reference: D-DUMP File Dumper* for additional information.

Parent topic: About Compilation Examples on page 9

Related information
Executing on the Target on page 11
Recompiling Individual Files on page 11
Preserving Assembly Output on page 12

3.2.2. Executing on the Target

Run a compiled program on a target with command-line access.

About This Task

Context for the current task

Procedure

Execute **a.out** with some arguments on the target:

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

Note: To run a program compiled for WindISS:

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

This will print:

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

Parent topic: About Compilation Examples on page 9

Related information
Compiling Simple Programs on page 10
Recompiling Individual Files on page 11
Preserving Assembly Output on page 12

3.2.3. Recompiling Individual Files

Separate compilation is a time-saving solution when recompiling larger programs.

About This Task

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:

Procedure

1. Use the following command to create object files and then stop.

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

   The files **file1.o** and **file2.o** will be created.
2. Create the executable program as follows.

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

   Note: 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**.

3. Alter **file2.cpp**.
4. Rebuild **prog2**.

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

   Note: The compilation process is usually automated with utilities similar to make, which finds the minimum command sequence to create an updated executable.

Parent topic: About Compilation Examples on page 9

Related information
Compiling Simple Programs on page 10
Executing on the Target on page 11
Preserving Assembly Output on page 12

3.2.4. Preserving Assembly Output

Two options are available to preserve and examine assembly code.

About This Task

Procedure

Complete one of the following

Option	Description
Use the -S option stops compilation after generating the assembly.	`dplus -O -S file1.cpp` The file is automatically named *basename* **.s**, **file1.s** in this example:
Use the -Xkeep-assembly-file option with a command that generates an object file.	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` This will preserve the assembly file in addition to the object, naming it *basename* **.s**.

Parent topic: About Compilation Examples on page 9

Related information
Compiling Simple Programs on page 10
Executing on the Target on page 11
Recompiling Individual Files on page 11

3.3. Executing Driver Programs

Learn how to execute complier drivers.

About This Task

Procedure

Use one of the following methods to execute a driver:

Option	Description
Add *versionDir* **/hostDir/bin to your path for UNIX or** *versionDir* **\hostDir\bin for Windows.**	
Create an alias or batch file that includes the complete path directly.	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.
Do the following to invoke an older copy of a driver:	• 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.

Option	Description
	• Create an alias or batch file that includes the complete path of the desired version.

3.4. About Setting Options

If a tool is executed with no options on the command line, no configuration file, and no environment variables set, then all options have their default values.

In practice, each tool is usually executed with some options set 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.

3.5. Compiler -X Options

Compiler command-line -X options provide fine control over many aspects of the compilation process.

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 0*x*), 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. 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.

3.6. Option Defaults

How are options interpreted if they are missing, or missing a value?

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: -Xname =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.

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

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

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.

For more information about -X options, see the Compiler Options Reference User's Guide.

3.7. About Configuration Variable Precedence

Configuration variables may be set in three places:

- In the operating system environment (see the *Wind River Diab Compiler Getting Started*).
- 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.

3.8. Startup Processing Overview

Each tool processes the command line and configuration files at startup in the following manner.

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 described in Standard Configuration Files on page 17. 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.

Table 1. Simple 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)	
`DFLAGS:   -O`	As **DFLAGS** is a convenient way to give options with an - environment variable. For more information, see the *Wind River Diab Compiler Getting Started*.
Command line `dcc -ttarget2 -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.

About Configuration File, Precedence, and Use on page 16
For the most part, configuration files are used internally by the compiler suite to support multiple target processors.
About the DENVIRON Configuration Variable on page 17
The **DENVIRON** configuration variable is set in **default.conf**.
Standard Configuration Files on page 17
Standard versions of two configuration files, **dtools.conf** and **default.conf**, are provided in the compiler suite installation. A customized **user.conf** file can be created by the user.
About UFLAGS1, UFLAGS2, DFLAGS Configuration Variables on page 21
Configuration file processing gives you several ways to provide options.
About UAFLAGS1, UAFLAGS2, ULFLAGS1, ULFLAGS2 Configuration Variables on page 21
UAFLAGS and ULFLAGS provide before and after options for the assembler and linker.

3.8.1. About Configuration File, Precedence, and Use

For the most part, configuration files are used internally by the compiler suite to support multiple target processors.

The default target configuration is stored in the *versionDir* **/conf/default.conf** configuration file.

A tool (**dcc**, **dplus**, **das**, or **dld**) identifies the main configuration file using the **DCONFIG** environment variable as described in Startup Processing Overview on page 15. If **DCONFIG** is not set, it then looks for **dtools.conf** in the *installDir* **/** *version* **/conf/** directory.

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.conf** file is structured broadly as shown in Table 1 on page 18. This shows how the compiler combines the various environment variables and command-line options. **dtools.conf** also serves as an example of how to write the configuration language.

Wind River recommends that you do not modify **dtools.conf**. If an alternate configuration is needed, do one of the following:

- Set defaults and specific options by using the -t option on the command line to set **DTARGET**, **DOBJECT**, **DFP**, and **DENVIRON** (for information about selecting a target, see the *Wind River Diab Compiler Getting Started*).
- Create your own **user.conf** file.

The **dtools.conf** configuration file includes **default.conf** and **user.conf** near the filepath. 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.

Parent topic: Startup Processing Overview on page 15

Related information
About the DENVIRON Configuration Variable on page 17
Standard Configuration Files on page 17
About UFLAGS1, UFLAGS2, DFLAGS Configuration Variables on page 21
About UAFLAGS1, UAFLAGS2, ULFLAGS1, ULFLAGS2 Configuration Variables on page 21

3.8.2. About the DENVIRON Configuration Variable

The **DENVIRON** configuration variable is set in **default.conf**.

The **DENVIRON** configuration variable 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 Table 1 on page 18, 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 (for more information, see the *Wind River Diab Compiler Getting Started*).

Parent topic: Startup Processing Overview on page 15

Related information
About Configuration File, Precedence, and Use on page 16
Standard Configuration Files on page 17
About UFLAGS1, UFLAGS2, DFLAGS Configuration Variables on page 21
About UAFLAGS1, UAFLAGS2, ULFLAGS1, ULFLAGS2 Configuration Variables on page 21

3.8.3. Standard Configuration Files

Standard versions of two configuration files, **dtools.conf** and **default.conf**, are provided in the compiler suite installation. A customized **user.conf** file can be created by the user.

Configuration File, Precedence, and Use

For the most part, configuration files are used internally by the compiler suite to support multiple target processors. The default target configuration is stored in the *versionDir* **/conf/default.conf** configuration file.

A tool (**dcc**, **dplus**, **das**, or **dld**) identifies the main configuration file using the **DCONFIG** environment variable as described in Startup Processing Overview on page 15. If **DCONFIG** is not set, it then looks for **dtools.conf** in the *installDir* **/** *version* **/conf/** directory.

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.conf** file is structured broadly as shown in Table 1 on page 18. This shows how the compiler combines the various environment variables and command-line options. **dtools.conf** also serves as an example of how to write the configuration language.

Wind River recommends that you do not modify **dtools.conf**. If an alternate configuration is needed, do one of the following:

- Set defaults and specific options by using the -t option on the command line to set DTARGET, DOBJECT, DFP, and DENVIRON (for information about selecting a target, see the *Wind River Diab Compiler Getting Started*).
- Create your own **user.conf** file.

The **dtools.conf** configuration file includes **default.conf** and **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.

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 Table 1 on page 18, 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 (for more information, see the *Wind River Diab Compiler Getting Started*).

Table 1. 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 About the DENVIRON Configuration Variable on page 17.

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.
	`$DFLAGS`	**$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 *$UFLAGS1*. See also the *Wind River Diab Compiler Getting Started*.
	`$*`	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.

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 Table 1 on page 18, **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 Table 1 on page 18, 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-count=800000
```

- **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.

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 Table 1 on page 18 for additional details.

Parent topic: Startup Processing Overview on page 15

Related information
About Configuration File, Precedence, and Use on page 16
About the DENVIRON Configuration Variable on page 17
About UFLAGS1, UFLAGS2, DFLAGS Configuration Variables on page 21
About UAFLAGS1, UAFLAGS2, ULFLAGS1, ULFLAGS2 Configuration Variables on page 21

3.8.4. About 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.
- *UFLAGS1* is expanded before command-line options and files, and UFLAGS2 after command-line options.

  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*, but are written without a "$" when being set by an assignment statement.

  ```
  # Limit memory for optimization.
  UFLAGS2=-Xparse-count=800000
  ```

- *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.

Parent topic: Startup Processing Overview on page 15

Related information
About Configuration File, Precedence, and Use on page 16
About the DENVIRON Configuration Variable on page 17
Standard Configuration Files on page 17
About UAFLAGS1, UAFLAGS2, ULFLAGS1, ULFLAGS2 Configuration Variables on page 21

3.8.5. About UAFLAGS1, UAFLAGS2, ULFLAGS1, ULFLAGS2 Configuration Variables

UAFLAGS and ULFLAGS provide before and after options for the assembler and linker.

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.

Parent topic: Startup Processing Overview on page 15

Related information
About Configuration File, Precedence, and Use on page 16
About the DENVIRON Configuration Variable on page 17
Standard Configuration Files on page 17
About UFLAGS1, UFLAGS2, DFLAGS Configuration Variables on page 21

3.9. About the Configuration Language

The purpose of configuration file processing is to provide values for options. The simplest type of configuration file is an ordinary text file containing multiple lines in which 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.

About Configuration Statements and Options on page 22
A configuration file consists of a sequence of *statements* and *options* separated by whitespace.
About Configuration Language Comments on page 23
Pound signs (#) are used to comment text in configuration files.
About String Constants on page 23
About Variables on page 24
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.
Sample Configuration Variables Reference on page 25
Review examples illustrating how configuration variables are set.
Configuration Statements Reference on page 25
Find implementation details for configuration language statements.

3.9.1. About Configuration 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 About Variables on page 24). 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
```

Parent topic: About the Configuration Language on page 22

Related information
About Configuration Language Comments on page 23
About String Constants on page 23
About Variables on page 24
Sample Configuration Variables Reference on page 25
Configuration Statements Reference on page 25

3.9.2. About Configuration Language Comments

Pound signs (#) are used to comment text in configuration files.

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"
```

Parent topic: About the Configuration Language on page 22

Related information
About Configuration Statements and Options on page 22
About String Constants on page 23
About Variables on page 24
Sample Configuration Variables Reference on page 25
Configuration Statements Reference on page 25

3.9.3. About 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
```

Parent topic: About the Configuration Language on page 22

Related information
About Configuration Statements and Options on page 22
About Configuration Language Comments on page 23
About Variables on page 24
Sample Configuration Variables Reference on page 25
Configuration Statements Reference on page 25

3.9.4. About 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.

Setting a variable in a configuration file involves an assignment statement. (See Configuration Statements Reference on page 25) and evaluating a variable, that is, using its value, involves preceding it with a " **$**" character. See Startup Processing Overview on page 15 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 the example below and also Table 1 on page 18.

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.)

Example: 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
```

Parent topic: About the Configuration Language on page 22

Related information
About Configuration Statements and Options on page 22
About Configuration Language Comments on page 23
About String Constants on page 23

Sample Configuration Variables Reference on page 25
Configuration Statements Reference on page 25

3.9.5. Sample Configuration Variables Reference

Review examples illustrating how configuration variables are set.

Variable assignment examples

See About Variables on page 24 to understand the assumptions behind the behavior described here.

Variable	Evaluates To	Comment (see assumptions above)
$DFLAGS	"-XO"	Environment variable.
$DFP	"soft"	Value is as if **- WD** set the **DFP** configuration variable (see the *Wind River Diab Compiler Options Reference*).
$-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**.

Parent topic: About the Configuration Language on page 22

Related information
About Configuration Statements and Options on page 22
About Configuration Language Comments on page 23
About String Constants on page 23
About Variables on page 24
Configuration Statements Reference on page 25

3.9.6. Configuration Statements Reference

Find implementation details for configuration language statements.

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.
```

error Statement

The **error** statement terminates configuration file processing with an error. See the **switch** statement for an example (switch Statement on page 27).

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**.

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
```

include Statement

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

include *file*

The contents of the 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 Standard Configuration Files on page 17.)

If the given file does not exist, the statement is ignored. Example:

```
include user.con
```

print Statement

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

print *string*

Example:

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

require Statement

The **require** statement is identical to the **include** statement (include Statement on page 27) in all respects but one: it is an error if a required file does not exist.

switch Statement

The **switch** statement 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
```

remove Statement

In order to suppress warnings from Diab tools when they encounter a user-specified option, use the **remove** statement in the **user.conf** file.

The **remove** statement removes a variable of type **-***x*, from the set of specified options, where *x* is a user-specified option that has just been used on the command line.

For example, assume that you add the following code for your option -Xmy_own_option to **user.conf**:

```
if ($-Xmy_own_option){
   print My own option has been set!
}
```

When the following command is executed:

```
dcc -Xmy_own_option file.c
```

the Diab tool (**dcc** in this case) generates the following output:

```
My own option has been set!
warning (dcc:1561): unknown option -Xmy_own_option
```

To avoid generating a warning about your custom option, change the code in **user.conf** to the following:

```
if ($-Xmy_own_option){
    print My own option has been set!
    remove -Xmy_own_option
}
```

This ensures that any configuration files that are parsed after **user.conf**, or tools that are subsequently invoked, do not encounter the option and generate the warning.

Parent topic: About the Configuration Language on page 22

Related information
About Configuration Statements and Options on page 22
About Configuration Language Comments on page 23
About String Constants on page 23
About Variables on page 24
Sample Configuration Variables Reference on page 25

4. TARGET CONFIGURATION

4.1. About Target Configuration

The compiler tools use a *target configuration* to set the compiling and linking parameters for a given target processor. Setting the target configuration selects the startup modules and libraries used in compilation.

A complete target configuration specifies the following:

- target processor (e.g., ARMV8A)
- object module format (e.g, ARMv8 AAPCS ABI, little-endian code, big-endian data)
- type of floating point support (e.g., software floating point)
- execution environment (e.g., using a RAM disk for I/O)

4.2. Setting a Target Configuration

You can set the target configuration for compilation in the following ways:

Using dctrl -t

> The dctrl -t command can be used before build time to set a new default target configuration. dctrl -t sets the default target configuration for all subsequent builds. Once you have run dctrl -t, you do not need to specify the target configuration when compiling, linking, or assembling, except when you want to use a non-default configuration.
>
> dctrl -t can be used in these ways:
>
> - Interactively. Typing dctrl -t with no arguments brings up a wizard that walks you through your the target configuration.
> - With command-line arguments. Using the syntax described in Target Configuration Syntax on page 29, you can specify all of the configuration parameters manually.
>
> Refer to Viewing the Default Compilation Configuration on page 30 for information on how to see the default compilation configuration.

Using the -t option with a driver (dcc or dplus)

> You can specify the target configuration parameters for a specific compilation with -t, as described in Target Configuration Syntax on page 29. This is useful when you want to use a non-default compilation configuration.

Using configuration variables

> See Using Configuration Variables to Set the Target Configuration on page 32.

4.3. Target Configuration Syntax

The syntax for target configuration is the same for both dctrl -t and drivers.

The syntax is:

```
command  -ttof :env
```

where

- *command* is either a driver (dcc or dplus) or dctrl
- *t* is the target processor
- *o* is the object file format
- *f* is the floating point support
- *env* is the execution environment

Valid configuration parameters for each architecture are listed in the spo1502313626755.html. You can also run the dctrl -t interactive wizard, by specifying no arguments on the command line, to see valid choices.

For example, the following command

```
$ dctrl -tARMV8RFS:cross
```

sets the default compilation target configuration to:

- ARMV8R device (*t* = "ARMV8R")
- ARM AAPCS ABI, little-endian code, big-endian data object module format (*o* = "F")
- software floating point (*f* = "S")
- use RAM disk for I/O (*env* = "cross")

Likewise, the following command

```
$ dcc -tARMV7AMN:simple
```

sets the target configuration for the current compilation to:

- ARMV7A device (*t* = "ARMV7A")
- ARM AAPCS ABI, little-endian code, little-endian data object module format (*o* = "M")
- no floating point (*f* = "N")
- use only character I/O (*env* = "simple")

See Setting the Execution Environment on page 31 for information about compilation environments.

4.4. Viewing the Default Compilation Configuration

These methods may be used to see the current default compilation configuration:

- Use the -Xshow-target option, e.g.:

  ```
  % dcc -Xshow-target
  ```

- View the file **default.conf** in the *versionDir* **/conf** subdirectory.

To see the options associated with a particular **-t** option, invoke a compiler driver with the **-t** option, the -# option (which causes the driver to show the command line used to invoke each tool), and the -Wa, -# option (which causes the assembler, when invoked by the driver, to show options which it extracts from the configuration files).

4.5. Setting the Execution Environment

Set the appropriate execution environment for compilation with the *env* element of the -t option.

Environments

(See Target Configuration Syntax on page 29 for target configuration syntax.)

The following are permitted values for the *env* element. (Not all environments are valid for all targets.)

cross

Use a RAM disk for I/O

other

Enter a user-defined string here

rtp

Use VxWorks user mode (see below)

simple

Use only character I/O

vxworks*xy*

Use VxWorks kernel mode (see below)

windiss

Use the Diab WINDISS instruction set simulator

VxWorks Environments

To build VxWorks applications, you must specify the appropriate VxWorks execution environment.

- For user mode (RTP or real-time process), use the :rtp option: -t*tof*:rtp
- For kernel mode, use the :vxworks*xy* option, where *xy* represents a VxWorks release number (for example, "69" signifies VxWorks release 6.9): -t*tof*:vxworks69.

For more information, see the documentation that accompanied your VxWorks development tools.

Note: The following behaviors apply when using the Diab compiler in VxWorks development:

- When you specify a VxWorks execution environment (:rtp or :vxworks*xy*), the standard C libraries linked to your application are different from the compiler's native C libraries.
- The Wind River Diab Compiler is not provided for VxWorks development for all processors. Consult your VxWorks documentation to determine if your processor is supported.
- Specifying a VxWorks execution environment turns on -Xieee754-pedantic by default.

4.6. Using Configuration Variables to Set the Target Configuration

In special cases, it may be useful to use methods other than the standard commands for changing the target configuration.

Target configuration information is stored in the following *configuration variables*:

- *DTARGET* for the processor
- *DOBJECT* for the object module format
- *DFP* for the type of floating point support
- *DENVIRON* for the target execution environment

These configuration variables are stored in *versionDir* **/conf/default.conf**.

To change the values of these variables, you may do any of the following.

- Manually edit the **default.conf** configuration file to change the default settings for any of the *DTARGET*, *DOBJECT*, *DFP*, and *DENVIRON* 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 the *Wind River Diab Compiler Options Reference*). This overrides both the values of the variables in **default.conf** and environment variables. For example:

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

Note: For additional information, and order of precedence when more than one of these methods is used, see Standard Configuration Files on page 17, and Startup Processing Overview on page 15.

5. DIRECTORY STRUCTURE

5.1. About Components and Directories

All of the files that make up the Wind River Diab Compiler and its associated programs, utilities, header files, and so on are located in subdirectories of a single root directory.

Terminology

The following terminology is used to refer to the root and related subdirectories:

- *installDir* represents the full pathname of the root directory. The root directory contains *versionDir* 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.
- *versionDir* is the name of the complete path for a single version of the compiler.
- *hostDir* is the name of a subdirectory under *versionDir* 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

Default Installation Names

These names for a default installation depend on the host file system. The following table assumes that the version number is 5.9*x* and shows examples for common installations. For other systems, see the installation procedures shipped with the media.

System	**Default** *versionDir*	**Default with** *hostDir*
UNIX	**/usr/lib/diab/5.9.***x*	**/usr/lib/diab/5.9.***x* **/***host*
Solaris		**/usr/lib/diab/5.9.***x* **/SUNS**
Linux		**/usr/lib/diab/5.9.***x* **/LINUX386**
PCs	**C:\diab\5.9.***x*	**C:\diab\5.9.***x* **\***op-sys*
Windows		**C:\diab\5.9.***x* **\WIN32**

Note: Instructions and examples for Windows apply to all supported versions of Microsoft Windows. 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 "`/`".

Program Files on page 34
The subdirectories of *versionDir* include compiler suite programs.
Configuration, Header, and Source Files Reference on page 34
The subdirectories of *versionDir* include configuration files, header files, and source files.
Architecture Startup Module and Libraries Reference on page 35
The subdirectories of *versionDir* include architecture startup modules and libraries.
Floating Point-Specific Libraries and Sub-Libraries Reference on page 36
The subdirectories of *versionDir* include floating point libraries.

5.1.1. Program Files

The subdirectories of *versionDir* include compiler suite programs.

Subdirectory or File	Contents or Use
hostDir **/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 architecture-specific description file controls assembly.
dld	The linker. Generates executable files from one or more object files and object libraries (archives).
dar	dar archiver. Creates an object library (archive) from one or more object files.
dbcnt	**dbcnt** basic block counter. Generates profiling information from files compiled with -Xblock-count.
dctrl	Utility to set default target for compiler, assembler, and linker.
ddump	ddump 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.
llopt	Assembly-level optimizer started by the driver. It reschedules the instruction sequence to avoid stalls in the processor pipeline and does some peephole optimizations. The -W1 compiler option is passed to llopt.
hostDir **/lib/**	Programs and files used by programs in **bin**.
ctoa **etoa**	C and C++ compilers. A separate architecture-specific description file directs code generation.

Parent topic: About Components and Directories on page 33

Related information
Configuration, Header, and Source Files Reference on page 34
Architecture Startup Module and Libraries Reference on page 35
Floating Point-Specific Libraries and Sub-Libraries Reference on page 36

5.1.2. Configuration, Header, and Source Files Reference

The subdirectories of *versionDir* include configuration files, header files, and source files.

Subdirectory or File	Content or Use
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 About dcc and dplus Drivers and Subprograms on page 3 for details. Other **.conf** files for particular boards or operating systems may also be present.
default.dld	Default linker command file. Alternative and sample **.dld** linker command files are also found in this directory. For PowerPC, for example, processors using the VLE (Variable Length Encoding) instruction set use different default linker command files.) For more information, see the discussion of defaults in the *Wind River Diab Linker User's Guide*.
dmake/	**dmake** startup files. See the *Wind River Diab Compiler Utilities Reference: dmake Makefile Utility*.
example/	Example files used in the *Getting Started* manual and elsewhere.
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 the *Wind River Diab Compiler C Library Reference*.
src/	Source code for replacement routines for system calls. These functions must be modified before they can be used in an embedded environment. See Compiler Options and Pragmas for Embedded Development on page 169.

Parent topic: About Components and Directories on page 33

Related information
Program Files on page 34
Architecture Startup Module and Libraries Reference on page 35
Floating Point-Specific Libraries and Sub-Libraries Reference on page 36

5.1.3. Architecture Startup Module and Libraries Reference

The subdirectories of *versionDir* include architecture startup modules and libraries.

Subdirectory or File	Contents or Use
ArchName **E/**	ELF library and startup code directories for RH850, which is little-endian.
crt0.o	Start up code to initialize the environment and then call **main**. The source for **crt0.o** is **crt0.s** under the directory **src/**.
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 (for more information, see the *Wind River Diab Compiler Getting Started*).

Subdirectory or File	Contents or Use
	For example, for PowerPC, **PPCE/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. **PPCE/simple/libc.a** includes the above four sublibraries plus **libchar.a** providing basic character I/O. **PPCE/cross/libc.a** includes the above four sublibraries plus **libram.a**, which adds RAM-disk-based file I/O. For details, see the *Wind River Diab Compiler C Library Reference*.
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**.
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).
libdabr.a	C++ customers who do not want or need to use exception handling and run-time type identification (RTTI) but who want the additional standard library functions for C++ found in **libd.a** should link against **libdabr.a** (with -ldabr). See C++ Standard and Abridged Libraries on page 87 for more information, including how to also not use dynamic memory handling (e.g., **malloc( )** and **free( )**).
libg.a	Functions to generate debug information for some debug targets.
windiss/libwindiss.a	Support library for WindISS instruction-set simulator when supplied. Note: implicitly also uses **cross/libc.a**.

Parent topic: About Components and Directories on page 33

Related information
Program Files on page 34
Configuration, Header, and Source Files Reference on page 34
Floating Point-Specific Libraries and Sub-Libraries Reference on page 36

5.1.4. Floating Point-Specific Libraries and Sub-Libraries Reference

The subdirectories of *versionDir* include floating point libraries.

Support for No Floating Point

Table 1 on page 37 displays ELF floating point stubs for floating point support of "None" (represented in target configurations as "N") (e.g., for PowerPC, **PPCEN**).

Table 1. No Floating Point

Subdirectory or File	**Contents or Use**
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.

ELF Software Floating Point Libraries

Table 2 on page 37 displays the libraries for ELF software floating point support, represented in target configurations by "S" (e.g., for PowerPC, **PPCES**).

Table 2. ELF Software Floating Point Support

Subdirectory or File	**Contents or Use**
libcfp.a	Floating point functions called by user code.
libimpfp.a	Conversions between floating point and other types.
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**.

ELF Vector and Hardware Floating Point Libraries

Table 3 on page 37 displays directories and files for vector or hardware floating point support, represented in target configurations as "V" (vector) or "H" (hardware).

Table 3. ELF Vector/Hardware Floating Point Support

NECE	RH850 v1, v1e 16-bit SDA addressing.
NECF	RH850 v1, v1e 32-bit absolute addressing.
NECE2E	RH850 v2/v3, 16-bit SDA addressing.
NECE2G	RH850 v2/v3, 23-bit SDA addressing.
NECE2L	RH850 v2/v3, 23-bit absolute addressing.
NECE2F	RH850 v2/v3, 32-bit absolute addressing.

Note: The linking of separately compiled object modules with different floating point settings can lead to erroneous behavior. This can occur when pre-compiled objects/libraries are compiled with a particular setting then linked with some other floating point setting; in particular, the mismatch of 'F' vs. 'G'. This can lead to one module assuming that a double is 32-bits and in the other module it is 64-bits. Argument passing and return values from/to these modules would be invalid.

Parent topic: About Components and Directories on page 33

Related information

Program Files on page 34
Configuration, Header, and Source Files Reference on page 34
Architecture Startup Module and Libraries Reference on page 35

6. COMPILER STANDARDS AND LIMITS

6.1. ANSI C89 Standard

The Wind River Diab Compiler meets or exceeds the C89 Standard.

C89 Minimum Limits and Compiler Specifications

The following lists the minimum limits required by Section 2.2.4.1 of the ANSI X3.159-1989 C standard. The Wind River Diab Compiler meets or exceeds these limits in all cases. When not limited only 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: no limit)
- 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 (Wind River: no limit)
- 31 parameters in one function definition and call (Wind River: no limit)
- 31 parameters in one macro definition and invocation (Wind River: no limit)
- 509 characters in a logical source line (Wind River: no limit)
- 509 characters in a string literal (after concatenation)
- 32,767 bytes in an object
- 255 case labels in a **switch** statement

6.2. ANSI C99 Standard

The Wind River Diab Compiler meets or exceeds the C99 Standard.

C99 Minimum Limits and Compiler Specifications

The following lists the minimum limits required by the C99 standard ISO/IEC 9899:1999. The Wind River Diab Compiler meets or exceeds these limits in all cases. When not limited only by available memory (effectively unlimited), the C and C++ limit is shown in parentheses. "No limit" is shown in some cases for emphasis.

- 127 nesting levels of compound statements, iteration statements, and selection statements
- 63 nesting levels of conditional inclusion (Wind River: no limit)
- 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic, structure, union, or incomplete type in a declaration
- 63 nesting levels of parenthesized declarators within a full declarator

- 63 nesting levels of parenthesized expressions within a full expression
- 63 significant initial characters in an internal identifier or a macro name (each universal character name or extended source character is considered a single character)
- 31 significant initial characters in an external identifier (each universal character name specifying a character short identifier of 0000FFFF or less is considered 6 characters, each universal character name specifying a character short identifier of 00010000 or more is considered 10 characters, and each extended source character is considered the same number of characters as the corresponding universal character name, if any)
- 4095 external identifiers in one translation unit
- 511 identifiers with block scope declared in one block
- 4095 macro identifiers simultaneously defined in one preprocessing translation unit (Wind River: no limit)
- 127 parameters in one function definition (Wind River: no limit)
- 127 arguments in one function call (Wind River: no limit)
- 127 parameters in one macro definition (Wind River: no limit)
- 127 arguments in one macro invocation (Wind River: no limit)
- 4095 characters in a logical source line (Wind River: no limit)
- 4095 characters in a character string literal or wide string literal (after concatenation)
- 65535 bytes in an object (in a hosted environment only)
- 15 nesting levels for #included files
- 1023 case labels for a switch statement (excluding those for any nested switch statements)
- 1023 members in a single structure or union
- 1023 enumeration constants in a single enumeration
- 63 levels of nested structure or union definitions in a single struct-declaration-list

6.3. About Symbol Name Limitations

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).

6.4. About Memory Limitations

Memory is dynamically allocated as required.

Memory allocation

Memory allocation 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.

Memory Limitations on Optimization

In addition, the amount of memory the compiler is allowed to use to delay code generation in order to perform inter-procedural optimizations is limited internally Use **-Xparse-count** to specify the number of nodes that the compiler should use for building internal tables; roughly, there's one node for each operator or operand, and the more nodes allocated, the more memory is allocated. The default value is 300,000 nodes with **-O** and 600,000 nodes with -XO. For information about these options, see the *Wind River Diab Compiler Options Reference*.

Debugging Limitations

The compiler does not generate correct debug information if there are more than 1023 included files.

7. C LANGUAGE COMPATIBILITY MODES

7.1. Compilation Modes and Options for C Language Dialects

C code can be compiled in conformance with several different standards. These choices may facilitate porting of existing C programs.

Choosing the Latest Standards

We recommend that customers who want to use the latest standards use -Xdialect-std, which always selects the latest C (and C++) standards, and corresponding libraries, supported by Diab. Only very rarely should customers need to explicitly pick an older standard. Code written for older standards will usually work fine with -Xdialect-std.

To avoid breaking legacy applications, the default standard for C is C89 with legacy Diab C libraries. (For C++ the default is the C+
+03 standard.)

Modes and Options

Mode	Option	Note
ANSI	`-Xdialect-ansi`	Conform to ANSI X3.159-1989 with some additions as shown in ANSI vs. Strict ANSI Compilation Mode on page 43. Synonyms: -Xansi, -Xa
Strict ANSI	`-Xdialect-strict-ansi`	Conform strictly to the ANSI X3.159-1989 standard. See also ANSI vs. Strict ANSI Compilation Mode on page 43. Synonyms: -Xstrict-ansi, -Xc.
C11	`-Xdialect-c11`	Conform to the ISO/IEC 9899:2011 standard. Note: The Diab compiler supports the C11 language, but not its libraries.
C89	`-Xdialect-c89`	Conform to the ISO/IEC 9899:1990 standard.
C99	`-Xdialect-c99`	Conform to the ISO/IEC 9899:1999 standard. Synonym: -Xc-new.

The preceding table is relevant to C and not to C++, with the following exception: -Xdialect-strict-ansi (and its equivalent -Xstrict-ansi) does affect the C++ compiler. For more information, see Compilation Modes and Options for C++ Language Dialects on page 86 and refer to the description in the *Wind River Diab Compiler Options Reference*.

The default for this architecture is to use the C89 compilation mode (with the **etoa** compiler and the -Xdialect-c89 option).

7.2. ANSI vs. Strict ANSI Compilation Mode

The compilation modes for ANSI and strict ANSI handle various data types, key words, syntax elements, and so on, differently.

Modes and Functionality

Functionality	**ANSI**	**Strict ANSI**
long float is same as **double**.		
The **long long** type is defined, and a warning is generated when **long long** is used.		x
The **long long** type is defined, but no warning is generated when when **long long** is used.	x	
The **asm** keyword is defined.	x	
The **volatile**, **const**, and **signed** keywords are defined.	x	x
"Double underscore" keywords (for example, __**inline**__ and __**attribute**__) are defined.	x	
The type of a hexadecimal constant = 0x80000000 is **int**.		
The type of a hexadecimal constant = 0x80000000 is **unsigned int**.	x	x
In ANSI it is legal to initialize automatic arrays, structures, and unions. The compiler always accepts this and is silent.	x	x
In ANSI it is legal to initialize automatic arrays, structures, and unions. The compiler always accepts this but gives a warning.		
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.	x	
When two integers are mixed in an expression, they cause conversions and the result type is "unsigned wins". Example: `((unsigned char)1 > -1)` which is 0 if (u) and 1 if (s).		
When two integers are mixed in an expression, they cause conversions and the result type is "smallest possible wins".	x	x

Functionality	**ANSI**	**Strict ANSI**
When a bit-field is promoted to a larger integral type, sign is always preserved.	x	
When prototypes are used and the arguments do not match an error is generated.	x	x
When prototypes are used and the arguments do not match a warning is generated.		
Float expressions are computed in **float**.	x	x
Float expressions are computed in **double**.		
When an array is declared without a dimension in an invalid context an error is generated.	x	x
When an array is declared without a dimension in an invalid context a warning is generated.		
When an array is declared with a zero dimension, a warning is generated.		x
Incompletely braced structure and array initializers are parsed top-down. May be controlled by the -Xbottom-up-init option.	x	x
Incompletely braced structure and array initializers are parsed bottom-up. May be controlled by the -Xbottom-up-init option.		
When pointers and integers are mismatched, an error is generated. May be controlled by the **-Xmismatch-warning** option.	x	x
When pointers and integers are mismatched, a warning is generated. May be controlled by the **-Xmismatch-warning** option .		
Trigraphs (for example, "**??**") sequences, are recognized.	x	x
Illegal structure references generate an error. 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.	x	x
Illegal structure references generate a warning. If more than one defined structure contains a member, an error is generated.		
Comments are replaced by a space.	x	x
Comments are replaced by nothing.		
Macro arguments are replaced in strings and character constants. Example: `#define x(a) if (a) printf("a\n");`		
A missing parameter name after a **#** in a macro declaration generates an error.		x

Functionality	**ANSI**	**Strict ANSI**
Characters after an **#endif** directive will generate a warning.		x
Preprocessor errors are errors.	x	x
Preprocessor errors are warnings.		
Preprocessor recognizes vararg macros. (Not available with -Xpreprocessor-old option.)	x	x
__STDC__ macro is predefined to 0.	x	
__STDC__ macro is predefined to 1.		x
__STDC__ macro is not predefined.		
__STDC__ macro can be undefined with **#undef**.	x	
__STRICT_ANSI__ macro is predefined.		x
Spaces are legal before C preprocessor **#**-directives.	x	x
Parameters redeclared in the outer most level of a function will generate an error.	x	x
Parameters redeclared in the outer most level of a function will generate a warning.		
If the function setjmp( ) is used in a function, variables without the **register** attribute will be forced to the stack.		
If the function setjmp( ) is used in a function, variables without the **register** attribute will be forced to registers (r).	x	x
C++ comments "**//**" are recognized in C files.	x	
Predefined macros without leading underscores (for example, **unix**) are available.	x	
The following construct, in which a newly defined type is used to declare a parameter, is legal: `f(i) typedef int i4; i4 i; {}`		

7.3. C Dialect Default Compilation Mode

The default for this architecture is to use the C89 compilation mode (with the **etoa** compiler and the -Xdialect-c99 option).

The -## option can be used to print subprogram command lines with arguments, including those that specify the compiler and compilation mode.

We recommend that customers who want to use the latest standards use -Xdialect-std, which always selects the latest C (and C++) standards, and corresponding libraries, supported by Diab. Only very rarely should customers need to explicitly pick an older standard. Code written for older standards will usually work fine with -Xdialect-std.

7.3.1. About Alternate Compilation Modes

For compiling C source code, invoking **dcc** with -Xc-new or -Xc-old determines which compiler is used, either **ctoa** or **etoa**, and one of the -Xdialect... options determines which compilation mode is used. In turn, these choices determine whether the legacy Diab C libraries (for C89) or the Dinkumware C libraries (for C99) are used. Note that the **ctoa** compiler cannot be used with C99 mode, it only supports C89 mode. For information about the compiler options, see Compilation Modes and Options for C Language Dialects on page 42 and the *Wind River Diab Compiler Options Reference*.

Note: Compiling C++ code (with the **dplus** driver) always invokes the **etoa** compiler, and links with one of the Dinkumware C++ libraries, depending on the linker option used (see C++ Standard and Abridged Libraries on page 87).

Parent topic: C Dialect Default Compilation Mode on page 45

Related information
About Compilation Mode and C Library Version on page 46

7.3.2. About Compilation Mode and C Library Version

The compilation mode determines which versions of the C library is linked. For C89 the legacy Diab libraries are used (see the *Wind River Diab Compiler C Library Reference*). For C99, the Dinkumware C libraries are used (see the *Dinkum C99 Library* reference, which is included in the Diab compiler documentation set).

Parent topic: C Dialect Default Compilation Mode on page 45

Related information
About Alternate Compilation Modes on page 46

8. COMPILER IMPLEMENTATION-DEFINED BEHAVIOR

8.1. C Standard and Implementation Defined Features

The ANSI C standard X3.159-1989 leaves certain aspects of a C implementation to the tools vendor.

This section describes how Wind River has implemented these details. Note that there are differences between C and C++; this appendix addresses C only.

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

Translation Reference on page 47
Environment Reference on page 49
The function called at startup is called main( ). It can be defined in three different ways.
Library Functions Reference on page 49
Library functions have certain Wind River defined characteristics.

8.1.1. Translation Reference

Diagnostics

See the error messages reference guide.

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 the -Xwchar entry in the *Wind River Diab Compiler Options Reference.*

Unless specified by the use of the -Xchar-signed or -Xchar-unsigned options, the treatment of plain **char** as a **signed char** or an **unsigned char** is as defined in Table 1 on page 98. For information about -Xchar-signed and -Xchar-unsigned, see the *Wind River Diab Compiler Options Reference*).

Integers

Integers are represented in two's-complement binary form. The properties of the different integer types are defined in Table 1 on page 98..

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 numerator 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 Table 1 on page 98.

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.

For more information about the implementation of structures and unions, bit-fields, and enumerations, see Table 1 on page 98, About Bit-Fields on page 101, Classes, Structures, and Unions on page 102 and About C++ Classes on page 102.

Anonymous Unions

Anonymous unions are not supported by the C89 standard. They are supported with the C99 standard. To compile for C99 use the –Xdialect-c99 option.

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 the *Wind River Diab Compiler Options Reference*). The name of the included file is passed to the operating system (Pragmas on page 57).

The **#pragma** directives supported are described in Pragmas on page 57.

The preprocessor treats a pathname beginning with "**/**", "****", and a "driver letter" (**c :**) as an absolute pathname. All other pathnames are taken as relative.

Parent topic: C Standard and Implementation Defined Features on page 47

Related information
Environment Reference on page 49
Library Functions Reference on page 49

8.1.2. Environment Reference

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

Definition of main( ) With No Arguments

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

Definition of main( ) With Two Arguments

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 transformed 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[]) {...}
```

Definition of main( ) With Three Arguments

With three arguments, **argc** and **argv** are used 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[]) {...}
```

Parent topic: C Standard and Implementation Defined Features on page 47

Related information
Translation Reference on page 47
Library Functions Reference on page 49

8.1.3. Library Functions Reference

Library functions have certain Wind River defined characteristics.

Implementation-Specific Library Function Characteristics

- 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:

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 the entries for setbuf( ) and setvbuf( ) in the *Wind River Diab Compiler C Library Reference* 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 generated by the perror( ) and strerror( ) functions may be found in file **errno.h** in the **sys** subdirectory of the **include** subdirectory (see About Components and Directories on page 33 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 Program Arguments, Environment Variables, and Predefined Files on page 182). 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.

Parent topic: C Standard and Implementation Defined Features on page 47

Related information
Translation Reference on page 47
Environment Reference on page 49

9. ADDITIONS TO ANSI C AND C++

9.1. Predefined Preprocessor Macros

The compiler provides a set of predefined preprocessor macros.

Predefined Preprocessor Macro Definitions

Note: Macros that do not start with two underscores ("__") are not defined if option -Xdialect-strict-ansi is given.

__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 1 when compiling C++ code, otherwise undefined.

__DATE__

The current date in "*mm dd yyyy* " format; it cannot be undefined.

__DCC__

The constant 1. Use this to test whether the Wind River Diab Compiler is being used.

_DIAB_TOOL

Indicates the Wind River Diab Compiler is being used. The **__DCC__** macro is a better choice because it is not disabled by -Xdialect-strict-ansi.

__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 file name, including its path. This macro 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).

__hardfp

Hardware floating point support.

__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.

__nofp

No floating point support.

__v850__

Target flag used by various tools.

__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 the -Xchar-signed, -Xchar-unsigned entry in the *Wind River Diab Compiler Options Reference*.

__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.

9.2. Preprocessor Directives

The preprocessor recognizes a set of additional directives that can be used for definition of preprocessor variables, error display and management, comment insertion, file inclusion, and message display.

#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 the -Xcpp-dump-symbols entry in the *Wind River Diab Compiler Options Reference*.

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 on page 56 and #warn and #warning Preprocessor Directives on page 56,.

#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 on page 55 and #warn and #warning Preprocessor Directives on page 56.

#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 on page 55 and #info, #inform, and #informing Preprocessor Directives on page 56.

9.3. About Pragma Operation

Pragma directives are not preprocessed. A warning is issued for unrecognized pragmas. 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.

Pragmas on page 57
The compiler supports a set of pragmas to allow use special features in its handling of input.

9.3.1. Pragmas

The compiler supports a set of pragmas to allow use special features in its handling of input.

align Pragma

```
#pragma align [ ([[max_member_alignment],[min_structure_alignment] [, byte-swap]] )]
```

The **align** pragma, provided for portability, is a synonym for pack Pragma on page 63.

align_functions Pragma

```
#pragma align_functions alignment
```

Use this pragma to change the default alignment of functions (for function declarations and definitions). If set and non-zero, *alignment* must be a power of two. If *alignment* is zero or not specified, alignment is set to the default.

See also always_inline Pragma on page 58.

align_variables Pragma

```
#pragma align_variables alignment
```

Use this pragma to change the default alignment of variables (for global and static variable declarations and definitions). If set and non-zero, *alignment* must be a power of two. If *alignment* is zero or not specified, alignment is set to the default.

See also align_functions Pragma on page 57.

always_inline Pragma

```
#pragma always_inline function ,...
```

The **always_inline** pragma marks a function to be inlined. This can be used to force inlining if optimizations are disabled or if a function contains more nodes than the limit set by -Xinline (see the *Wind River Diab Compiler Options Reference*).

Note: If inlining is explicitly disabled this pragma has no effect. This will happen with **-g** or **-g2**. To allow inlining, use **-g3** instead. See the **-g** entry in the *Wind River Diab Compiler Options Reference*.

See also always_inline Attribute on page 74.

contract Pragma

The **FP_CONTRACT** pragma is included as part of C99 compliance. The **FP_CONTRACT** pragma causes floating expressions to be evaluated as atomic expressions, omitting rounding errors implied by the source code and the expression evaluation method.

```
#pragma STDC FP_CONTRACT [ ON | OFF | DEFAULT ]
```

FP_CONTRACT pragmas may be placed either outside external declarations or preceding all explicit declarations and statements in a compound statement. The pragma stays in effect until another **FP_CONTRACT** pragma is reached; if no **FP_CONTRACT** pragma is encountered, the pragma stays in effect until either the end of the file (for pragmas placed outside external declarations) or the end of the compound statement (for pragmas placed inside a compound statement). See the C99 standard for specific information on placement of the **FP_CONTRACT** pragma.

By default, contraction is off.

error Pragma

```
#pragma error "string"
```

Display "*string*" on standard error as an error and halt compilation. See also info Pragma on page 60 and warning Pragma on page 67.

flatten Pragma

```
#pragma flatten function ,...
```

If possible, inline all function calls made by the function marked with this pragma.

See also flatten Attribute on page 75.

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 Register Usage Reference on page 112 for a list of valid register names.

The following rules apply:

- Only registers that are preserved across function calls may be assigned to global variables.

  Using fixed registers like **r4** or **r5** as global registers may result in undefined behavior.
- 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.

This pragma is useful for reserving a register for use with the linker **REGISTER** specification. (For more information about the **REGISTER** specification, see the *Wind River Diab Compiler Linker User's Guide*.)

The pragma must appear before the first definition or declaration of the variable being assigned to a register. 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
```

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.

hdrstop Pragma

```
#pragma hdrstop
```

Suppress generation of precompiled headers. Headers included after **#pragma hdrstop** are not saved in a parsed state. See About Precompiled Header File Reuse on page 95 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 on page 58 and warning Pragma on page 67.

inline Pragma

```
#pragma inline func ,...
```

Inline the given function whenever possible. The pragma must appear before the definition of the function. Unless whole-program optimization (WPO) is enabled, a function can be inlined only in the module in which it is defined. See the -Xwhole-program-optim entry in the *Wind River Diab Compiler Options Reference* and Whole-Program Optimization on page 130 for more on whole-program optimization.

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. Example:

```
#pragma inline swap

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

Note: The **inline** pragma has no effect unless optimization is selected (with the -XO or **-O** options).

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. Note the following:

- 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 Embedding Assembly Code on page 115). 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.
- 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 */
}
```

For crtical interrupts, there exists:

```
#pragma interrupt_critical function , . . .
```

For more information about interrupt functions, see the entries for -Xinterrupt..., -Xnested-interrupts (including register use requirements), and -Xstack-probe in the *Wind River Diab Compiler Options Reference*.

io Pragma

```
#pragma io varname  location
```

The io pragma centralizes the IO register definitions. The io pragma locates the variable *varname* at the fixed address *location*. The variable is not part of the symbol table of the compilation unit. Other modules do not link to the symbol. The reason for this configuration is that the expected use is to include a central header file with the corresponding IO register definition into any files that need access to such registers.

Note: This pragma is used in the header files supplied by Renesas which define the more than 11,000 hardware registers on the RH850. It is not expected that users will make use of it outside of this context except in unusual circumstances.

Even though the symbols declared with #pragma io are not part of the symbol table for the executable program, they will be included in the debug information if the source is compiled with an option to generate debug information. This option allows symbolic debuggers to recognize the symbolic names as equivalent to the corresponding fixed addresses.

no_alias Pragma

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

Ensure that the variable *var1* is not accessed in any manner (through pointers etc.) other than through the variable name; ensure 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:

```
#pragma no_alias *d, *s1, *s2
add(double *d, double *s1, double *s2, int n)
{
        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 the -Xargs-not-aliased entry in the *Wind River Diab Compiler Options Reference*.

noinline Pragma

```
#pragma noinline function , . . .
```

This pragma indicates that inlining should not be used with the given function.

See Target-Independent Optimizations on page 134, for information on inlining.

See also flatten Attribute on page 75.

no_pch Pragma

```
#pragma no_pch
```

Suppress all generation of precompiled headers from the file where **#pragma no_pch** occurs. See About Precompiled Header File Reuse on page 95, for more information.

no_return Pragma

```
#pragma no_return function , . . .
```

Ensure 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

Ensures that *function* does not modify any global variables (it may use global variables).

function ({ *global* | *n* } ,...)

Ensures 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 on page 66, which also ensures 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 have 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 Keywords Reference on page 68.

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 (note that the latter also affects unions). The order of precedence is values -X options lowest, then the **packed** pragma, and __**packed**__ or **packed** keyword highest.

Note: 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, if required by the selected processor. See the -Xalign-min entry in the *Wind River Diab Compiler Options Reference*.

When a structure is packed, if 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 re-coded as:

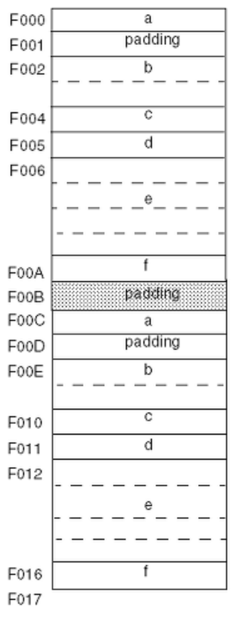
```
struct1.i = struct1.i + 3;
```

In addition, for packed structures, an **enum** member will use the smallest type sufficient to represent the range**,** see the -Xenum-is-... entry in the *Wind River Diab Compiler Options Reference*.

Note that for the following examples, in some cases later examples depend on earlier ones.

```
#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)          /* Same as #pragma pack(1,1), no padding. */
struct S1 {
    char c1                       /* 1 byte at offset 0 */
    long i1;                     /* 4 bytes at offset 1 */
    char d1;                     /* 1 byte at offset 5 */
};                                     /* total size 6, alignment 1 */

#pragma pack (8)                 /* Use "natural" 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. */
                                        /* total size 11, alignment 2 */
```

If the pack directive changes the alignment of a basic type to something the hardware cannot support, a run-time function call is inserted by the compiler. For example:

```
#pragma pack (1)
typedef struct {
    char c;
    float f;
    long l;
} S1;

float get_f1(S1* p) { return p->f; }

// compiled for hardware floating point which has alignment constraints
dcc -tPPCEH:windiss -S pack.c

     .globl get_f1
get_f1:
     stwu r1,-16(r1)
     mfspr r0,lr
     addi r3,r3,1
     stw r0,20(r1)
#$$fn 0x1ff9 0x3fff 0xfffff 0x1
#$$tl 0x8
     bl __DIAB_rd_pk_fl ; <--- read a packed float
     lwz r0,20(r1)
     mtspr lr,r0
     addi r1,r1,16
#$$tl 0x0 0x2
     blr
#$$ef
```

pure_function Pragma

`#pragma pure_function` *function* `,...`

Ensures 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 on page 62, which only ensures 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. Once a section class has been so defined, code and variables may be assigned to it with **#pragma use_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 **section** pragma is discussed in detail in C++ Code and Data Memory Location on page 167 and section and use_section Pragma Syntax on page 155.

For information on how to generate a different section name for every function or variable, see the entry for -Xsection-split-name in the *Wind River Diab Compiler Options Reference*.

unroll Pragma

```
#pragma unroll (unrollFactor)
```

Controls the unrolling factor of a single loop, where *unrollFactor* is a positive integer. This pragma overrides the command-line setting of the -Xunroll compiler option. Note that if you also use the -Xsize-opt option, the unroll factor will be set to zero, and this pragma will have no effect.

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 C++ Code and Data Memory Location on page 167 and section and use_section Pragma Syntax on page 155.

warning Pragma

```
#pragma warning "string"
```

Display "*string*" on standard error as a warning and continue compilation. See also error Pragma on page 58, and info Pragma on page 60.

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.

A global definition will override a weak definition when it is encountered. A symbol may be defined in more than one module as long as the following are both true:

- No more than one of the definitions is global.
- All the other definitions are weak.

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 the -Xlocal-data-area entry in the *Wind River Diab Compiler Options Reference*, and Local Data Area Optimization and -Xlocal-data-area on page 165.

Parent topic: About Pragma Operation on page 57

9.4. Keywords Reference

The compiler supports a set of keyword extensions, that support embedding assembly code, using a bit data type, inlining function calls, and so on.

__asm and asm Keywords

Used to embed assembly language (see Embedded Assembly Code That Accesses the Stack or Frame Pointer Directly on page 122) and use the information found in Other Additions to ANSI C and C++ on page 79.

__attribute__ Keyword

See Attribute Specifiers Reference on page 73.

__bit Keyword

The **__bit** keyword is used for the **bit** data type. It is only supported with the **etoa** compiler front end (**dplus** or **dcc** with -Xc-new).

The bit data type can only store the values zero and one, and is always unsigned. It is allowed as a structure or union member, as a function parameter, as a function return type, or as a switch expression. It can be declared as volatile or register. Arrays of the bit data type and pointers to the bit data type are not allowed.

The following operators can be used with the bit data type:

```
! ~ sizeof < > <= >= == != & ^ | &&
                || = &= ^= |=
```

The bit data type is promoted to **long double**, **double**, or **float** if the other operator argument is **long double**, **double** or **float**, respectively. Otherwise, integer promotion is performed as required.

The conversion rules for the bit data type are as follows:

- Bit-to-numeric: Bit value (0 or 1) is promoted to target types.
- Bit-to-boolean: (C++ only) A bit value of 0 is converted to false, a bit value of one is converted to true.
- Numeric-to-bit: Bit is 0 if numeric type is 0, otherwise the bit is 1.
- Boolean-to-bit: (C++ only) A boolean value of false is converted to 0, a boolean value of true is converted to 1.

bool, pixel, vec_step, and vector AltiVec Keywords

The compiler supports the following keywords for AltiVec targets: **bool**, **pixel**, **vec_step**, and **vector**.

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 (see the -Xkeywords entry in the *Wind River Diab Compiler Options Reference*).

__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 the entry for -Xinline in the *Wind River Diab Compiler Options Reference*. 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** in Target-Independent Optimizations on page 134, 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__** and **interrupt** keywords provide 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 those which are volatile, and a special return instruction is used. **__interrupt__** works like the interrupt Pragma on page 60. The **interrupt** keyword can also be used; see the -Xkeywords entry in the *Wind River Diab Compiler Options Reference*.

Example:

```
__interrupt__ void trap() {
```

Note: The example is an interrupt function.

For more information about interrupts, see the -Xinterrupt and -Xnested-interrupts compiler options.

__interrupt_critical__ and interrupt_critical Keywords

The return instruction will be generated for an interrupt function depending if the selected target supports critical and non-critical interrupts. For more information see command line option -Xcritical-interrupts as described in the *Wind River Diab Compiler Options Reference*.

For more information about interrupts, see the -Xinterrupt and -Xnested-interrupts compiler options.

long long Keyword

The compiler supports 64-bit integers for all of the architecture's 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); }
```

__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 on page 63 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 Basic Data Types Reference on page 98 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 (note that the latter also affects unions). 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 on page 63 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) struc
t
                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.

__thread__ Keyword

Supported for VxWorks only. For more information, see the VxWorks documentation.

__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.

9.5. About Attribute Specifiers

Attribute specifiers have the form **__attribute__((** *attribute-list* **))**, where *attribute-list* is a comma-delimited list of *attributes*.

As an alternate to the form **__attribute__**, you can also use **__attribute** (without the trailing underscores), but only with the **ctoa** front end, and not the **etoa** front end.

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() ;
```

When an attribute specifier modifies a **struct**, **union**, or **enum**, it can appear immediately before the keyword, or after the closing brace. Example:

```
struct b {
char b;
int a; } __attribute__((aligned(2))) str1;
```

The same applies for type definitions. Example:

```
typedef __attribute__((aligned(32))) struct myStruct {
int a;
int b; } myStruct;
```

For non-structure fields, or typedef'd 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)));
__attribute__((aligned(1024))) myStruct foo3;
myStruct _attribute__((aligned(512))) foo4;
myStruct foo5 __attribute__((aligned(32)));
```

Note: Attribute specifiers on declarations take precedence over attributes specified in type definitions.

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:

```
// a and b are aligned on 4-byte boundaries
typedef __attribute__((aligned(4))) char AlignedChar;
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.

As one would expect, attributes can be part of macro definitions. Example:

```
#define MY_ALIGNMENT_32 const __attribute__((aligned(32)))
volatile MY_ALIGNMENT_32 myStruct foo6;
```

Attribute Specifiers Reference on page 73
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.

9.5.1. Attribute Specifiers Reference

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.

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 on page 76. 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**.

always_inline Attribute

The **always_inline** attribute marks a function to be inlined. This can be used to force inlining if optimizations are disabled or if a function contains more nodes than the limit set by -Xinline (see the *Wind River Diab Compiler Options Reference*).

Note: If inlining is explicitly disabled this attribute has no effect. This will happen with **-g** or **-g2**. To allow inlining, use **-g3** instead. For information about these options, see the *Wind River Diab Compiler Options Reference*.

See also align_variables Pragma on page 57.

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.

__attribute__((constructor)) works by putting the affected functions in special sections whose names also contain the specified priority. These sections are sorted by the linker, and then gathered into an array of function pointers in **crtlibso.c**. Then __exec_ctors( ) runs each function in that array.

For more information, see Startup and Termination Code on page 170.

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 Startup and Termination Code on page 170.

flatten Attribute

If possible, inline all function calls made by the function marked with this attribute.

```
__attribute__((flatten))
```

See also flatten Pragma on page 58.

noinline Attribute

This attribute indicates that inlining should not be used with the given function. In this example, the function foo( ) will not be inlined:

```
__attribute__((noinline)) void foo() {}
```

See Target-Independent Optimizations on page 134, for information on inlining.

See also noinline Pragma on page 62.

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 on page 76). **__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;
```

Note the placement of the attribute after the structure definition. Any other placement may result in compilation errors.

When used with **aligned**, the **packed** attribute takes precedence as discussed in aligned(n) Attribute on page 73.

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 on page 76.

section(name) Attribute

To specify a linker section in which to place a function or variable, use the following syntax:

```
__attribute__((section("name")))
```

This creates a section called *name* and places the designated code in it. For 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 ELF targets, the compiler also supports several variants of the optional *flags* and **@***types* parameters, using the following syntax:

```
__attribute__ ((section("name","flags","@types")))
```

where *flags* is a quoted string that may contain any combination of the following characters:

a

Section may be allocated.

r

Read-only section.

w

Section is writable.

and where *type* may be one of the following:

@nobits

Section does not contain data (it only occupies space).

@note

Section contains data that is used by something other than the program.

@progbits

Section contains data.

(For greater control over sections, use **#pragma section**. See Code and Data Location in Memory: section and use_section Pragmas on page 155 and section and use_section Pragma Syntax on page 155.)

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 assigned to the **.data** section, not **.bss**.

See Section Classes and Their Default Attributes on page 160 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.

stack protect Attribute

Enable stack protection on this function.

```
__attribute__((stack_protect))
```

Enable stack protection on this function (equivalent to previous).

```
__attribute__((stack_protect(1)))
```

Disable stack protection on this function.

```
 __attribute__((stack_protect(0)))
```

See Stack Smashing Protection on page 149 for an explanation of Stack Smashing Protection.

use_frame_pointer Attribute

This attribute instructs the compiler to copy the stack pointer of the function that is marked to another register, and to perform all stack accesses through this register rather than using the actual stack pointer. To make a function use a frame pointer, use the following attribute designation:

```
__attribute__((use_frame_pointer))
```

This allows the user to modify the stack pointer inside of the function without endangering the integrity of the function itself (that is, its own stack frame accesses).

The original stack pointer is restored in the function epilogue.

Note: This attribute can only ensure the integrity of the stack accesses of the designated function and its callers --but not of the functions that the designated function itself calls (as they will use the potentially modified stack pointer). Manually changing the stack pointer is a hazardous undertaking and requires careful verification.

Parent topic: About Attribute Specifiers on page 71

9.6. Intrinsic Functions

The compiler implements a set of intrinsic functions to provide access to both architecture-independent functions and architecture-specific instructions. The architecture-specific instructions are referred to as intrinsic assembly.

Intrinsic Functions and Their Use

Some of the key intrinsic functions are described below. For information about others, see the header files under *installDir* **/diab/5.9.x.x/include/diab**.

See the processor manufacturer's documentation for details on machine instructions.

For RH850 targets, the -Xintrinsic-mask and -Xdisable-intrinsic-mask options have no effect.

You can specify which intrinsics should be enabled with -Xintrinsic-mask, or disabled with -Xdisable-intrinsic-mask, each of which uses a hex bit mask that corresponds to an intrinsic. Mask bits can be **OR**'d to select more than one. The procedure to selectively disable intrinsic funcions is discribed in the *Options Reference*.

The default intrinsic selection (bit mask) is 0xf.

Function	Mask	Description
`alloca` (*integral*)	default	Allocate temporary local stack space for an object of size *integral*. Return a pointer to the start of the object. The allocated memory is released at return from the current function.
`__alloca` (*integral*)	default	Same as alloca( ).
`__builtin_expect(long`*exp*, `long`*c*`)`	default	Provide the compiler with branch prediction information. This function tells the compiler that we expect the integral expression *exp* to be equal to *c*, so the compiler can optimize accordingly. *exp* is also the return value.

See also Use of Intrinsic Assembly Instead of asm Statements on page 123.

Note:

- Functions taking *integral* arguments first sign-extend their arguments to 32 bits.
- Functions taking **long long** arguments first sign-extend their arguments to 64 bits.
- These functions are not prototyped and return an **int** (32 bits) by default. A prototype may be used to define a different return type.

Note: Note that __attribute__((intrinsic_function)) is part of the implementation of intrinsic functions. It is used by Wind River to implement intrinsic functions and is not intended for end-user use.

Intrinsic Assembly Example

```
void enable () {
_ei();
}
```

9.7. Other Additions to ANSI C and C++

The compiler provides additions to ANSI C and C++ that provide support for 64-bit bit-fields, C++ style comments, alloca( ), and so on.

Support for 64-bit Bit-fields

The Wind River Diab Compiler supports 64-bit bit-fields (e.g., they may be used in variables of type **long long**).

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 Intrinsic Functions on page 78 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; for example:

```
int some_global_var asm("r22");
```

This assigns the variable **some_global_var** to **r22**.

Note: Only registers that are preserved across function calls may be assigned to global variables. Using fixed registers like **r4** or **r5** as global registers may result in undefined behavior.

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)
```

Note that unlike the standard, one-argument version of sizeof( ), only types (not variables) are allowed as the first argument. See the __typeof__( ) extension if variables are needed (see __typeof__ Keyword (C only) on page 71).

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	Using the old front-end (-Xc-old): 1 (char is **unsigned** by default) Using the new front-end (C programs with -Xc-new, or C ++ programs): 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	23
union	24

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 */
```

Statement Expressions

The **etoa** front end supports GCC-style statement expressions. This applies to C++ programs or C programs compiled with **-Xc-new** (see the *Wind River Diab Compiler Options Reference*).

For example:

```
({int j; j = f(); j;})
```

Branches into a statement expression are not allowed. In C++ mode, branches out are also not allowed. Variable-length arrays, destructible entities, try, catch, local non-POD class definitions, and dynamically-initialized local static variables are not allowed inside a statement expression.

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") ;
>
```

Static Initialization of Flexible Array Members

The Wind River Diab Compiler supports static initialization of flexible array members when using C99. For example, consider the example below, which defines a structure with a flexible array member:

```
typedef struct array_carrier{
      unsigned int some_integer;
      unsigned char var_length_char_array[];
} array_carrier;
```

According to ISO/IEC 9899:1999 (E) §6.7.2.1 paragraph 20, any initialization attempt on **var_length_char_array** within **array_carrier** is invalid.

However, the Wind River Diab Compiler allows initialization of flexible array members like **var_length_char_array** as follows:

```
array_carrier a1 = { 1, {1,2,0}};
array_carrier a2 = { 2, {1,2,3,4,0} };
```

Note that the following:

```
sizeof(a1)==sizeof(a2)==sizeof(array_carrier)==offsetof(array_carrier, var_length_char_array)
```

as defined per ISO/IEC 9899:1999 still holds. In other words, the sizeof( ) operator is unaware of the resulting size of a structure initialized in this manner.

This behavior can be turned off by using -Xstrict-ansi.

Initialization of a Range of Array Members With Same Value

The compiler provides an extension to the C89 and C99 standards (in a manner similar to the GNU compiler) that allows you to initialize a range of array elements with the same value using the following syntax:

[*firstIndex* ... *lastIndex*] = *value*

For example:

```
int nums[] = { [0 ... 4] = 5, [6 ... 10] = 10, [7 ... 11] = 20 };
```

To use this feature you must use the -Xdialect-C89, -Xdialect-c99, or -Xc-new option. It will not work with options for earlier standards, which disable C89 and C99 features (such as -Xstrict-ansi).

Zero-Sized Structures and Unions As Members of Structures

With the -Xc-new option, the compiler supports zero-sized structures and unions as members of other structures. For example:

```
struct A {
};

struct B {
int i;
struct A a;
};
```

10. C++ FEATURES AND COMPATIBILITY

10.1. Compilation Modes and Options for C++ Language Dialects

C++ code can be compiled in conformance with several different standards. These choices may facilitate porting of existing C++ programs.

Choosing the Latest Standards

We recommend that customers who want to use the latest standards use -Xdialect-std, which always selects the latest C++ (and C) standards, and corresponding libraries, supported by Diab. Only very rarely should customers need to explicitly pick an older standard. Code written for older standards will usually work fine with -Xdialect-std.

To avoid breaking legacy applications, the default standard for C++ is the C++03 standard. (For C, the default is the C89 standard.)

C++ Modes and Options

Table 1 on page 86 lists the options for using different C++ standards.

Table 1. C++ Modes and Options

Mode	Option	Note
03	-Xdialect-c++03	Support the C++ 2003 standard.
11	-Xdialect-c++11	Support the C++ language standard ISO/IEC 14882:2011, often referred to as "C++11." See C++11/14 Support on page 86.
14	-Xdialect-c++14	Support the C++ language standard ISO/IEC 1488:2014, often referred to as "C++14" (sometimes as "C++11/14"), and a "bare metal" subset of the C++14 library. See C++11/14 Support on page 86.
Strict ANSI (C++)	-Xdialect-strict-ansi	Synonym: -Xstrict-ansi. For C++, -Xdialect-strict-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.

C++11/14 Support

By default, the Diab C++ compiler supports the ANSI C++ standard (ISO/IEC FDIS 14882:2003), aka the "C++03" standard. The Diab C++ compiler also supports the C++14 language standard (ISO/IEC 14882:2011 and ISO/IEC 14882:2014) and a "bare metal" subset of the C++14 library.

To enable C++14 usage, use the compiler/linker option -Xdialect-c++14.

Note: This option must be specified to both the compiler and the linker because it also selects a different library implementation.

The Diab C++14 library implementation includes all functionality that does not require extended operating system support. This includes all library functionality except for the following:

- Atomic Operations Library
  - atomic
- Chron and Signal Utilities
  - chrono
  - csignal
- Thread Support
  - condition_variable
  - future
  - mutex
  - shared_mutex
  - thread
- Filesystem Library
  - filesystem
- Localization Library
  - locale
  - clocale
  - codevcvt

To link against the C++ libraries use the linker option -lstl. This will automatically select the correct implementation depending on whether or not -Xdialect-c++14 is specified.

10.2. About Header Files

The C++ compiler supports all ANSI-specified header files.

Generally C++ uses the same header files as C (see the *Wind River Diab Compiler C Library Reference*), 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 About Migrating from C to C++ on page 89.

10.3. C++ Standard and Abridged Libraries

The Wind River Diab Compiler provides both regular and abridged versions of standard C++ libraries, **libstl.a** and **libd.a**. These abridged libraries are for those users who don't want predefined library support for exception handling or Run-Time Type Identification (RTTI).

Libraries with Exception Handling and RTTI

The C++ standard libraries **libstl.a** and **libd.a** provide full support for exception handling and RTTI.

To link to **libstl.a**, use the -lstl or -lstlstd option. (The options are equivalent.)

To link to **libd.a**, use the -ldr option.

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 the *Wind River Diab Compiler C Library Reference*.

Note: VxWorks developers should not specify either -lstl or -lstlstd. To select a C++ library for VxWorks projects, see the documentation that accompanied your VxWorks development tools.

The default operation for C++ is to build with exceptions enabled using the standard STL. However, no STL libraries are linked by default, so the linker option -lstlstd (or the equivalent -lstl) must be used to link in the standard libraries.

For projects that use the complete C++ library, exception-handling must be enabled (-Xexceptions, the default).

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.

Libraries without Exception Handling and RTTI

Because the abridged libraries **libstlabr.a** and **libdabr.a** don't inlcude exception-handling functions or RTTI support, they produce smaller, faster executables than the complete versions; however, the difference in size and speed varies from project to project. (Additionally, **libstlabr.a** doesn't provide complete STL functionality.) The more an application uses the unabridged libraries, the greater the benefit from switching to the abridged version.

To use **libstlabr.a** you must:

- specify the -lstlabr option to the linker
- specify the -Xc++-abr compiler option. For example:

  ```
  dplus -Xc++-abr file1.cpp
  ```

  -Xc++-abr automatically disables exception-handling (-Xexceptions=off).

To use **libdabr.a**, link with -ldabr. Note that you must specify this option to both the compiler and the linker.

Note: VxWorks developers should not specify either -lstlabr or -ldabr. To select a C++ library for VxWorks projects, see the documentation that accompanied your VxWorks development tools.

For projects that use the abridged libraries, exception handling may be enabled (with -Xexceptions), as long as no exception propagates through the library.

For more information on avoiding exception handling and RTTI, as well as how to avoid dynamic memory allocation, see Avoiding Exception Handling, RTTI, and Dynamic Memory Allocation on page 184.

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 *versionDir* **/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.

For more information on C and C++ libraries in general, see Architecture Startup Module and Libraries Reference on page 35.

10.4. About Migrating 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 that are used 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 8.27 C++ Argument Passing, p. 119.

Differences between C and C++

A few general differences between C and C++ are listed below. For more information, see C/C++ Language and Target Architecture Documentation on page 1.

- 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.

10.5. Implementation-Specific C++ Features Reference

Various features of C++ 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 Startup and Termination Code on page 170.

Templates

Function and class templates are implemented according to the standard. For strict adherence to the standard, compile using the -Xstrict-ansi option. This ensures that non-standard usage is identified as an error.

For example, the standard requires two-phase name lookup, which defines the process for looking up identifiers in templates. With -Xstrict-ansi this is enforced, but without it the lookup rules are relaxed to conform to the behavior of many older compilers.

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.

-Ximplicit-templates

Instantiate each template wherever used. This is the default.

-Ximplicit-templates-off

Instantiate templates only when explicitly instantiated in code.

-Xcomdat-info-file

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

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.

For more information about these options, see the *Wind River Diab Compiler Options Reference*.

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: For information about implementing exceptions in a multitasking environment, and about eliminating exception handling and RTTI, see Hardware Exception Handling on page 175, Library Exception Handling Modification on page 175, and Avoiding Exception Handling, RTTI, and Dynamic Memory Allocation on page 184

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 -Xnamespace-off disables namespaces; -Xnamespace-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.

Pure Virtual Function Calls

Pure virtual functions should never be called in C++; however, it is possible for a pure virtual function to be invoked, unintentionally, at runtime. (Typically, this occurs when the pure virtual function is called from the constructor of an abstract base class.) Although such an invalid call results from an error in the C++ program, the compiler cannot always detect the error at compile time.

Here is an example of a program that "accidentally" calls a pure virtual function (doSomething( )):

```
#include <stdio.h>

class AbstractBase
{
public:
         AbstractBase();
         virtual void doSomething() = 0;
    };

    void doIt(AbstractBase * p)
    {
         p->doSomething();
    }
    AbstractBase::AbstractBase()
    {
         /* Whoops--even though we're constructing a Derived object,
         at this point our type is AbstractBase, not Derived.
     The next line will end up calling AbstractBase::doSomething(),
         which is a pure virtual function */

         doIt(this);
    }

    class Derived : public AbstractBase
    {
    public:
         virtual void doSomething()
         {
               printf("Derived::doSomething\n");
         }
    };

    int main()
    {
         Derived d;
    }
```

When a pure virtual function is called, the library function __cxa_pure_virtual( ) is invoked. The default implementation of __cxa_pure_virtual( ) prints the following message and aborts:

```
C++ runtime abort: a pure virtual function was called abnormal process termination
```

However, you may provide your own definition of __cxa_pure_virtual( ). For example:

```
extern "C" void __cxa_pure_virtual( )
{
     printf("Whoops! A pure virtual function was called\n");
     abort();
}
```

10.6. C++ Name Mangling

C++ name mangling is defined by the industry-standard C++ ABI (originally known as the IA64 C++ ABI).

About Name Mangling

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.

Protecting C Functions from Mangling

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 About Migrating from C to C++ on page 89 for examples.

Demangling Utility

To interpret a mangled name, use the following command:

```
ddump -F
```

and then interactively enter mangled names one per line.

ddump displays the demangled name after each entry.

If the entry is not a valid mangled name, the output is the same as the input. **ddump** demangles names used in the current ABI as well as names from the ABI used up to release 5.8 of the compiler.

Entry to ddump	Interpreted result
`_Z6myfuncv`	`myfunc( )`
`_Z6mymainiPPc`	`mymain (int , char **)`
`mymain__FiPPc(old ABI)`	`mymain (int , char **)`

Note: If the compiler for your architecture produces mangled names with a prefix of two underscores ("__") instead of just one, use only one underscore when you use ddump –F to demangle names.

Protecting C Functions from Mangling on page 93
Learn to protect C function names from conflicts.
Using the Demangling Utility on page 94
Wind River provides a utility to assist with demangling C function names.

10.6.1. Protecting C Functions from Mangling

Learn to protect C function names from conflicts.

About This Task

When C code is linked with C++ code, the C functions must be declared appropriately.

Procedure

Declare functions with the extern "C" linkage specification.

This tells the C++ compiler not to mangle their names.

Note: The **main** function is never mangled.

See About Migrating from C to C++ on page 89 for examples.

Parent topic: C++ Name Mangling on page 92

Related information
Using the Demangling Utility on page 94

10.6.2. Using the Demangling Utility

Wind River provides a utility to assist with demangling C function names.

About This Task

To interpret a mangled name:

Procedure

1. Enter the following command:

   ```
   ddump -F
   ```

2. Interactively enter mangled names one per line.

   For example:

   ```
   _Z6myfuncv myfunc( )
   _Z6mymainiPPc mymain
   mymain__FiPPc(old ABI) mymain (int , char **)
   ```

   If the entry is not a valid mangled name, the entry is echoed unmodified. **ddump** demangles names used in the current ABI as well as names from the ABI used up to release 5.8 of the compiler.

   ddump displays the demangled name after each entry.

Parent topic: C++ Name Mangling on page 92

Related information
Protecting C Functions from Mangling on page 93

10.7. About setjmp and longjmp

Wind River recommends not using setjmp( ) and longjmp( ) in C++ code.

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: 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 de-allocations not performed through constructors and destructors of automatic objects.

10.8. Using Precompiled Header Files

Compilation time can be reduced by using precompiled header files.

About This Task

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. Precompiled headers may be used with the C++ compiler, or with the C compiler if the -Xc-new option is used. The easiest option to use is -Xpch-automatic.

Procedure

Use the format -Xpch-automatic.

```
dplus -Xpch-automatic file1.cpp
```

Note: 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.

file1.cpp is compiled using precompiled headers.

This means that a set of header files is saved in a pre-parsed 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.

10.9. About Precompiled Header File Reuse

Compilation time can be reduced by reusing precompiled header files.

A precompiled header built for source file A can only be reused when compiling source file B if the compilation environment for *B* is identical to the one for *A*. One piece of the compilation environment is the header search path. This is determined by the three factors:

- The compiler's internal header search path (which is the same so long as the same compiler is used to compile both A and B)
- Any explicit **-I** or **-YP** flags that appear on the command line
- The location of the source file being compiled. For example, if both A and B have a line like

  ```
  #include "foo.h"
  ```

  the search path for looking up **foo.h** includes the directory containing A in the first case and the directory containing B in the second case. If these directories are different (i.e. if A and B live in different directories), then the header search path is different. In a sense the command line is different--it is as if the first file was compiled with -Idir A and the second was compiled with -Idir B .

About PCH Files on page 96

Naming a PCH File on page 97

10.9.1. About 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 pre-parsed 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.

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 pre-parsed 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.

Parent topic: About Precompiled Header File Reuse on page 95

Related information
Naming a PCH File on page 97

10.9.2. Naming a PCH File

About This Task

If you want to specify a name for the generated PCH file, use **-Xpch-create=** *filename* instead of -Xpch-automatic:

Procedure

Follow the pattern in this example.

```
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
```

Parent topic: About Precompiled Header File Reuse on page 95

Related information
About PCH Files on page 96

11. INTERNAL DATA REPRESENTATION

11.1. Basic Data Types Reference

The compiler supports a standard set of C/C++ data types for the architecture's microprocessors.

C and C++ Data Types, Alignments, Sizes, and Ranges

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 1. Basic Data Types

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	4	range (-263, 263-1)
unsigned long long	8	4	range (0, 264-1)
enum (Note 2)	4	4	same as **int**
	1	1	with **-Xenum-is-best**, and fits in **unsigned char** or **signed char** (depending the default type for the architecture)
	2	2	with -Xenum-is-best, and fits in **unsigned short** or **signed short** (depending the default type for the architecture)
pointers	4	4	all pointer types; the **NULL** pointer has the value zero

Data Type	Bytes	Align	Notes
float	4	4	IEEE754-1985 single precision
double	8	8	IEEE754-1985 double precision
long double	8	8	IEEE754-1985 double precision
reference	4	4	C++: same as pointer (Note 3)
ptr-to-member	8	4	C++: pointer to member
ptr-to-member-fn	12	4	C++: pointer to member function

Notes

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**.

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 that is sufficient to represent the values of the enumeration constants is used. That is, the first of the following:

- the default **char** type for the architecture (**signed char** or **unsigned char**)
- the non-default **char** type for the architecture (**signed char** or **unsigned char**)
- **short**
- **unsigned short**
- **int**
- **unsigned int**
- **long**
- **unsigned long**

Note that **long long** is not available for enumerated types.

A reference is implemented as a pointer to the variable to which it is initialized.

Vectors are not currently supported for ppc64 targets.

The following quadword data types are not supported for ppc64 targets:

- __int128
- signed __int128
- unsigned __int 128

For more information about the options in question, see the *Wind River Diab Compiler Options Reference*.

11.2. Vector Data Types Reference

In addition to the basic data types, the compiler supports vector data types for RH850G4MH targets. To make use of these data types, set **#include <asm.h>**.

Vector Data Types

Note that there exists a "half-precision floating point" type **float16_t** which has 2 bytes size and 2 bytes alignment. This type is exclusively available as an intrinsic type in context with vector operations for target RH850G4MH. It is not a built-in type in the C language, so you can't actually use arithmetic operators (+, -, *, /) on variables of this type.

The compiler supports the following vector data types for RH850G4MH.

Table 1. Vector Data Types

Vector Data Type	Description	Bytes
float32x4_t	128-bit vector composed of 4 float elements	16
float16x4_t	64-bit vector composed of 4 float16 elements	8
int32x4_t	128-bit vector composed of 4 int elements	16
uint32x4_t	128-bit vector composed of 4 unsigned int elements	16

Note: -Xuse-fpsimd must be set to enable vector data types and intrinsics for RH850G4MH.

11.3. About Byte Ordering

Byte ordering defines whether the least significant byte or the most significant byte of any multibyte type is at the lowest address.

All data is stored in little-endian order. That is, with the least significant byte of any multi-byte type at the lowest address. To access data in big-endian order, see the *byte-swapped* parameter for the **#pragma pack** in pack Pragma on page 63 and __packed__ and packed Keywords on page 70.

11.4. About 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.

11.5. About Bit-Fields

Bit-fields can be of type **char**, **short**, **int**, **long**, or **enum**. Plain bit-fields are unsigned by default. The -Xbit-fields-signed option (C only) or the **signed** keyword can be used to make bit-fields signed.

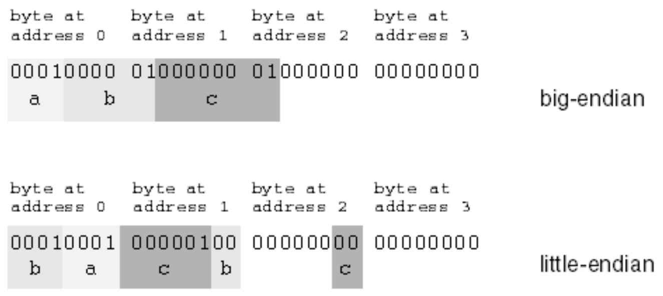
Bit-field allocation depends on whether the system is big-endian or little-endian.

Consider as an example a **struct** with three bit-fields (**a**, **b**, and **c**) of length 4 bits, 6 bits, and 8 bits. For example:

```
struct S
{
     int a : 4;
     int b : 6;
     int c : 8;
} S ;
```

The figure below shows how the bits are allocated if **a**, **b**, and **c** are all set to equal 1 (the most-significant bit is to the left):

Figure 1. Bit-Field Allocation and Endian-ness



rh850 architectures are little-endian.

Bit-field allocation follows these rules:

- Bits within a bit-field are always allocated in ascending order of address (regardless of endian-ness). Thus, the 1 in each bit-field is shown in the least-significant bit slot of that bit-field.
- Bit-fields are allocated over the bytes in the **struct** in ascending order of address, reflecting the order in which they're declared (regardless of endianess). Thus, bit-field **a** occupies byte 0, bit-field **b** occupies bytes 0 and 1, and bit-field **c** occupies bytes 1 and 2.
- Within a byte on little-endian systems, bit-fields are allocated from least-significant bit to most-significant.
- 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.
- 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 -Xbit-field-compress option (see the *Wind River Diab Compiler Options Reference*).
- When a bit-field is promoted to a larger integral type, the compiler preserves sign as well as value unless -Xstrict-bitfield-promotions, -Xdialect-strict-ansi, or -Xstrict-ansi is enabled.

11.6. Classes, Structures, and Unions

The size of classes, structures, and unions is determined by their members and the required padding. Alignment is in relationship to the size of the member with the largest alignment.

Alignment

The alignment of **class**, **struct**, and **union** aggregates is the same as that of the member with the largest alignment.

Size

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 on page 63 and __packed__ and packed Keywords on page 70 for more information.

11.7. About 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.

Internal Representation of C++ Classes

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 **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.

C Class Pointers to Members on page 104
About Virtual Function Table Generation on page 105
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).

11.7.1. C Class Pointers to Members

The pointer-to-member type (non-static) is represented by two objects. One for pointers to member functions, and one for all other pointers to member types. The offsets below are relative to the class instance origin.

An object for a pointer to non-virtual or virtual member functions has three parts:

voffset
index
vtbl-offset or Function Pointer

The *voffset* field is an integer that is used when the virtual function table is located in a virtual base class. In this case it contains the offset to the virtual base class pointer + 1. Otherwise it has a value of 0.

The index field is an integer with two possible meanings:

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 index > 0

The index field is an index in the virtual function table. The third field, *vtbl-offset* , is used as an offset to the virtual function table pointer of type integer

A null pointer-to-member function has zero for the second and third fields.

An object for a pointer-to-member of a non-function type has two parts:

voffset
moffset

The *voffset* field is used in the same way as for pointer-to-member functions.

The *moffset* field is an integer that is the offset to the actual member + 1. A null pointer to member has zero for the *moffset* field.

Parent topic: About C++ Classes on page 102

Related information
About Virtual Function Table Generation on page 105

11.7.2. About Virtual Function Table Generation

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**.

Parent topic: About C++ Classes on page 102

Related information
C Class Pointers to Members on page 104

11.8. Linkage and Storage Allocation References

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.

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).
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.

Note: Allocation of static variables is as per Section Classes and Their Default Attributes on page 160.

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

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.

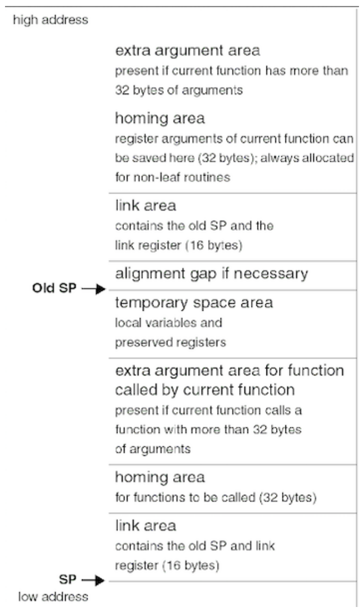
12. CALLING CONVENTIONS

12.1. Stack Layout Reference

The stack frame illustration helps describe the interface between a function caller and the called function. Stack layout, argument passing, returning results, and register use are described in detail.

AIX Stack Frame Illustration

In COFF mode, the stack layout follows the AIX Standard. The diagram shows the stack frame after completion of the prolog in the called function (**SP** = stack pointer, **r1**).


high address
extra argument area
present if current function has more than 32 bytes of arguments
homing area
register arguments of current function can be saved here (32 bytes); always allocated for non-leaf routines
link area
contains the old SP and the link register (16 bytes)
alignment gap if necessary
Old SP
temporary space area
local variables and preserved registers
extra argument area for function called by current function
present if current function calls a function with more than 32 bytes of arguments
homing area
for functions to be called (32 bytes)
link area
contains the old SP and link register (16 bytes)
SP
low address

The EABI stack frame uses considerably less stack space because it does not reserve the 32 byte homing area needed in the AIX stack frame. The homing area is used to save the argument registers if the function has a variable argument list (**stdarg**/**varargs**) function or takes the address of a parameter. The EABI assumes that this is not a common case and trades this space with a somewhat more complex **varargs** scheme.

For "classic" PowerPC architecture, to facilitate certain optimizations, the stack pointer is aligned on an 8-byte boundary by inserting a 4-byte alignment gap if necessary.

AltiVec and e500 targets diverge from the model shown above. Consult the manufacturer's documentation for more information about these architectures.

12.2. About Argument Passing

How arguments are passed to the parameters of subroutines is defined by the calling convention.

Argument Passing Operations

Registers **r6**-**r9** are used for argument passing. Arguments are assigned starting with **r6** unless the a return pointer is used. Types which require two registers start on an even register. This can leave unused argument registers.

12.3. About C++ Additions to C Conventions

In C++, the same lower-level conventions are used as in C, with several additions.

The following additions to the C low-level conventions are provided for C++ argument passing:

- 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 C++ Name Mangling on page 92.
- 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 C++ Name Mangling on page 92 and the *Wind River Diab Compiler Utilities Reference*.

C++ Argument Passing Reference on page 109
C++ argument passing additions are described in detail.
Returning Results on page 110
Data types and the registers to which they are returned.

12.3.1. C++ Argument Passing Reference

C++ argument passing additions are described in detail.

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.

Parent topic: About C++ Additions to C Conventions on page 109

Related information
Returning Results on page 110

12.3.2. Returning Results

Data types and the registers to which they are returned.

Data Types and Return Operations

Registers **r10** and **r11** are used for return values. 4-byte scalar values are returned in **r10**. If the value is less than 4-bytes it will be signed/zero extended. 8-byte scalar values are returned in **r10**/**r11**. The lower 32-bits are in **r10**, the upper 32-bits are in **r11**.

The return value for a structure may be returned in **r10**/**r11** or through a hidden argument point contained in **r6**. The criteria used to determine if registers are used depends on the return type and alignment.

Type	Size	Alignment	Use
structure, union, array, ldouble	1	-	use **r10**
structure, union, array, ldouble	2	2	use **r10**
structure, union, array, ldouble	4	4	use **r10**
structure, union, array, ldouble	8	4/8	use **r10**/**r11**
otherwise	-	-	Use pointer to return address (given in r6)

For example:

```
typedef struct Sc1 { C a; } Sc1;
typedef struct Sc2 { C a; C b; } Sc2;

// return in r10: size = 1
Sc1 returnSc1() { Sc1 s1; s1.a = 5; return s1; }

// return through pointer: size = 2, alignment = 1
Sc2 returnSc2() { Sc2 s2; s2.a = 5; return s2; }
```

Class, Struct, and Union Return Types

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.

Parent topic: About C++ Additions to C Conventions on page 109

Related information
C++ Argument Passing Reference on page 109

12.4. Data Types and Return Operations

Registers **r10** and **r11** are used for return values. 4-byte scalar values are returned in **r10**. If the value is less than 4-bytes it will be signed/zero extended. 8-byte scalar values are returned in **r10**/**r11**. The lower 32-bits are in **r10**, the upper 32-bits are in **r11**.

The return value for a structure may be returned in **r10**/**r11** or through a hidden argument point contained in **r6**. The criteria used to determine if registers are used depends on the return type and alignment.

Type	Size	Alignment	Use
structure, union, array, ldouble	1	-	use **r10**
structure, union, array, ldouble	2	2	use **r10**
structure, union, array, ldouble	4	4	use **r10**
structure, union, array, ldouble	8	4/8	use **r10**/**r11**
otherwise	-	-	Use pointer to return address (given in r6)

For example:

```
typedef struct Sc1 { C a; } Sc1;
typedef struct Sc2 { C a; C b; } Sc2;
// return in r10: size = 1
Sc1 returnSc1() { Sc1 s1; s1.a = 5; return s1; }

// return through pointer: size = 2, alignment = 1
Sc2 returnSc2() { Sc2 s2; s2.a = 5; return s2; }
```

About Class, Struct, and Union Return Types on page 112

12.4.1. About Class, Struct, and Union Return Types

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.

Parent topic: Data Types and Return Operations on page 111

12.5. Register Usage Reference

List of registers and description of their use by the compiler.

Compiler Use of Registers

Register	Comment	
r0	Zero fixed.	Base register referring to **.data**/**.bss** sections.
r1	To be saved by the caller (caller save).	
r2		Reserved for the operating system (this reservation may be cancelled by a compiler option).
r3(sp)		Stack Pointer.
r4(gp)	Fixed or to be saved by the caller (caller save).	Global Pointer for PID. This is always a fixed register.
r5(tp)	To be saved by the caller (caller save) or fixed.	Global Pointer.
r6	To be saved by the caller (caller save).	First parameter. When the return value is structure or union, the first parameter points to the address of return value area, and first, second (and so on), parameters are treated as second, third (and so on), parameters, respectively.

Register	Comment	
r7		Second parameter.
r8		Third parameter.
r9		Fourth parameter.
r10		Return value (lower 32bits).
r11		Return value (upper 32bits).
r12		
r13		
r14		
r15		
r16		
r17		
r18		
r19		
r20	Guaranteed through function call (callee save).	
r21		
r22		
r23		
r24		
r25		
r26		
re27		
r28		
r29		
r30(ep)	Fixed or guaranteed through function call (callee save).	Switch by compiler option.

Register	Comment	
r31(lp)	Guaranteed through function call (callee save) link pointer.	Link pointer.

13. EMBEDDING ASSEMBLY CODE IN C/C++ CODE

13.1. Embedding Assembly Code

Assembly code can be embedded in C and C++ source files using **asm** macros or **asm** strings.

asm and _asm Keywords

The asm keyword is used for both macros and strings. The **__asm** keyword is a synonym for asm; the two can be used interchangeably. Note that **asm** is not defined in C modules if the -Xdialect-strict-ansi option is used.

asm Strings and asm Macros

There are two ways of using the **asm** keyword: as an **asm** string or an **asm** macro.

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.
asm macro	Expanded inline where called. Depending on the arguments given, different assembly code sections may be inlined. Functions containing **asm** macros may be inlined without restriction.	Parameters matched by type per storage mode lines. 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.

Note that you can also call some assembly instructions directly (without using **asm** strings or macros) by including the file **diab/asm.h**. See Use of Intrinsic Assembly Instead of asm Statements on page 123.

❶ **CAUTION:** **When embedding assembly code, you must use only scratch registers. See** Register Usage Reference on page 112**.**

13.2. Asm Macros

While **asm** strings can be useful for embedding simple assembly fragments, they are difficult to use with variables inside the assembly code. The **asm** macro provides a more flexible way to embed assembly code in C/C++ source files.

asm Macro Syntax

An **asm** macro definition looks much like a function definition, including a return type and parameter list, and a function body.

The syntax is as follows:

`asm[volatile] [return-type] macro-name( [ parameter-list ] ){`	
`% storage-mode-list ! register-list asm-code`	(must start in column 1) ("**!**" must start in column 1)
`}`	(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 Returning Results on page 110 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. See Storage Mode Line--Describing Parameters and Labels on page 117. 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 on page 118 for details.
- *asm-code* is the code to be generated by the macro.
- final right "**}**"closes the body; it must start in column 1.

asm Macro Operation

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.

When option -Xforce-prototypes is given, an asm macro prototype is required before its first use.

The qualifier "void" is required in an asm macro prototype when no arguments are expected.

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.

It is valid for the compiler to issue either error messages (below), when checking asm prototypes:

- "etoa:5844: function ... has no prototype"
- "etoa:4195: an asm function must be prototyped"

This depends on the context that the compiler first discovers the prototype error.

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, 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 the following registers:

- **r6**, **r7**, **r8**, and **r9**

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 About Argument Passing on page 108).

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" Error

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.

In addition, 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 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 **r6**-**r19**.

In addition, 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.

If the "**!**" is present without any list, the compiler assumes that no scratch or link registers are used by the macro.

See Register Usage Reference on page 112 for more information about registers.

Note: If supplied, the *register-list* must be complete, that is, must name all scratch registers used by the macro.

It must also 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.

Optimization

Optimization can affect the storage mode of parameters to **asm** macros. For example, if the optimizer stores a frequently-used constant in a register, then it will be passed to the asm macro as a **reg** instead of a **con**. If the asm macro has a **% con** storage mode, this will generate a "no matching asm pattern exists" error. There is no way to know in advance if this will happen, but you can define a multiple-body asm macro with different storage-mode lines for each possibility.

```
__asm void fooMacro(const UINT32 param)
{
% reg param
!
 /* code for when the parameter is stored in a register */
%con param
!
  /* code for when the parameter is stored as a constant */
}
```

Another alternative, for ARM, and MIPS, Rh850, and PowerPC is to use intrinsic assembly. See Use of Intrinsic Assembly Instead of asm Statements on page 123.

CAUTION: Although the assembler (das) recognizes GNU syntax and labels, the assembly-level optimizer (llopt) does not. So, for example, mixing Wind River Diab Compiler syntax with GNU numeric register designations will produce an error. Similarly, using GNU-style locals in asm string statements will produce an error. In such cases, you must use non-GNU designations. (For information about GNU-style locals, see the ***Wind River Diab Compiler Assembler User's Guide*****.)**

asm Statements and Special Register Names

Some architecture families may include hundreds of special registers. Therefore, when using **asm** macros and **asm** strings, don't assume that the compiler (including optimizers and the assembler) will recognize a special register by name. Instead, use **#define** statements to specify special registers.

For example:

```
#define regID 0
#define sellID 5

asm volatile void LOAD_VM(int inValue)
{
%reg inValue
```

```
     ldvc.sr    inValue,regID,sellID
}

asm volatile void STORE_VM(int inValue)
{
%reg inValue
     stvc.sr regID,inValue,sellID
}
```

asm Macro Use Example #1

In this example, a test-and-set instruction is used to wait on and then seize a semaphore when it becomes free. The example assumes that the parameter **semaphore_p** points to an arbitrary (shared) memory address and that its address is contained in a CPU register.

```
asm void semaphore_seize (volatile int *semaphore_p)
{  %reg semaphore_p;
  .trylock:
      ldl.w    [semaphore_p], r1 # semaphore will be in register
      cmp      r0, r1            # link success?
      bnz      .waitloop         # No, snooze and try again
      mov      1, r1             # token for semaphore
      stc.w    r1, [semaphore_p]
      cmp      r0, r1
      bnz      .exit
  .waitloop:
      snooze
      br      .trylock
  .exit:
}

/* Unlock the semaphore */
asm void semaphore_unseize (volatile int *semaphore_p)
{

%reg semaphore_p;
    st.w      r0,0[semaphore_p] # token for semaphore
}


#pragma section CONTROL far-absolute RW address=0xf0000
#pragma use_section CONTROL mem_semaphore
volatile int mem_semaphore;
void seize (volatile int *reg_semaphore_p)
{
    semaphore_seize (& mem_semaphore);
}
```

asm Macro Use Example #2

Here is another, simpler, example of asm macros you may find in our standard example **bubble.c** from your *install_dir* /*example* sub-*directory* .

```
asm short get_short (short *address_p) /* Assembler macro: read and */
{ /* byte-swap a 2-byte I/O port. */
% reg address_p; /* Parameter passing method. */
    ld.h 0[address_p],r1 /* Get I/O port value to r1. */
    andi 0xff,r1,r10 /* Value's msb to temp reg. */
    sar 8,r1 /* Value >>= 8. */
    shl 8,r10 /* Make it the msb in the temp reg. */
    add r10,r1 /* Combine bytes. */
    andi 0xffff,r1,r10 /* place short in return reg */
}
```

13.3. Asm String Statements

An **asm** string statement provides a simple way to embed instructions in the C/C++ source code.

asm String Statement Syntax

The syntax for asm string statements is as follows:

```
asm[volatile]("string");
```

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* in asm Macros on page 115). The optional **volatile** keyword prevents instructions from being moved before or after the string statement. The volatile keyword implies a .set noreorder / .set reorder assembler directives to disable the low level optimizations on the asm statements. The volatile keyword should be preferred over using explicit .set noreorder / .set reorder directives combination. (See .set directives in the *Diab Compiler Assember User's Guide*).

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 asm Macros on page 115.

asm String Statement Operation and Restrictions

Note: **asm** string statements are primarily useful for manipulating data in static variables and special registers, changing processor status, and so on, 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. The **asm** macro facility is recommended when these issues must be taken into account.

Before using **asm** string statements, consider the following:

- 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 may or may not be inlined, depending on what type of optimization, if any, is used.
- 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. Do not, for example, precede a label with a space or a tab. For instruction lines, however, it is not required that they begin with a space, or a tab, or a label. Assembler directives may start in column one but only if the assembler -Xlabel-colon option is enabled (see the *Wind River Diab Compiler Options Reference*).
- 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.
- Although the assembler (**das**) recognizes GNU syntax and labels, the assembly-level optimizer (**llopt**) does not. So, for example, mixing Wind River Diab Compiler syntax with GNU numeric register designations will produce an error. Similarly, using GNU-style locals in **asm** string statements will produce an error. In such cases, you must use non-GNU designations. (For information about GNU-style locals, see the *Wind River Diab Compiler Assembler User's Guide*.)
- Do not assume that the compiler will recognize special register names. Use #define statements to specify special registers. (See asm Macros on page 115.)
- Do not use **.section** directives in **asm** statements with DWARF2 or later versions of the DWARF format.

Using asm String Statements to Disable Interrupts

The following sequence of **asm** string statements disables hardware interrupts.

Note: A scratch register is used in the example.

```
asm(" di");
```

13.4. Embedded Assembly Code That Accesses the Stack or Frame Pointer Directly

In general, avoid making direct use of the stack or frame pointer in embedded assembly code.

About This Task

Certain optimizations provided by the assembly-level optimizer (llopt) can resize the stack and therefore require stack access adjustments.

For example, the following would cause an error because **llopt** can only adjust instructions with an offset (for example, **number[sp]**):

```
asm(" add r4,r1,r3"); // r1 is stack pointer on
                ppc
```

For stack adjustments, please refer to **__attribute__((use_frame_pointer))**.

13.5. Use of Intrinsic Assembly Instead of asm Statements

An intrinsic assembly feature that can be used instead of **asm** statements is available for some architectures.

Intrinsic Assembly Use

To use intrinsic assembly, the file **diab/asm.h** must be included to access some assembly language instructions with function-call syntax--without using **asm** strings or **asm** macros.

Instructions that have a trivial one-on-one mapping with C operators are usually not available as intrinsic functions. Intrinsics allow the use of instructions not normally generated by the compiler. But since the compiler understands these intrinsics, optimization is not inhibited in any way.

For a complete list of available intrinsic assembly instructions, refer to the following file and the architecture-specific file it includes:

versionDir **/include/diab/asm.h**

Note: If a parameter is **const**, then the caller of this intrinsic function is not allowed to call the function by a variable; only literals are allowed in that case.

Intrinsic Assembly Example

```
#include <diab/asm.h>

int count_leading_zeros(unsigned i)
{
        return __sch0l(i);
}
```

13.6. Setting of Memory or Scheduling Barriers by Intrinsics

There are two intrinsic functions available for all Diab targets which will act as barriers between what is above the call and what is below.

Being void functions they require that the call be made in a way that acts as a C/C++ sequence point (ie. as it's own statement ending in a semicolon or in a comma expression).

```
void __scheduling_barrier(void);
```

acts as an instruction scheduling barrier.

```
void __memory_barrier(void);
```

acts as both an instruction scheduling barrier and a memory barrier.

Several processors support a memory barrier instruction. The meaning of these instructions can be different for each processor, but in general they enforce that the processors data read/write buffers flush before the instruction completes.

RH850 and V850 intrinsics are recognized directly by the compiler. The declaration will not show up in the header file.

14. OPTIMIZATION

14.1. Optimization Overview

Optimizations have two purposes: to improve execution speed and to reduce the size of the compiled program.

Compiler Options for Optimization

Most optimizations are activated by the -O option. A few are activated by the -XO option. For information about these options, see the *Wind River Diab Compiler Options Reference*.

See also the discussion of optimization and debugging with the -g option in the entry for that option. Note that using debug and optimization options together will generate a binary file that differs from one generated with only optimization options.

Types of Optimizations

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

Optimizations fall into three categories: local, function-level, and program-level.

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

- 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 call optimization
- Undefined variable propagation
- Unused assignment deletion
- Variable location optimization
- Variable propagation

Program Global Optimizations Across Multiple Functions

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

14.2. Optimization Hints

The Diab compiler attempts to produce code that is as compact and efficient as possible. However, user's have unique knowledge about their projects that allows them to use the compiler in a way that generates the most optimal code.

Balancing Speed and Size

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.

For more information, see the -Xinline entry in the the *Wind River Diab Compiler Options Reference*. For a definition of internal nodes, see the -Xunroll entry. Other options that control inlining include -Xexplicit-inline-factor and **-Xinline-explicit-force**.

-Xunroll-size=*n*

Controls the maximum size of a loop body to be unrolled. See also the -Xunroll entry in the *Wind River Diab Compiler Options Reference*.

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 inter-procedural 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-count=*n* option, where *n* is the number of nodes to use in internal table generation--roughly, one node for each operator or operand--may be used to specify how much memory should use for optimization (the more nodes, the more memory). For more information, see the -Xparse-count entry in the *Wind River Diab Compiler Options Reference*.

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, 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. See the *Wind River Diab Compiler Options Reference* for more information about these options.

The following guidelines summarize which optimizations to use in varying situations:

- 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 inter-procedural optimizations. Use -Xparse-count to increase the number of nodes to use for building internal tables, e.g.:

```
-Xparse-count=800000
```

- 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-count=800000 -Xinline=50"`
`export DFLAGS`
	(UNIX)

`set DFLAGS=-XO -Xparse-count=800000 -Xinline=50`	(Windows)

- If possible, disable exceptions and run-time type information (-Xexceptions-off, -Xrtti-off). This can reduce code size significantly.

Coding for Optimization

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;
}
```

- Allocate variables to the small data and small const areas. See the descriptions of the -Xsmall-data and -Xsmall-const options in the *Wind River Diab Compiler Options Reference*, and the description of **#pragma** section, all in Code and Data Location in Memory: section and use_section Pragmas on page 155 and section and use_section Pragma Syntax on page 155.
- 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:

```
#pragma no_alias *d, *s1, *s2
void add(double d[100][100], double s1[100], double s2[100])
{
        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 Embedding Assembly Code on page 115.
- Avoid setjmp( ) and longjmp( ). When the compiler finds setjmp( ) in a function, a number of optimizations are turned off. 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: Both ways are valid according to ANSI.

- If possible, eliminate C++ exception-handling code. For more information, see Avoiding Exception Handling, RTTI, and Dynamic Memory Allocation on page 184.

14.3. Whole-Program Optimization

Optimization is traditionally done at compile time, on object (**.o**) files, with each object module optimized separately. Whole-program optimization (WPO) enables an entire program to be optimized at once, for all modules as a unit (for example, inlining functions across all modules).

WPO Operation

With WPO, a complete representation of each translation unit, together with the compile flags, is stored in a special section of the corresponding object file (here, a *translation unit* is more or less a preprocessed source file.) These representations allow the linker to recompile the object files as needed.

Full optimization is then implemented at link time (and not compile time). At link time, the properties of global variables and functions are analyzed based on their usage throughout the program. If a variable or function is identified as unsafe , whole program optimizations are not implemented for those elements of the program. A variable or function is identified as unsafe if any of the following apply:

- Referenced from a file that was not compiled with the -Xwhole-program-optim option.
- Explicitly referenced in the linker command file.
- Defined as the entry point using the -e linker option.
- Address taken.
- Marked volatile.

WPO Compiler Options

The primary option for WPO is -Xwhole-program-optim, which creates the special sections in the object file. This option takes a *mask* value that allows you to modify default operations (such as inlining functions from other modules but not exporting functions to other modules from the module being compiled).

When -Xwhole-program-optim is used, the following options are enabled by default:

-Xwpo-inline

Allows calls to functions that are defined in different modules to be inlined.

-Xwpo-const

Detects global variables that have an initializer, but are never subsequently written. Such variables are treated as constants, and references to them are replaced by the initial value.

-Xwpo-no-alias

Detects global variables that never have their address taken. This information can be used by the rest of the optimizer to remove redundant loads and stores.

-Xwpo-delete-return-value

Detects global functions whose return values are never used, and optimizes the functions by removing computations leading to the return statements.

-Xwpo-dead

Removes (non-volatile) variables and functions that are not referenced in the program.

The following option (which is not enabled by default) is also provided:

-Xwpo-inline-single-calls

Functions that are only called once across the entire program are treated as if they were marked with **__attribute__((always_inline))**, and, if possible, inlined regardless of their size, or the value of -Xinline.

The logical pairs to these options (such as -Xno-wpo-inline) can be used to disable the default options on an individual basis.

For more information about these options, and those discussed below, see the *Wind River Diab Compiler Options Reference*.

WPO Linker Option

The -Xwhole-program-optim linker option uses the special sections created by the compiler option of the same name to recompile and optimize the object files that are being linked.

WPO Utility Options

WPO also provides the following utility options:

-Xwhole-program-output-dir= *dir*

Specify the directory for storing WPO object files (instead of using the default temporary directory, which is cleaned at the end of linking).

-Xwhole-program-diagnostics= *mask*

Print out diagnostics according to the value *mask* .

-Xwhole-program-jobs= *n*

Parallelize the link-time optimization process by recompiling *n* files at a time. This may improve overall build time on machines with multiple cores.

Customizing WPO

You can use the logical pairs of the default WPO options (such as -Xno-wpo-inline) to disable the defaults on an individual basis (see WPO Compiler Options on page 130).

You can prevent whole program optimizations from being performed on a specific variable or function by marking it as **volatile** (in the case of a function, mark the return value as **volatile**). Conversely, if expected optimizations are not being carried out for a particular variable or function, make sure it does not fall into one of the categories described as unsafe (see the *Wind River DIAB Compiler Options Reference*).

To prevent inlining of a particular function, use **__attribute__((noinline))** to prevent both normal and inter-module inlining.

You can prevent every function in a module from being inlined across modules by either:

- Not compiling that module with -Xwhole-program-optim at all (or equivalently, compiling it with -Xwhole-program-optim=0).
- Compiling it with -Xwhole-program-optim=0x5 (that is, **0x1 | 0x4**). The **0x4** mask means import only, which inlines functions from other modules but does not export functions from the module being compiled.

WPO Treatment of Static Functions and Variables

Since all non-local symbols are exposed to optimizations during WPO, a file static has to be mangled so that no two statics from different modules with the same name will clash in the name space from the point of view of WPO. This is why all statics are mangled, externalized and become global. Static functions and variables are marked with **__STF** and **__STV** prefixes respectively. For architectures where symbols are prefixed automatically with underscores, these prefixes will have an extra underscore.

WPO Usage Caveats

The following caveats and suggestions apply to using WPO.

- You can freely mix object files compiled with -Xwhole-program-optim with object files compiled without that option on the link line. However, only the former will participate in whole-program optimization.
- Any compiler optimization options (such as -XO) specified in conjunction with -Xwhole-program-optim usually apply at link time rather than compile time. The assumption is that you will be linking with -Xwhole-program-optim so that all optimization will take place at link time.

  However, if you need the original object files to be optimized (for example while building libraries), add 2 to your -Xwhole-program-optim mask.
- Object files compiled with -Xwhole-program-optim in one release of the compiler may not be optimizable by a later version of the linker. However this will not be a fatal error; the linker will simply produce a warning and any such object files will be linked directly instead of being recompiled and linked.
- When a project produces multiple executables (possibly sharing some of the same object files), each link line must use a different value of -Xwhole-program-output-dir. This is especially important if caching (-Xwhole-program-optim flag 8) is enabled, because, due to the aggressive optimizations employed by WPO, the artifacts generated for one link cannot be reused to produce a different executable.

WPO Examples

The following examples show how whole-program optimization may be used.

In this case four program files are compiled with a default WPO setting. Compilation is halted (with -c) after the object files are created.

```
$ dcc -c -XO -Xwhole-program-optim wp-1.c wp-2.c wp-3.c
                    wp-main.c wp-1.c: wp-2.c: wp-3.c: wp-main.c:
```

The files are now linked to produce the executable **wp_test**. The linker generates optimized copies of the files and links those. The files are stored in the specified output directory (**ocache**); the cache is enabled (by adding 0x8 to the -Xwhole-program-optim mask); and WPO diagnostics are produced. Note that the -XO option specified previously at compile time will be used for link-time optimization.

```
$ dcc -o wp_test \ -Xwhole-program-optim=9
                    -Xwhole-program-diagnostics=2 \ -Wl,-Xwhole-program-jobs=4 \
                    -Wl,-Xwhole-program-output-dir=ocache wp-1.o wp-2.o wp-3.o wp-main.o
                @@ Recompiling: wp-1.o @@ Recompiling: wp-2.o @@
                    Recompiling: wp-3.o @@ Recompiling: wp-main.o @@ Relinking
                ...
```

wp-2.c is then recompiled and a new **wp-2.o** module produced. No diagnostics are produced; -XO is again specified:

```
$ dcc -c -XO -Xwhole-program-optim wp-2.c
                wp-2.c:
```

Finally, all three files are compiled again the same way they were in the second example. Since **wp-1.o** uses optimizations contained in the recompiled **wp-2.o**, it is automatically recompiled by the linker:

```
$ dcc -o wp_test \ -Xwhole-program-optim=9
                    -Xwhole-program-diagnostics=2 \ -Wl,-Xwhole-program-jobs=4 \
                    -Wl,-Xwhole-program-output-dir=ocache wp-1.o wp-2.o wp-3.o wp-main.o
                @@ Recompiling: wp-1.o @@ Recompiling: wp-2.o @@
                    Relinking ...
```

14.4. Code Factoring

Code factoring is a link-time whole program size optimization that looks for functions that end with the same sequence of instructions ("common tail") and replaces the end of one function with a jump to the same code sequence in another function. This reduces the size of the code at the cost of some performance due to the extra branches. Refer to the documentation of -Xcode-factor Compiler X Option in the *Option Reference* for details.

Code factoring works during the final link and it always operates across the entire program, regardless of whether WPO is activated. It is not connected to WPO, although the exact factoring that occurs may depend on whether WPO is enabled, because the optimized code will be different in those two cases. In case of WPO, code factoring will be applied to the link of the recompiled object files.

14.5. SDA Optimization

SDA optimization attempts to maximize small data area (SDA) use by moving variables from non-SDAs into SDAs.

SDA Optimization Operation

SDA optimization is an optional part of whole-program optimization (WPO; see Whole-Program Optimization on page 130). It is not, however, enabled by default when WPO is enabled.

SDA optimization is enabled by setting bit **0x100** in the -Xwhole-program-optim flag. For example use:

```
-Xwhole-program-optim=0x101
```

SDA optimization can be disabled for particular classes of small data area with the -Xdisable-SDA option (see the *Wind River Diab Compiler Options Reference*.

Note: The SDA optimization is incompatible with the link-time object-file cache (mask bit **0x8**), and is automatically disabled if the cache is enabled.

With SDA optimization, the variables moved to specific SDAs are taken from a so-called *candidate* section, and are chosen based on static variable usage analysis. SDA optimization attempts to move the most frequently used variables into SDAs, provided the SDA has enough free space. The pairs of candidate sections and SDAs are as follows:

Candidate	Target SDA
.text	**.sdata2**
.data	**.sdata**

Note: Both candidate and target SDA are output section names. That is, they refer to the sections in the final ELF file.

If you do not want a variable that would normally belong to one of the candidate sections to be moved, you must ensure that this variable ends up in a non-candidate section. You can do so, for example, by using the linker command file or **use_section** pragmas.

14.6. Target-Independent Optimizations

The compiler provides a set of optimizations that can be used with all target architectures.

Disabling Optimizations

It is possible to disable some of the target-independent optimizations described below with the **-Xkill-opt** option (see the *Wind River Diab Compiler Options Reference*).

CAUTION: The -Xkill-opt option is reserved for internal Wind River use. It should be used only on the advice of Wind River Customer Support.

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.

Various Optimizations

These include:

- several registerization optimizations designed to reduce read and write operations, including: registerization of memory references across loops; registerization of function memory references; and registerization of loop-invariant loads
- loop unswitching (loop splitting)
- tail merging (merges common code out of **if**/**then**/**else** and **switch** statements)
- optimization for the __**builtin_expect** intrinsic function (See Intrinsic Functions on page 78)

Tail Recursion

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

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. For the definition of *node* see the -Xunroll entry in the *Wind River Diab Compiler Options Reference*. Assembly files saved with -S show the number of nodes for each function.

Note: Inlining will increase compile-time memory usage, depending on the size and complexity of the source program.

In some cases, an out-of-line copy is made of an inlined function. This is necessary if the function is declared to be **extern** or if a reference is made to the address of a **static** function.

To be inlined, the called function must be in the same file as the calling function.

Inlining can be triggered in the following ways:

- 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* in Keywords Reference on page 68.
- 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 Pragmas on page 57.

- Use option -XO to automatically inline functions of up to the number of nodes set by -Xinline (see the *Wind River Diab Compiler Options Reference*). Option -XO sets this value to 40 nodes by default.

In addition to **-Xinline**, the options -Xexplicit-inline-factor, -Xinline-explicit-force, and -Xwhole-program-optim also control inlining of functions.

Note: By default, code must be optimized by use of the -XO or -O option for inlining to occur. To force a function to be inlined without using -XO or -O, see Attribute Specifiers Reference on page 73.

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;
        }
        ...
}
```

Note: Some functions should not be inlined because they may depend on values that are set as part of the function call, such as the contents of the link register. To prevent a function from being inlined, add the following attribute:

__attribute__ ((noinline))

See Attribute Specifiers Reference on page 73 for details. For information about disabling inlining for any function containing assembly code, see the -Xinline-asm-off entry in the *Wind River Diab Compiler Options Reference*.

Argument Address Optimization

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

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

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 they are available on the target processor:

```
p++;                    ->
p[0] = 0;                      *++p = 0;
p++;
p[0] = 1;                      *++p = 1;
```

Increments are moved from the end of a loop to the beginning in order to use incrementing addressing modes--when they are available on the target processor:

```
while(*s++) ;           ->     s--; while(*++s) ;
```

Tail Call Optimization

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 option is used.

Note: In earlier releases (prior to version 4.3), the 0x100 mask was used to disable simple branch optimization.

Common Tail Optimization

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.

Variable Live Range Optimization

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

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

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

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

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

Small loops are unrolled to reduce the loop overhead and increase opportunities for rescheduling. -Xunroll option sets the number of times the loop should be unrolled and defines the maximum size of loops allowed to be unrolled. (See also the entries for -Xunroll and -Xunroll-size in the *Wind River Diab Compiler Options Reference*.)

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

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

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

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;
}
```

Minor Transformations to Simplify Code Generation

Some minor transformations are performed to ease recognition in the code generator:

```
if (a) return 1;       ->     return a ? 1 : 0;
return 0;
```

Register Coloring

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(j) - j;
        }
}
```

a and **j** use the same register.

Inter-procedural Optimizations

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

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

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 on page 58).

Extend Optimization

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

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

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

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

This optimization creates a Local Data Area (LDA) into which variables may be placed for fast, efficient base-offset addressing. See Local Data Area Optimization and -Xlocal-data-area on page 165 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.

CAUTION: LDA is incompatible with split data (-Xsection-split=0x02), because LDA must place variables into a contiguous area, and therefore into the same section.

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 Profiling in an Embedded Environment on page 185.

14.7. Example of Optimizations With bubble.c

Compiling the **bubble.c** program provides an example of several of the optimizations generated by the compiler, and how they interact with each other.

Compiler Options

Compilation for this example uses the following command and options:

```
    dcc -tV850ES:windiss -Xpass-source -O -Xinline=40  -Xsize-opt -S bubble.c
```

Note that the option -Xpass-source causes the source to be included intermixed as comments with the assembly code in **bubble.s**.

Code

The **bubble.c** program implements sorting 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;
}
```

Only the bubble( ) function is shown; however, 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. The numbers in the **Explanation** column of the table in Illustration of Optimizations on page 146 are associated with optimizations as follows:

Implemented Optimizations

The following optimizations are implemented in the code generated for **bubble.c**. The numbers appear at the appropriate location in the explanation of the generated assembly code in the table in Illustration of Optimizations on page 146.

(1)	remove entry and exit code
(2)	use scratch registers for variables
(3)	unused assignment deletion
(4)	complex branch optimization
(5)	loop strength reduction
(6)	loop count-down optimization
(7)	global common subexpression elimination
(8)	inlining of functions
(9)	constant and variable propagation

Illustration of Optimizations

Table 1. Illustration of Optimizations

C Code	Generated Assembly Code			Explanation
		.text		
		.align	1	
		.global	_bubble	
{ int change = 1; int i;	_bubble:			Start of **bubble( )** function. No entry code is necessary (1) since all variables are put in scratch registers (2) and the link register **LR** is not used. The stack pointer is not used and does not need to be adjusted. The assignment **change = 1** is eliminated (3) since it is used only in the first while test, which is known to be true and removed (4).
int test_count = 0;		mov	0,r13	**test_count = 0;** Use **r13** for **test_count**.
	.L3			Top of **while(change)** loop.
count--;		add	-1,r7	Decrement **count** by 1.
while (change) {		cmp ble	0,r7 .L14	Initial check if **for**-loop needs to be skipped (in case of **r7=count==0**).
change = 0;		mov	0,r11	**change = 0;**
for (i = 0; i < count; i++) {		mov mov	r6,r10 r7,r12	Initialization of the **for** loop. Loop strength reduction (5) has

C Code	Generated Assembly Code			Explanation
				replaced all references to **base[i]** with a created pointer, **$$2**, placed in register **r10**. Since no more references are made to **i**, loop count-down optimization (6) decrements **i** (**r12**) from count (**r7**) to **0**.
	`.L15`			Top of **for** loop.
`test_count++;`		`add`	`1,r13`	Increment **test_count**
		`ld.w`	`0[r10],r9`	**$$2[0]** is loaded to **r9**. Since this value is used later on, it is remembered in **$$4** (7).
		`ld.w`	`4[r10],r8`	**$$2[1]** is remembered in **r8** (**$$3**) (7).
`if (base[i]>base[i+1]) {`		`cmp` `ble`	`r8,r9` `.L8`	**if** statement. See if a swap must take place. If not, branch to **.L8**.
`swap2(&base[i]);`				The function **swap2( )** is inlined (8). Variable propagation (9) removes the use of variables **tmp** and **ip**.
		`st.w`	`r8,0[r10]`	**ip[0]=ip[1](=$$3);**
		`st.w`	`r9,4[r10]`	**ip[1]=tmp (=$$4);**
`change = 1;`		`mov`	`1,r11`	Set **change** to 1.
`}`	`.L8`			

C Code	Generated Assembly Code			Explanation
		`add`	`4,r10`	**$$2** is incremented (5).
		`add`	`-1, r12`	**i** is decremented (6).
`}`		`cmp`	`0,r12`	**i** is compared with 0.
		`bne`	`.L15`	Branch back to top of **for** loop in case **i** is not yet 0.
`}`		`cmp`	`0,r11`	Compare **change** with 0.
		`bne`	`.L3`	Branch back to top of **while** loop in case **change** is 1.
	`.L14`			
`return test_count;`		`mov`	`r13,r10`	Move **test_count** to return value register.
		`jmp`	`[31]`	Jump back to what the link register points to.
`}`				

Note: The assembly code described above is provided as an example, and may not exactly match the code that your version of the compiler produces from **bubble.c**.

15. SAFETY AND SECURITY

15.1. Stack Smashing Protection

Stack smashing protection (SSP) is a security feature of the Diab compiler that, when enabled, attempts to detect stack-based buffer overflow attacks at runtime. If an attack is detected, a user defined fatal error hook is called.

SSP does not prevent buffer overflows, rather it attempts to detect attacks after they have occurred, but before the attacker is able to take advantage of the corrupted stack to execute arbitrary code.

SSP is not a form of "bounds checking" and is not intended to be used as a way to catch programming errors, though in some cases it may do so.

SSP does not detect overflows of heap based buffers.

SSP is intended to be used as one layer of protection against malicious attack. It is not a substitute for security conscious design, code audits, and so on, but rather, a final defense that attempts to minimize the impact of a successful stack-based buffer overflow.

15.2. Overview

SSP uses two protection mechanisms, canary and stack reordering.

The canary is a stack slot that lies between the local variable area and the return address. When a function is protected by SSP, the canary is initialized to a random value determined by the global variable **__stack_chk_guard** in the function prolog, and then checked against the same value in the function epilog. If the canary's value has changed, the fatal error hook **__stack_chk_fail** is called.

Stack reordering attempts to place arrays, which are vulnerable to overflow, closer to the canary than other variables, in particular pointers, that must be protected from corruption. This protects pointers and other scalar variables (such a loop indexes) from being corrupted in the event of a successful buffer overflow.

Note: unless they are marked volatile, or have their address taken, local variables will usually be in registers, and so will not be directly vulnerable to corruption as a result of buffer overflow.

In some cases it might be impossible for the compiler to separate arrays from other stack variables. This can happen if there are structs containing arrays and non-arrays. For example if a local variable has the following type:

```
struct
{
    char a[8];
    int * p;
};
```

then there is no way for the compiler to reorder local variables in such a way that the pointer p is protected from overflow of array a. If such a case is detected, the compiler will give the following warning:

```
warning: SSP - in function '<func>', variable '<var>' is still vulnerable
```

```
   after stack reordering because of its struct layout
```

15.3. Stack Smashing Example

Here is a highly simplified “toy” example that illustrates the problem that SSP attempts to solve.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

   __attribute__((noinline))
   void unsafeFunction(const char* untrustedData, int untrustedSize)
   {
           char A[8];
           memcpy(A, untrustedData, untrustedSize);
           // A real application would now do something with array 'a'
   }
   void bad(void)
   {
           printf("ERROR: ATTACK SUCCEEDED!\n");
           exit(1);
   }
   typedef void (*FN)(void);
   FN attackData[] = {bad, bad, bad, bad, bad, bad};
   int main()
   {
           // Simulate processing data that has come from an untrusted source
           unsafeFunction((const char*)attackData, sizeof(attackData));
           return 0;
   }
```

Here *unsafeFunction* represents an application function that handles data from an external source without first checking the size of that data. One way this can occur in a real application is when strcpy is used to copy untrusted data to a local buffer.

This flaw allows an attacker to inject arbitrary data into the local buffer A.

Stack frame before attack:

Return Address	Correct return address
...	
A[5:8]	
A[0:4]	

Stack frame after attack:

Return Address	bad
...	bad
A[5:8]	bad
A[0:4]	bad

Note that the return address has been corrupted by the attack. When the function returns, instead of returning to the caller, it will return to an address of the attacker's choosing.

If the above example is compiled without SSP enabled, the following output will be produced:

```
ERROR: ATTACK SUCCEEDED!
```

Without SSP enabled, the attacker was able to execute code of their choice (in this case, the function **bad**).

When SSP is enabled, the stack frame changes as follows:

Stack frame before attack:

Return Address	Correct return address
...	
Canary	Value of __stack_chk_guard
A[5:8]	
A[0:4]	

Stack frame after attack:

Return Address	bad
...	bad
Canary	bad
A[5:8]	bad
A[0:4]	bad

Note that the attacker was still able to corrupt the stack frame. However in this case, before the function returns, it will check the canary against the value of **__stack_chk_guard**. It will detect that the canary has been corrupted and call **__stack_chk_fail**.

If the example is compiled with **-Xstack-protection**, the following output will be produced (with the default implementation of **__stack_chk_fail)**:

```
Error: Stack Smashing attempt detected!
abnormal process termination
```

With SSP enabled, the attack was detected before the attacker was able to execute arbitrary code.

15.4. Levels of Protection

Because adding a canary check adds both footprint and performance overhead, it may not be reasonable to add protection to every function. Diab offers four levels of protection.

Note that these levels only differ in which functions are protected, not in how they are protected.

- **-Xstack-protection-off** – no protection (this is the default)
- **-Xstack-protection** - protects functions that have local char arrays (possibly inside structs), or that use alloca
- **-Xstack-protection-strong** - protects functions that have local arrays of any type (possibly inside structs), or local variables that have their address taken, or that use alloca
- **-Xstack-protection-all** - protects all functions

In addition, protection can be specified on a function by function basis, using the following attribute:

```
__attribute__((stack_protect(n)))
```

For example:

```
__attribute__((stack_protect)) void protectedFunction() { … }
__attribute__((stack_protect(1))) void anotherProtectedFunction() { … }
__attribute__((stack_protect(0))) void unprotectedFunction() { … }
```

enables SSP on **protectedFunction**, and **anotherProtectedFunction**, and disables SSP on **unprotectedFunction**.

Note: The attribute always take precedence over the command line option.

Note: **alloca** should not be used in functions that may handle untrusted data. When SSP is enabled, the compiler will give the following warning if a function uses **alloca**:

```
warning: SSP - alloca() used in function '<func>'
```

15.5. Diagnostic

To see which functions are protected by SSP, use the command line option **-Xstack-protection-verbose**.

About This Task

This command will provide output like the following:

```
info: SSP enabled on '<func>' because <reason>
```

For example:

```
 info: SSP enabled on ‘foo’ because has char array
```

15.6. User Responsibilities

The user's role in ensuring safety and security.

__stack_chk_guard

In order for SSP to be effective, it is essential that an attacker not be able to guess the value of **__stack_chk_guard**. It is recommended that this variable be initialized to a different random value every time the application starts up (or more frequently for long-running applications).

Here is a sketch of how this might be done;

```
#include <stdint.h>
 uintptr_t __stack_chk_guard;
 int main()
 {
     get_random_value (&__stack_chk_guard, sizeof(__stack_chk_guard));
 ...
 }
```

where *get_random_value* is a hypothetical device specific function (not provided in the Diab standalone environment) that uses hardware sources of randomness to initialize a range of bytes to random bits.

__stack_chk_fail

Although a default version of the fatal error hook **__stack_chk_fail** is provided in the Diab library, users should always override this function with an application specific implementation. When implementing this function, please consider the following:

- This function will only be called after an attacker has already successfully modified data on the stack
- No stack data should be trusted
- Only very low level functions that are guaranteed to not be using compromised data should be called
- Under no circumstances should this function return!

The default implementation of this function is as follows:

```
__attribute__((noreturn)) void __stack_chk_fail(void)
```

```
    {
            write(2, "Error: Stack Smashing attempt detected!\n", 40);
            abort();
    }
```

Non-executable stack

Although not strictly required for SSP to be effective, it is strongly recommended that customers use the memory protection unit of their processor to mark the stack as non-executable. In the event of an attack that is not caught by SSP, this protection measure makes it harder for attackers to run arbitrary code.

16. CODE AND DATA LOCATION AND OPTIMIZATION

16.1. Code and Data Location in Memory: section and use_section Pragmas

The **section** and **section_use** pragmas are used to locate code and data in non-default memory areas.

By default, the compiler generates architecture-specific code for locating and accessing code and data in memory which will be suitable for many cases. 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 with default attributes.

The **section** pragma can be used to change default attributes of sections, to define new sections, and to control the assignment of code and variables to particular sections; as well as to control the access mode.

The **use_section** pragma can be used to assign any variable or function individually to a predefined section class, or to a user-defined section class.

C++ Compiler Limitations

The C++ compiler has the following limitations for **section** and **use_section** pragmas:

- 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 the *Wind River Diab Compiler Options Reference* for more information about 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()
{
}
```

Related Information

- section and use_section Pragma Syntax on page 155
- section and use_section Pragma Behavior and Usage on page 157

16.2. Section and use_section Pragma Syntax

Short reference description.

section Pragma Syntax

```
#pragma section class_name  [istring]  [ustring]  [addr-mode]  [acc-mode]  [address=x]
```

class_name

Required. Symbolic name for a predefined or user-defined section class to hold objects of a particular *class* (code, initialized variables, or uninitialized variables).

A section *class_name* is used only in writing **#pragma section** and **#pragma use_section** directives.

> Note: To change the name of a default section, you use one of the -Xname- options (see the *Wind River Diab Compiler Options Reference*).

istring

Name of the actual section to contain initialized data. For variables, this means those declared with an initializer (for example, **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 (for example, **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.

acc-mode

Accessibility to the section.

address=*x*

Specific address at which to place variables and functions. The address "x" should point to a free memory location.

#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 namespace scope variables, class static member variables, global and namespace scope functions, and class member functions that follow the pragma.

use_section Pragma Syntax

```
#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* (code, initialized variables, or uninitialized variables).

A section *class_name* is used only in writing **#pragma section** and **#pragma use_section** directives.

Related Information

- Code and Data Location in Memory: section and use_section Pragmas on page 155
- Section Classes and Their Default Attributes on page 160
- Pragma Addressing Mode for Functions, Variables, Strings on page
- section Pragma Access Mode: Read, Write, Execute on page 163

16.3. Section and use_section Pragma Behavior and Usage

The **section** and **section_use** pragmas are used to locate code and data in non-default memory areas. Their behavior is governed by a set of defaults and a defined order of precedence. They can be used to specify sections for initialized and uninitialized variables, to locate variables and functions at absolute addresses, and so on.

Initialized and Unitialized Sections

When defining a data section class to hold variables, the **section** pragma can specify two sections: one for initialized variables and one for uninitialized variables, or either section by itself. These physical sections appear in the object file and may be manipulated during linking.

- The *istring* parameter is the name of actual section 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. For example:

  ```
  ".text", ".data", ".init"
  ```

- The *ustring* parameter is the name of actual section 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. Generally, **COMM** sections are gathered together by the linker and placed at the end of the **.bss** output section. For example:

  ```
  ".bss", ".data", "COMM"
  ```

Consider the following:

```
#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 */
    int 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.

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 and how large they are. Each predefined section class is defined by default values for all of its attributes.

Default Class Values

If a **section** pragma for a class is used with no values for one or more of the attributes, those attributes are always restored to their default values. This is true even for a user-defined *class_name* (the table shows the default attributes in this case as well).

Precedence for Multiple section Pragma Directives

A **#pragma section** directive for the section class (*class_name*), applies to all subsequent declarations and definitions until the next **#pragma section** directive for the same section class is reached.

If a function or variable has multiple declarations within a single translation unit (in this context a definition counts as a declaration), the section attributes for the function or variable are updated based on whether the -Xpragma-section-first or -Xpragma-section-last option is used.

If the -Xpragma-section-first option is used (the default), the first declaration or definition is used (for both functions and variables).

For functions, for example:

```
int function();

#pragma
section CODE ".code"
int function() {return 0;}
int
function2() {return 0;}
int function3();

#pragma
section CODE ".code2"
int function2();

#pragma
section CODE ".code3"
int function3() {return 0;}
```

Here, function( ) goes into **.text**, while function2( ) and function3( ) both go into **.code**.

For variables, the section attributes are set at the definition that is seen first, but can be overridden by an initialization. For example:

```
extern int var[100];

#pragma
section DATA ".initialized_data2" ".uninitialized_data2"
int var[100];
extern int var2[100];

#pragma
section DATA ".initialized_data3" ".uninitialized_data3"
```

```
int var[100] = {1}; /* initialization */

#pragma
section DATA ".initialized_data4" ".uninitialized_data4"
int var[100];
int var2[100]; /* tentative definition, not initialization */
```

Here, *var* goes into **.initialized_data3**, but *var2* goes into **.uninitialized_data2**.

If the -Xpragma-section-last option is used (non-default), the last declaration or definition is used (for both functions and variables). If the preceding examples were compiled with -Xpragma-section-last, the following would occur:

- function( ) goes into **.code**, function2( ) goes into **.code2**, and function3( ) goes into **.code3**.
- *var* goes into **.initialized_data4** and *var2* goes into **.uninitialized_data4**.

Sections with the Same Name

You may create initialized and uninitialized sections with the same name; however, this practice is discouraged. Consider the following example. You create two sections called **.my_sect**:

```
#pragma section DATA  ".my_sect"             /* initialized */
#pragma section DATA  ".my_sect" ".my_sect"  /* uninitialized */
```

In your linker command file, you specify **.my_sect** as uninitialized with the **BSS** notation:

```
.my_sect (BSS) : {} > out_sect
```

In this case, the linker will ignore the **BSS** specification, and, without issuing a warning, treat **.my_sect** as initialized data. It does so because uninitialized sections are normally cleared out at startup; since you're mixing uninitialized and initialized data, this would mean that data might be lost. (You can force the linker to recognize the **BSS** specification by using **=BSS** instead.)

Using the section Pragma Address Option to Locate Variables and Functions at Absolute Addresses

The **section** pragma **address**=*n* option provides a way to place variables and functions at a specific absolute address in memory. With this form, the linker puts 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.

The advantages of using absolute sections include:

- 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
                                //   effect on placement of "fun"
int 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.

Related Information

- Code and Data Location in Memory: section and use_section Pragmas on page 155
- section and use_section Pragma Syntax on page 155
- Section Classes and Their Default Attributes on page 160
- Pragma Addressing Mode for Functions, Variables, Strings on page
- section Pragma Access Mode: Read, Write, Execute on page 163

16.4. Section Classes and Their Default Attributes

Each section class has a set of default attributes that can be modified with the **section** and **use_section** pragmas.

Description of Section Classes and Their Default Attributes

The default attributes specify the section for initialized data, the section for unitilized data, the address-mode, and the access-mode.

Table 1. Section Classes and Their Default Attributes

Section		**Default**			
class_name	**Description and Example**	*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;` Size in bytes - Xsmall-data.	**.data**	**COMM**	**far-absolute**	**RW**
SDATA	Variables, size in bytes <= -Xsmall-data: `static int i ;`	**.sdata**	**.sbss**		**RW**
CONST	**const** variables `const int a[10] = {1, ...};` Size in bytes > -Xsmall-const	**.text**	n/a	**far-absolute**	**R**
SCONST	**const** variables, size in bytes <= -Xsmall-const: `const int ic = 53;`		n/a	**near-code**	**R**
STRING	string constants: `"hello\n"`	**.text**	n/a	**far-absolute**	**R**

Section		**Default**			
class_name	**Description and Example**	*istring*	*ustring*	*addr-mode*	*acc-mode*
user-defined	`#pragma section USER ...`	**.data**	**COMM**	**far-absolute**	**RW**

Section Classes and Their Default Attributes

Default Values for Section Names

The section names shown in the table assume the default value for option -Xconst-in-text.

Dynamically Initialized C++ Constant Variables

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.

Small Data and Small Constants

Small **data** and **const**: both -Xsmall-data and -Xsmall-const are set to 8 by default.

Local Data Optimization

Global and static scalar variables may be placed in a *local data area* if the following are true:

- Optimization is in effect (either -O or -XO is present).
- -Xsmall-data or -Xsmall-const is zero.
- -Xlocal-data-area is non-zero.

The default for -Xlocal-data-area is 32,767 bytes.

The local data area is 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.

Small Data Area (SDA) Sections: Including and Excluding Data

On occasion, you may discover that data that you intend to go into Small Data Area sections does not wind up there as expected, even if you have used, for example, a section pragma or **__attribute__((section))** to explicitly place data in an SDA section.

One of the ways this can happen is if you declare a variable before defining its type. For example, in this case the variable *bar* will be inserted into an SDA section, as expected:

```
struct foo; struct foo /* type defined before variable
                declared */ { short x; } extern struct foo bar; /* bar declared */
```

Whereas in this example, *bar* will not be placed in an SDA section:

```
struct foo; extern struct foo bar; /* variable declared
                before type defined */ struct foo /* type declared */ { short x; }
```

In this second example, because *bar* is declared at a place where its type is still incomplete, the compiler cannot make any assumptions about its size, and so it must conservatively assign it a section that requires far addressing.

On the other hand, it is possible to intentionally force linker command file symbols into non-SDA sections by declaring them as array types of unspecified size in the source code. For example:

```
extern char __BSS_START[], __BSS_END[];
```

Since the array size is unknown, the compiler treats such symbols, which are defined in the linker command file, as non-SDA. See the *Wind River Diab Linker Reference*.

Related Links

- section and use_section Pragma Syntax on page 155
- Initialized Data Location in Text or Data With -Xconst-in-text on page 164
- Local Data Area Optimization and -Xlocal-data-area on page 165

16.5. Section Pragma Access Mode: Read, Write, Execute

The **section** pragma's *acc-mode* option defines how the section can be accessed.

Access Mode Options

The *acc-mode* option can be specified as 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++. The compiler uses "**O**" to mark sections generated for templates as COMDAT; the linker then collapses identical instantiations into a single instance.

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.

If -Xconst-in-text=0 then the **CONST**, **SCONS T**, and **STRING** section classes have will have access mode **RW** (read/write) rather than the default **R** (read only).

Multiple instances of a constant allocated to a section with no write access (**W**) may be collapsed by the compiler to a single instance.

Related Links

- section and use_section Pragma Syntax on page 155
- Section Classes and Their Default Attributes on page 160
- Initialized Data Location in Text or Data With -Xconst-in-text on page 164

16.6. Initialized Data Location in Text or Data With -Xconst-in-text

Locate initialized data in "text" or "data" sections with -Xconst-in-text.

Data Location With -Xconst-in-text

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 locating section class data in text or data sections based on a mask setting. The syntax is as follows:

```
-Xconst-in-text=mask
```

where the mask bit is set to 1 to locate the data in a text section, or cleared to locate the data in a data section. The masks are as follows:

0x1

Controls **const** variables in the **CONST** section class.

0x2

Controls small **const** variables in the **SCONST** section class.

0x4

Controls string data in the **STRING** section class.

By default, the mask for -Xconst-in-text is **0x7.**

Table 1. RH850 Section Classes and Mask Bits

Section Class	Mask Bit	"text" Section with Mask Bit Set to 1	"data" Section with Mask Bit Set to 0
CONST	0x1	**.text** (default)	**.data**
SCONST	0x2	**.sdata2** (default)	**.sdata**
STRING	0x4	**.text** (default)	**.data**

Note: 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). If a section is moved from its default by -Xconst-in-text, this will be a change from its usual default access mode.

Note: The **.sdata2** section is placed with the **.text** section by the default linker command file (perhaps to be located in ROM), and so is considered to be a "text" section.

For example, -Xconst-in-text=3 means that initialized **const** variables and small **const** variables should be placed in their usual default "text" sections, **.text** and **.sdata2**, respectively, while strings should be placed in the **.data** section rather than their usual **.text** section.

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 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.

Related Links

- section Pragma Access Mode: Read, Write, Execute on page 163

16.7. Local Data Area Optimization and -Xlocal-data-area

The compiler supports local data area (LDA) optimization with the -Xlocal-data-area and -Xlocal-data-area-static-only options.

-Xlocal-data-area Optimization Operations

The optimization provided by -Xlocal-data-area works as follows:

- The LDA optimization applies only to static and global variables of scalar types--not arrays, structures, unions, or classes (for C++).
- The LDA optimization applies only to scalar variables not assigned to the small data or small **const** areas. This is the case if you compile for no small data for F targets (as in RH850FN): where -Xsmall-data=0 and -Xsmall-const=0. For E targets (as in RH850EN) and for G targets (as in RH850GN) -Xsmall-data=8 and -Xsmall-const=0 by default.
- An LDA is allocated for each module, and static and global scalar variables that 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. Local Data Area optimization is incompatible with -Xsection-split=0x02.
- 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.

(Or the **.sbss** section for small uninitialized variables if -Xsmall-data is > 0.)

But with LDA optimization, variables to be put into the LDA are put there whether initialized or not; and if any LDA variables are initialized, 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 32,767 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.

16.8. Small Data Areas and Small Constant Areas

For certain architectures, "small" data and constant areas use pre-defined sections that can optionally be created by the compiler to improve reference efficiency for widely used static or public variables.

See also Code Generation for Different Addressing Modes.

Small Data and Small Constants

Code to reference variables in these areas is smaller because each area uses a "near" 16-bit addressing mode.

Note: Small Data Area is not available for PPC64.

The linker defines symbols for the base address of the small data area or areas.

The startup module **crt0.s** loads the appropriate value into the base register. See the appropriate startup module for a given architecture.

Small Data Area and Small Constant Area Creation

The small data and constant areas are created using the -Xsmall-data=*n* and -Xsmall-const=*n* options:

- If -Xsmall-data=*n* is present and *n* > 0, then non- **const** variables of size in bytes <=*n* will be placed in the pre-defined **SDATA** section class rather than the default **DATA** section class.
- If -Xsmall-const=*n* is present and *n* > 0, then **const** variables of size in bytes <=*n* will be placed in the pre-defined **SCONST** section class rather than the default **CONST** section class.

Note: Symbols are assigned to the small data area at compile time. The linker determines the final address and performs the necessary relocation. If a given symbol should not be assigned to the small data area, override the default with **#pragma use_section**.

For example, if a symbol is assigned an absolute address at link time, force it to the **DATA** section class with the following:

```
    #pragma use_section DATA symbol_name
```

To use the small data area, the linker command file should allocate the initialized and uninitialized small data sections together. The base address for the small data area is the address of the start of this combined section + 0x7ff0.

All of the small data is then accessible with a 16-bit offset from the base address. The relative sizes of the two sections does not matter as long as their combined size is not larger than 64 KB - 0x10 bytes.

The compiler uses base + offset addressing mode to access objects in **SDATA** and **SCONST** section classes. The compiler uses r4(gp) as the default base register to access objects in SDATA section and the compiler uses r5(tp) as the default base register to access objects in **SCONST** sections. The offset of an object is later determined by the linker. The linker will generate an error if the offset exceeds the max limit allowed for the compiler generated instruction. For example: for 16-bit addressing mode, the linker will generate an error to access an object whose offset is > 32767 or < -32768.

With Diab 5.9.7.0, a new feature to access **SDATA or SCONST** objects beyound the permissible default offsets is available. The compiler option -Xsmall-data-registers=<n> can be used to specify additional base registers for small data and constants. These additional base registers on RH850 are **r20**, **r21**, **r22**, **r23** and **r24**.

The default value for **-Xsmall-data-registers** is 1 (even if the option is not explicitly used.) which means that only **r4** and **r5** are available for small data and small constants respectively. -Xsmall-data-registers=2 means **r20** will be used as an additional base register, while -Xsmall-data-register=3 means **r20**, r21 are available as additional base registers as required. The max value of -Xsmall-data-register is therefore **6** for RH850.

Any additional base register available will be used either by the small data or small constants as needed. Consider the size of small data is 120KB and small constants as 60KB: In this case the additional base register **r20** will be used for small data. Consider another example where the size of small data is 60 KB while the small constants are 120 KB. In this case **r20** will be used for small constants. If both small data and constants are 120 KB each, then we need 2 additional registers **r20** and **r21** and that can be done by using -Xsmall-data-registers=3 option.

The linker throws an error if the number of additional base registers were less than required. For example:

```
Error: Insufficient additional base registers. Required : 3, Specified: 2;
```

When any of the **r20-r24** is not used as an additional base register, the compiler is free to use it for any other purposes. That's why it is important that all files in a project are compiled with the same value of **-Xsmall-data-register** to have a consistent usage of these registers across the application. The linker warns if different files in a project were compiled with different values of -Xsmall-data-register option, in such cases it uses only the common subset of base registers found in all files. For example: if **file1.c** was compiled with **r20**, **r21** as additional bases, **file2.c** was compiled with **r20** as additional bases, **file3.c** was compiled as **r20**, **r21**, **r22** as additional bases, then the linker will only use **r20** as additional base register.

16.9. C++ Code and Data Memory Location

The C++ class layout follows the industry-standard C++ ABI. How and where the individual parts of C++ code are located in memory depends on the class element.

Class member functions (whether static or not) are located like any other function. That is, by default they are placed in the **CODE** section class. Placement can be controlled using the pragmas **section** and **use_section**, but the pragmas cannot take scope into consideration. That is, if there are functions with identical names, but in different classes or namespaces, there is no way to distinguish between the functions in pragmas, because pragmas do not differentiate between class or namespace prefixes for functions.

Static class member variables are located like any other variables. That is, by default they are placed in the **DATA**, **SDATA**, **CONST**, or **SCONST** section classes, depending on their type. Placement can be controlled using the pragmas **section** and **use_section**, but the pragmas, cannot take scope into consideration (as for functions).

Non-static class members are accumulated into the runtime size of the class instance footprint (as for ordinary C structures). The final size of a class instance footprint also depends on polymorphism and vtables. Such memory is either taken from the stack or heap, or from whatever section into which one places a class instance.

Any class containing virtual functions or derived from a class containing virtual functions gets an additional (pseudo) class member that is a pointer to the class vtable. The actual size of the vtable depends on how the class is used in the derivation tree of other classes. For more information, consult the C++ ABI. The location of the vtable defaults to the **CONST** section class, and is therefore affected by the -Xname-const option. If only the vtables should be placed in a special section, use the -Xname-vtbl option.

Type information tables are generated for each class and put into the **CONST** section class. The size of the type information depends on the class complexity. The location defaults to the **CONST** section class, and is hence affected by -Xname-const. If only the type information tables should be placed in a special section, use -Xname-rtti.

Type information names are generated for each class and put into the **CONST** or **SCONST** section classes, depending on the size of the name. Virtual table tables are generated as required and put into **CONST** or **SCONST** section classes. Both, type information names and virtual table tables, cannot be placed using pragmas, but they are affected by -Xname-const and -Xname-sconst options.

Destructors and constructors are placed in the **CODE** section class. Pragmas cannot be used to directly place an individual constructor or destructor, but the location is affected by changes to the definition of the **CODE** section class (for example, using -Xname-code).

17. EMBEDDED SYSTEMS DEVELOPMENT

17.1. Compiler Options and Pragmas for Embedded Development

The compiler provides a set of options and pragmas that control code generation in ways that are useful for embedded systems development.

Compiler Options

For more information about the following options, see the *Wind River Diab Compiler Options Reference*.

-Xaddr-*x*

Control addressing modes for data and code. See Pragma Addressing Mode for Functions, Variables, Strings on page .

-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.

-Xsize-opt

Minimize the size of the executable code.

-Xsmall-data, -Xsmall-const

Specify by size what data is to go into the small data area (SDA) and the small **const** area.

-Xstack-probe

-Xconst-in-text=0xf

Put strings and **const** data in the **.text** section together with code. See Initialized Data Location in Text or Data With -Xconst-in-text on page 164

-Xmember-max-align, -Xstruct-min-align

Options to pack structures in different ways. (Note that -Xstruct-min-align also affects unions.)

-Xtrace-table=0

Save space by not generating a back-trace table.

Pragmas

The following pragmas are useful for embedded systems development:

#pragma interrupt *func*

Specify that a function *func* is an exception handler. See the interrupt pragma in Pragmas on page 57.

#pragma pack

Control packing of structures and the byte order of members. See the pack pragma in Pragmas on page 57.

#pragma section...

Control placement and addressing of variables and functions. See the section pragma in Pragmas on page 57.

17.2. Startup and Termination Code

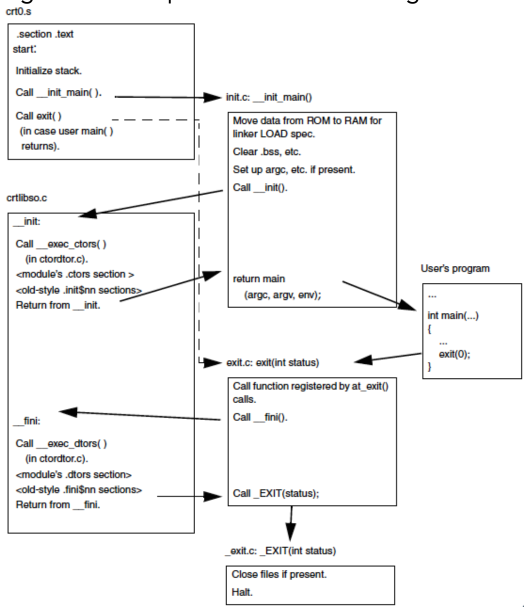
Default startup and termination for self-contained applications is provided with the compiler (applications that run under an operating system--such as VxWorks or Linux--work differently). The startup code for an embedded system must initialize the processor and run-time application, and the default functionality for both startup and termination may need to be modified for your target system.

Startup Code

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 _**exi t.c**.

The following overall schematic applies to all supported targets (and does not show some details). See the referenced modules for complete details.

Figure 1. Startup and Termination Program Flow



Location of Startup and Termination Sources and Objects

The source of **crt0.s** is located in an architecture-specific directory under **src**. Objects are in the library directories shown above.

The **init.c**, **crtlibso.c**, **exit.c**, and _**exit.c** files are in the **src** directory. Objects are in **libc.a**.

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 in a *linker command file*. For more information, see the discussion of linker command files in the *Wind River Diab Compiler Linker User's Guide*.

Insert assembly code as required to initialize the processor before **crt0.s** calls __init_main( ). 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.

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 1 on page 170.)

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 Run-time Initialization and Termination on page 174.

Note: Applications that use the malloc( )/free( ) routines supplied with the compiler must initialize the memory allocation library during program startup. Normally this is automatically handled by an __attribute__((constructor)) routine provided in the C library, which generates code in the **.ctors** section. (See Attribute Specifiers Reference on page 73 for information on the __attribute((constructor)) routine.) 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 the -Xdynamic-init entry in the *Wind River Diab Compiler Options Reference*.

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.

init.c: Copying Initial Values From ROM to RAM and 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), 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.

There are two options for copying initial values and initializing bss. You can either use a manual approach using five linker defined symbols, or you can use the more convenient copy table approach.

init.c requires five symbols to copy constants from ROM to RAM and to clear the **.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.

If you wanted to manually use the symbols (or other similar symbols that you defined in your linker command file), copying data could be done with:

```
memcpy(__DATA_RAM,__DATA_ROM,__DATA_END-__DATA_RAM);
```

And initialization of bss could be done with:

```
memset(__BSS_START,0,__BSS_END-__BSS_START);
```

As noted, these symbols are typically defined in a linker command file. The second option is to use copy tables. For more information, see the *Wind River Diab Compiler Linker User's Guide*.

init.c: 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 Program Arguments, Environment Variables, and Predefined Files on page 182 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.

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.

Stack Initialization

The initial stack is initialized by **crt0.s** to symbol **__SP_INIT**, typically defined in the linker command file. See Figure 1 on page 170, and for an example see the discussion of the linker command language in the *Wind River Diab Compiler Linker User's Guide*.

An additional symbol, **__SP_END**, is defined as the end of the stack in standard linker command files, as shown in the example. It is used in attempts to grow the heap and in stack checking.

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.

Note the following:

- Code compiled with -Xstack-probe 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.

Implementation Details for Stack Checking

Code is added to the prolog of each function (but see the note below) to determine whether the stack exceeds **__RTC_SP_LIMIT** (which is declared in **rtc.h** and initialized to **__SP_END**, and which is typically defined in the linker command file as shown in the **bubble.dld** example in the *Wind River Diab Compiler Linker User's Guide*).

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.

Dynamic Memory Allocation: 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 the discussion of command file structure in the *Wind River Linker User's Guide*. 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 the *Wind River Diab Compiler Options Reference*.

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 crtlibso.c and ctordtor.c on page 171 for details.

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 1 on page 170 and described in crtlibso.c and ctordtor.c on page 171.

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** and **destructor, destructor(n) Attribute** in Attribute Specifiers Reference on page 73 . To change the default priority for initialization and finalization functions, see the -Xinit-section-default-pri entry in the *Wind River Diab Compiler Options Reference*.

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 $** *nn* sections (instead of **.ctors** and **.dtors**). To use old-style initialization and finalization, enable -Xinit-section=2 (see -Xinit-section entry in the *Wind River Diab Compiler Options Reference*). In this mode, the compiler also supports the use of special **_STI__** *nn* _ and **_STD__** *nn* _ prefixes (as well as **constructor** and **destructor** attributes) to identify initialization and finalization functions and set their priority. In cases where both **.init$** *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.

17.3. Hardware Exception Handling

An embedded system application (without an operating system) must handle hardware exceptions itself. The compiler provides a library function as well as pragma support for exception handling.

raise( ) Function, interrupt Pragma, and section Pragma

The compiler provides the following support for interrupt routines:

- The library function raise( ), which can be called with an appropriate signal from the interrupt routine to raise a signal.
- A #pragma interrupt which specifies that a function is an exception handler.
- A **#pragma section** directive that can place exception vectors at an absolute address.

Hardware Exception Handling Documentation

For a description of the exception (interrupt) handing provided by hardware, see the current RH850 architecture guide.

17.4. Library Exception Handling Modification

The default library exception handling may be modified, if required.

Default Exception Handling

On error, many standard library functions set **errno** and return a null or undefined value as described for each function in the *Wind River Diab Compiler C Library Reference: 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).

Code for Exception Handling

__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).

17.5. Linker Command File

A custom linker command file must specify where to allocate code and data for embedded systems.

Linker Command File Properties

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 the *Wind River Diab Compiler Linker User's Guide* for more information about the command language an example.

Compiler Invocation With Custom Linker Command File

When invoking a compiler driver such as **dcc**, specify a non-default linker command file using the **-Wm** option:

```
-Wmpathname
```

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.

Default and Target-Specific Linker Command Files

If no -W m option is used, the linker will use file *versionDir* **/conf/default.dld**. Documentary comments are included in this file; see it for details. For more information about the -W m option, see the *Wind River Diab Compiler Options Reference*.

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.

17.6. POSIX/UNIX Functions

The compiler installation provides stubs for a set of POSIX/UNIX functions for embedded systems.

Source File Location

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.

For information about variants of these libraries for different object module formats, see About Components and Directories on page 33.

Modification and Use

To use the 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. For more information, see the *Wind River Diab Compiler Utilities Reference: D-AR Archiver*.
- If a copy of **libchar.a** or **libram.a** was modified, see the *Wind River Diab Compiler C Library Reference* for a detailed description of how the libraries are structured and searched.

Character I/O Functions

The predefined files **stdin**, **stdout**, and **stderr** use the __inchar( )/__outchar( ) functions in *versionDir* **/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`

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 (*versionDir* **/example/** *...*) for suggestions on recompiling **chario.c** for the selected target board.

File I/O Functions

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** or **windiss** is used as part of a **-t** *tof* **:cross** or **-t** *tof* **:windiss** option when linking.

For a convenient way to create RAM-disk files for use with these functions, see Program Arguments, Environment Variables, and Predefined Files on page 182.

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, see the *Wind River Diab Compiler C Library Reference*.

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.

satty( )

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.

Additional 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 Library Exception Handling Modification on page 175.

_exit( )

In **_exit.c**, closes all open files and halts. See Startup and Termination Code on page 170.

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 *init.c* in Startup and Termination Code on page 170.

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.

17.7. Communicating with the Hardware

Various features facilitate access to the hardware in an embedded environment.

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 About Argument Passing on page 108 for details.

Note: 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.

Embedding Assembler Code

Use the asm keyword to intermix assembler instructions in the compiler function. See Embedding Assembly Code on page 115 for details.

Accessing Variables and Functions at Specific Addresses

There are several ways to place a variable or function at a specific absolute address.

At compile-time you can use the **#pragma section** directive to specify that a variable should be placed at an absolute address (see section and use_section Pragma Behavior and Usage on page 157). The advantages of using absolute sections are as follows:

- 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 are:

```
// 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() {
// ...
}
```

Alternatively, at compile-time you can use 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();
```

Alternatively, at link time you can define the address of an identifier.

For example, in the C file use:

```
        extern volatile int mem_port; /* variable */
        extern int mem_func(); /* function */

        mem_port = mem_port + mem_func();
```

And then 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.

Alternatively, place the variables or functions in a special named section during compilation and then locating the section via a linker command file.

For more information, see the discussion of the linker command language in the *Wind River Diab Compiler Linker User's Guide*.

Shared Symbols in Multicore Systems

Using techniques described in Accessing Variables and Functions at Specific Addresses on page 180, it is possible to share symbols across cores in a multicore system.

A demonstration program is installed with the Wind River Diab Compiler at the following location:

installDir **/diab/** *releaseDir* **/example/multi_core_share**

The demonstration program is a proof of concept, and not a full implementation. It extracts symbol addresses from the linker output for one module, and uses these to generate a small assembly-language program to locate the symbols at the correct addresses for a second module.

Refer to the program source for complete details. The program is written for the PowerPC architecture, but can be adapted to other multicore systems.

17.8. Reentrant and Thread-Safe Library Functions

Most library functions are reentrant, and most of those are thread safe. Where this is not the case, reentrant and thread-safe versions may be created.

Reentrant Functions

Most library functions are reentrant, although in some cases this is impossible because the functions are by definition not reentrant.

In the *Wind River Diab Compiler C Library Reference: C Library Functions*, the "Reference" portion of each function description indicates "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 the *Wind River Diab Compiler C Library Reference: C Library Functions* for more information.

Thread-Safe Functions

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.

17.9. 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** utility brings the same capabilities to programs running in an embedded environment without the need for an operating system or file devices.

setup Utility Capabilities

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 the *File I/O* section of POSIX/UNIX Functions on page 176.

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 Startup and Termination Code on page 170 for related details.)

setup Utility Syntax

The syntax for **setup** is 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.

setup Usage

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.

17.10. Avoiding Exception Handling, RTTI, and Dynamic Memory Allocation

C++ exception handling and Run-Time Type Identification (RTTI) add some overhead to application footprints. Applications that don't require these features should be created with the following rules in mind.

Rules for Eliminating C++ Exception Handling and RTTI

When creating an application without C++ exception handling and Run-Time Type Identification (RTTI), observe the following rules:

- Remove C++ exception handling (**try**, **catch**, and **throw**) code from your application.
- Disable generation of any exception handling and RTTI code in your own compilation units by compiling them with -Xexceptions-off and -Xrtti-off.
- Use the abridged version of the C++ library **libstlabr.a**, which is free of exception handling and RTTI, in place of **libstrl.a**, by using -Xc++-abr to compile your code and linking with -lstabr.
- Use the abridged library **libdabr.a**, which does not include exception handling and RTTI support, in place of **libd.a**. To use **libdabr.a**, use the linker option -ldabr instead of -ld. Note that -ldabr must be specified for both the compiler and the linker.

For more information on C++ libraries, exception handling, and RTTI, see C++ Standard and Abridged Libraries on page 87.

Avoiding Dynamic Memory Allocation

Some applications are designed to avoid the use of dynamic memory allocation. Due to the details of the C++ ABI followed by the Diab compiler, even applications that avoid all dynamic memory allocation at runtime may nevertheless have a link-time dependency on malloc( ) and free( ).

To avoid any runtime or link-time dependency on malloc( ) or free( ), observe the following rules:

- Define the following functions, which will not be called at runtime unless the application explicitly uses non-placement forms of **new**, **new[]**, **delete**, or **delete[]**:

```
void *operator new(size_t size)
{
   // Error: should never be called
}

void *operator new[](size_t size)
{
   // Error: should never be called
}

void operator delete(void *ptr)
{
   // Error: should never be called
}

void operator delete[](void *ptr)
{
   // Error: should never be called
}
```

- Make sure that no static storage duration objects have destructors.

Additionally, provide the following functions if you wish to avoid any dependency on **libc.a**:

- If you're creating arrays of objects (with non-trivial constructors or destructors), then you must provide memset( ), either by linking against **libc.a** or by providing an application-specific definition.
- If you're using pure virtual functions, then define the following error handler:

  ```
  extern "C" void __cxa_pure_virtual(void)
  {
    // Error: will be called if application calls a
    // pure virtual function (i.e. programming error)
  }
  ```

- If you're using variables requiring guarded initialization, then define the following error handler:

  ```
  extern "C" void __diab_cxa_guard_recursive(void)
  {
    // Error: will be called if the initializer for a variable requiring
    // guarded initialization depends on the variable having already been
    // initialized (i.e. programming error)
  }
  ```

  Note that the following kinds of variables require guarded initialization:

  - function scope statics
  - template static data members

  This does not apply to variables of simple types, but only those that require non-trivial construction.

17.11. Profiling in an Embedded Environment

Profiling involves the collection of information about your program while it executes. The information can then fed back to the compiler for more optimal code generation. The compiler implements profiling with the -Xblock-count and -Xfeedback options.

Before You Begin

The *profile data* is written by the profiling code to a target file named **dbcnt.out**. You therefore must either have an environment in which target files may be connected to files on your host, or use the RAM-disk service (see *File I/O Functions* in POSIX/UNIX Functions on page 176).

Note: Profiling using -Xblock-count and -Xfeedback is not supported for VxWorks applications.

Procedure

1. Compile your code with -Xblock-count to insert counting code.

   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).

   For example:

   ```
   dcc -c -Xblock-count file1.c file2.c
   ```

2. Copy library module *versionDir* **/src/_exit.c** and modify it to write the profiling data back to your host system.

   See **_exit.c** for additional details.

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**, you would recompile with:

```
dcc -c -DPROFILING -DRAMDISK versionDir/src/_exit.c
```

3. Compile the rest of your program and link as usual.
4. 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**.
5. Use the **ddump** command to convert it to a binary file if the profiling information was transferred back to the host in ASCII format,

   For example:

   ```
   ddump -B -o dbcnt.out your-file-of-collected-profile-data
   ```

   The **dbcnt.out** output filename is used here because it is the default for the -Xfeedback option used in the next step.
6. Recompile the modules profiled with the -Xfeedback option.

   The compiler optimizes the code 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).

   For example:

   ```
   dcc -c -Xfeedback -XO file1.c file2.c
   ```

   By default, the -Xfeedback option assumes that **dbcnt.out** is the name of the file containing profile data. To specify a different file, use -Xfeedback=*profile-file* .

17.12. Multi-Channel Sequencer (MCS) Code in a System Image

Some CPUs include the Bosch GTM-IP v1.4 module. Each GTM (generic timer module) includes between three and eight Multi Channel Sequencer (MCS) sub-modules. Each MCS sub-module contains a micro controller with its own RAM and register set. The code that you write for each MCS sub-module must be combined with the code for the main CPU processor using a special procedure to create a single system image.

About Creating a System Image Including MCS Code

Note: The Wind River Multi Channel Sequencer assembler supports the GTM IP specification version 3.1.5.1.

The main CPU of a system that includes a GTM loads a single executable image. The image must include the code for each MCS, information about where to load the MCS executables in RAM, and information about any global symbols in the MCS code to which references are made in the main CPU's code.

The code for the main CPU executable and the MCS executables must be linked into a single system image that you can load from flash. But this involves a special process that does not involve linking all the object files.

The object files for each MCS must be linked using its own linker command file so that symbols such as branch labels are resolved to values in the scope of the executing processor's memory (that of the MCS's processor). If they were linked using the main CPU's linker command file, the MCS's symbols would resolve to addresses that lie in the scope of the main CPU and outside the scope of the MCS itself.

Because the MCS executables must be fully linked, they cannot (and must not) be linked again. Instead the required sections of each MCS executable must be extracted using **ddump**, and then added to the main CPU's executable using the **INCBIN** directive. All sections are required except **.symtab**, **.strtab**, and **.shstrtab**, and any sections whose size is zero.

In addition, because the binaries do not provide any location information, it must be defined in the linker command file for the main CPU executable to ensure that the sections are located at the correct offsets and that global symbols are defined at the proper locations.

Overview of Steps

The major steps that you must be perform to generate the system image for the main CPU are as follows:

1. Assemble and link the code for each MCS into an executable.
2. Extract the required sections as binary files, as well as the section offsets and global symbol offsets (as numbers), from each MCS executable as binary files.
3. Define external symbols that reside in the MCS code for the main CPU executable.
4. Link the MCS sections that you have extracted from the MCS executables with the executable for the main CPU.

When properly executed, this process ensures that the MCS executable sections are placed in their own RAM when the system boots, and that references in the main GTM application to global symbols in the MCS executables are resolved. (If this process is not correctly performed, it can result in problems that are difficult to debug.)

For example, if you add a **.org** statement to your MCS code that contains a global variable that has to be accessible by the main CPU, you must:

- Extract an additional section (**abs.offset**).
- Add a new line to the main CPU linker command file to place the extracted section at the proper offset.
- Define a new symbol for the global variable (which is in this section) in the linker command file for the main CPU's code.

Missing one of the steps can have very bad effects. Also note that changing the argument of a **.org** statement requires a modification to the linker command file for the main CPU's code, otherwise the change cannot take effect.

17.13. Creating a System Image with MCS Code

Creating a system image that includes MCS code involves creating the MCS executables, extracting the required sections, defining global symbols for the main executable (if required) that are in the MCS code, and linking the MCS sections with the main executable.

Procedure

1. Create the MCS Executables

   MCS code must be written in assembly. To build the code for an MCS, perform the following steps:

   a. Create a linker command file for the MCS.

      The **MEMORY** definition must match what is defined in **MCFG_CTRL** at runtime. In addition, the **SECTIONS** definition must map sections to memory locations in a way that allows the MCS code to operate correctly.

   b. Invoke the assembler on all assembly source files for the MCS using the MCS target option.

      Make sure the MCS target option you choose reflects whether the code is big-endian or little-endian. (See, for instance, the example code in Example: Creating a System Image With MCS Code on page 188.) For information about assembler target processor options, see the entry for the -t assembler option in the *Wind River Diab Compiler Options Reference*, and the *Wind River Diab Compiler Target Configuration Reference*.

   c. Create the MCS executable by linking the object files. Invoke **dld** with the linker command file and the MCS target option.

For information about linker target processor options, see the entry for the -t linker option in the *Wind River Diab Compiler Options Reference*, and the *Wind River Diab Compiler Target Configuration Reference*.

2. Extract Required Elements From the MCS Executables

   To extract the required sections, section offsets, and global symbol offsets from the MCS executables, perform the following operations using **ddump**:

   a. Extract all required sections from MCS executables as binary files.

      You must extract all sections except **.symtab**, **.strtab**, and **.shstrtab** and any sections whose size is zero.

   b. Extract the section offsets from the MCS executable sections you have extracted.

   c. Extract all global symbol offsets from MCS executables.

3. Define MCS Global Symbols for the Main CPU Code

   If the code for the main CPU executable references global symbols in MCS code, you may need to define them explicitly.

   External symbols in MCS memory are usually small in size. If they are defined only as external symbols, the compiler may use near-relative addressing for symbol accesses, with the expectation that the external symbols reside in one of the small data areas. If near-relative addressing is used, linking the primary CPU's executable with MCS executables will fail unless the appropriate small data area covers the MCS RAM (from the point of view of the main processor).

   If no small data area covers the MCS RAM(s) the addressing mode of external symbols must be specified in the code for the main CPU executable. For example:

```
#pragma section MCSVAR far-absolute RW
#pragma use_section MCSVAR short_in_mcs_ram, uint_in_mcs_ram
extern short short_in_mcs_ram;
extern unsigned int uint_in_mcs_ram;
```

4. Link MCS Sections With the Main CPU Executable

   To link the required sections that you have extracted from the MCS executable with the main CPU executable, you must do the following:

   a. Create a linker command file that includes information about the memory areas for the MCS code, and that ensures that they will be initialized by the main copy table.

      Note the following requirements:

      - The section must be placed at the proper offsets relative to the start of the MCS' areas in RAM. The offsets added to the linker command file for the main CPU's executable must match the ones in the MCS executable.

      - The global symbols must be defined at the proper offsets relative to the start of MCS RAM.

   b. Link the main CPU executable with the MCS code using the **INCBIN** linker directive.

   For information about **INCBIN**, see the discussion of section contents in the *Wind River Diab Compiler Linker User's Guide*.

17.14. Example: Creating a System Image With MCS Code

This example illustrates how to create an image for a system that contains MCS code, including automation of key steps with a Python script.

Hardware Configuration

This example assumes that the system has a main CPU with a GTM that has 4 MCS modules.

The main CPU has 8 MB of RAM, and 8 MB of ROM, and the MCS code and variables are copied from ROM to RAM at power-on.

From the point of view of the main CPU, the RAM of an MCS is mapped at **0xf0138000+(** *i* ***0x10000)**, where *i* is the MCS number, ranging from 0 to 3. The memory configuration for the 4 MCS modules is **SWAP** for MCS0, **BORROW** for MCS1, **DEFAULT** for MCS2, and again **DEFAULT** for MCS3.

Note that the addresses used in this example are fictional.

Creating the MCS Linker Command Files

There are no particular constraints on the MCS linker command files except for the following:

- The **MEMORY** definition must match what is defined in **MCFG_CTRL** at runtime.
- The **SECTIONS** definition must map sections to memory locations in a way that allows the MCS to operate correctly.

The linker command files for the MCS code in this example use the following syntax:

```
MEMORY { mp0: org = MP0_org , len = MP0_size mp1: org
                = MP0_size , len = MP1_size } SECTIONS { GROUP statements that map sections to mp0
                and/or mp1 }
```

For the configuration of memory in this example, the values need to be set as listed below (the name of the respective linker command files are also provided).

MCS	*MP0_org*	*MP0_size*	*MP1_size*	**File Name**
MCS0	0x0	0x1000	0x1000	**mcs0.dld**
MCS1	0x0	0x800	0x1800	**mcs1.dld**
MCS2	0x0	0x0	0x800	**mcs2.dld**
MCS3	0x0	0x1000	0x800	**mcs3.dld**

The **GROUP** statements depend on the sections used in the source code files for the MCS. For example, for MCS3, the text is mapped to **mp0**, and data to **mp1**, as follows:

```
GROUP : { .text(TEXT) : {} } > mp0 GROUP : { .data(DATA) :
                {} } > mp1
```

Note: For **MCS2** no mapping to **mp0** can be used, because the size of **mp0** for **MCS2** is zero.

Creating the MCS Executables

Assume that for MCS0 through MCS2 we have only one source file for each MCS, named **mcs0.s**, **mcs1.s**, and **mcs2.s** respectively--all using standard Diab syntax. For MCS3, we have two source files named **mcs3_0.mcs** and **mcs3_1.mcs**, both of which use CASPR-MCS syntax. The executables are created with two sets of commands.

Note: This example uses the syntax for big-endian code (-tMCSEN) instead of the syntax for little-endian code (-tMCSLN).

- Invoking the assembler on all assembler source files as follows:

```
das -tMCSEN:simple -o mcs0.o mcs0.s das
                        -tMCSEN:simple -o mcs1.o mcs1.s das -tMCSEN:simple -o mcs2.o mcs2.s da
s
                        -tMCSEN:simple -Xaccept-caspr-syntax -o mcs3_0.o mcs3_0.mcs das
                        -tMCSEN:simple -Xaccept-caspr-syntax -o mcs3_1.o mcs3_1.mcs
```

- Linking the objects into executables as follows:

```
dld -tMCSEN:simple -o mcs0.x mcs0.o mcs0.dld dld
                        -tMCSEN:simple -o mcs1.x mcs1.o mcs1.dld dld -tMCSEN:simple -o mcs2.x
mcs2.o
                        mcs2.dld dld -tMCSEN:simple -o mcs3.x mcs3_0.o mcs3_1.o mcs3.dld
```

Creating the Main CPU Object Files

The procedure for compiling the code for the main CPU is standard. For sake of simplicity in this example, there is one source file called **main.c**. It is compiled as follows (the type of target is not significant):

```
dcc -tRH850G4MHFH:simple -c -g -o main.o
                main.c
```

Creating the Main CPU Linker Command File

The linker command file for the main CPU requires special treatment in that you must include information about the memory areas for the MCS code, and that they are initialized by the main copy table and that they are loaded from the region named **rom**. See the **MEMORY** and **GROUP** statements in the example below.

Note that the include files referenced in the linker command file (**mcs0.dld.include** and so on) contain linker command file directives for the MCS code. The include files are generated by the script that is described in A Python Script for Automating MCS Code Inclusion on page 191.

```
-Xgenerate-copytables MEMORY { ram: org = 0x00000000, len =
                0x800000 rom: org = 0xff800000, len = 0x800000 MCS0: org = 0xf0138000, len = 0x200
0
                MCS1: org = 0xf0148000, len = 0x2000 MCS2: org = 0xf0158000, len = 0x800 MCS3: or
g =
                0xf0168000, len = 0x1800 } SECTIONS { GROUP : { .text (TEXT) : { *(.text) *(.rdata
)
                *(.rodata) *(.frame_info) *(.eh_frame) *(.j_class_table) *(.init) *(.fini) } .ctor
s
                ALIGN(4) : { ctordtor.o(.ctors) *(.ctors) } .dtors ALIGN(4) : { ctordtor.o(.dtors)
                *(.dtors) } .copytable : {} .sdata2 (TEXT) : {} } > rom GROUP LOAD(>rom)
                COPYTABLE: { .data (DATA) : {} .sdata (DATA) : { *(.sdata) *(.j_spdata) } .sbss
                (BSS) : {} .bss (BSS) : {} __HEAP_START = .; } > ram GROUP LOAD(>rom)
                COPYTABLE: { #include "mcs0.dld.include" } > MCS0 GROUP LOAD(>rom) COPYTABLE:
                { #include "mcs1.dld.include" } > MCS1 GROUP LOAD(>rom) COPYTABLE: { #include
                "mcs2.dld.include" } > MCS2 GROUP LOAD(>rom) COPYTABLE: { #include
                "mcs3.dld.include" } > MCS3 } __SP_INIT = ADDR(ram)+SIZEOF(ram); __SP_END =
                __SP_INIT-0x800; __HEAP_END = __SP_END; ___HEAP_START = __HEAP_START; ___HEAP_END
=
                __HEAP_END; ___SP_INIT = __SP_INIT; ___SP_END = __SP_END;
```

A Python Script for Automating MCS Code Inclusion

The Python script described in this section automates the following operations:

- Extracting the required MCS executables sections into binary files.
- Determining their offsets relative to the start of the MCS RAM.
- Identifying the global symbols.
- Generating an include file (such as **mcs0.dld.include**) that is added to the linker command file for the main CPU executable by the **dcc** preprocessor. The include file contains linker command directives for the required sections, for global symbols, and for their offsets.

The script is split into its key elements below, to facilitate their description. Note that the script is provided for illustrative purposes only, and not for use as-is.

First, some variables and a wrapper around the somewhat lengthy Popen( ) command are defined. Note the list of section names to ignore (**.symtab**, **.strtab**, and **.shstrtab**).

```
#!/usr/bin/env python import sys import subprocess
                ignoreSections=['.symtab', '.strtab', '.shstrtab'] elfExtension='.x' pathSep='/' d
ef
                execCmd(cmd): proc=subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE,
                stdin=subprocess.PIPE, stderr=subprocess.STDOUT, close_fds=True) return
                proc.stdout
```

Then the sortSections( ) function sorts the section table by section offset. The input is the raw text output of the **ddump -h** command. While sorting, the function also discards sections listed in the **ignoreSections[ ]** table and any section whose size is zero.

```
def sortSections(sectionTable): addrs=[] addrLookup={}
                sortedSections=[] for line in sectionTable.readlines(): if
                (not(line.startswith('[No]'))) & line.startswith('[') : line=line.rstrip()
                line=line.rstrip() lineSplit=[] for s in line.split(' '): if s!='':
                lineSplit.append(s) if ( eval(lineSplit[5])>0) & ( not(lineSplit[6] in
                ignoreSections ) ): addr=eval(lineSplit[3]) addrLookup[addr]=lineSplit
                addrs.append(addr) addrs.sort() for addr in addrs:
                sortedSections.append(addrLookup[addr]) return sortedSections
```

The handle_sections( ) function extracts all sections--except those in the **ignoreSections[ ]** table--from the MCS executable into binary files. It also adds two lines to the include file: the location counter at the right offset relative to the start of MCS RAM, and the **INCBIN** directive.

```
def handle_sections(baseName, noExtName, fileName, dld,
                startLabel): sectionTable=execCmd('ddump -h '+fileName) for lineSplit in
                sortSections(sectionTable): section=lineSplit[6] bin=noExtName+section+'.bin'
                execCmd('ddump -Ru -n '+section+' -o '+bin+' '+fileName) dld.write('. =
                '+startLabel+ ' + ' + lineSplit[3]+ ';\n') dld.write('.'+baseName+section+': {
                INCBIN('+bin+')}\n')
```

The handle_globals( ) function identifies all global symbols in the MCS executable and their respective offsets and adds this information to the include file.

```
def handle_globals(fileName, dld,startLabel):
                varTable=execCmd('ddump -N '+fileName) for line in varTable.readlines():
                line=line.rstrip() lineSplit=line.split('|') if len(lineSplit)>7: if
                lineSplit[4]=='GLOB ': dld.write(lineSplit[7]+' = '+ startLabel +' + '
                +hex(eval(lineSplit[1]))+';\n')
```

The main Python function performs the following operations:

- Stores the file name that it takes as its first argument--once without the extension and once without the extension and path--and uses these strings to create the include file (which the **dcc** preprocessor adds to the linker command file for the main CPU's code) and binary files for the extracted sections.
- Marks the beginning of the **MCS** RAM region in the include file with a symbol.
- Calls handle_sections( ) for each section to extract the required sections from the MCS executable to their respective binary files. It also adds two lines per section to the include file--setting the location counter to the right offset relative to the start of MCS RAM, and adding the **INCBIN** directive.
- Marks the end of the **MCS** RAM region in the include file with a symbol.
- Calls handle_globals( ) to add information about all the global symbols and their offsets to the include file.

```
if __name__ == "__main__": fileName=sys.argv[1]
                fileNameWOExt=fileName[0:fileName.rfind(elfExtension)] if
                fileNameWOExt.rfind(pathSep)>-1: fileNameBase=fileNameWOExt.split(pathSep)[-1]
                else: fileNameBase=fileNameWOExt startLabel='__'+fileNameBase+'_start__'
                endLabel='__'+fileNameBase+'_end__' dld=open(fileNameWOExt+'.dld.include','w')
                dld.write(startLabel+'=.;\n') handle_sections(fileNameBase, fileNameWOExt, fileNam
e,
                dld, startLabel) dld.write(endLabel+'=.;\n') handle_globals(fileName, dld,
                startLabel) dld.close()
```

Executing the Python Script

Assuming the file name for the script is **dldgen.py**, extracting the MCS sections, and generating the linker include files for all the MCS executables is achieved with the following commands:

```
./dldgen.py mcs0.x ./dldgen.py mcs1.x ./dldgen.py mcs2.x
                ./dldgen.py mcs3.x
```

Python Script Processing and Output

To clarify what the Python script does, assume that there is some code in the **.text** section of **mcs1.x** , and some variables in the **.data** section. Furthermore, assume that some code and variables had been placed at offset of 0x200 using a **.org 0x200** statement. Finally, assume we had a global symbols named **bar** in this section, and one more global symbol named **foo** at the very start of the data section.

The output of the following command:

```
./dldgen.py mcs1.x
```

is **mcs1.text.bin**, **mcs1.data.bin**, and **mcs1.abs.00000200.bin**, and **mcs1.dld.include**. The first three files contain the extracted sections, and the fourth file is the include file containing linker command file directives that the dcc preprocessor adds to the linker command file for the main CPU code. The content of the of the include file would look like this:

```
__mcs1_start__=.; . = __mcs1_start__ + 0x0; .mcs1.text: {
                INCBIN(../obj/mcs1.text.bin)} . = __mcs1_start__ + 0x200; .mcs1.abs.00000200: {
                INCBIN(../obj/mcs1.abs.00000200.bin)} . = __mcs1_start__ + 0x800; .mcs1.data: {
```

```
                INCBIN(../obj/mcs1.data.bin)} __mcs1_end__=.; bar = __mcs1_start__ + 0x200; foo =
                __mcs1_start__ + 0x800;
```

Note: The locations of the text and data sections conform to the definition in **mcs1.dld** (see Creating the MCS Linker Command Files on page 189). Also note how the global symbols are defined.

Creating the Final System Executable

At this point we have produced the object files and the linker command file for the main CPU, all the extracted sections from the MCS executables, and the linker command include files for all the MCS RAM areas.

The final executable is created by invoking the following commands:

```
dcc -E linkerCmd.dld | grep -v "\#" > linkerCmd.prep.dld
                dcc -tRH850G4MHFH:simple main.o -o main.x -Wm linkerCmd.prep.dld -Ws
                crt0.o
```

where **crt0.o** is the one shipped as part of the Diab tool chain for each target, and **linkerCmd.dld** is the linker command file for the main CPU's executable.

Note: The initialization code must perform all operations necessary to enable the GTMs (such as enabling their clocks).

The first invocation of dcc preprocesses the main CPU linker command file removes any line starting with a hash character. The second invocation invokes the linker to build the final executable **main.x**, which contains all the MCS executable elements (which are loaded into their respective RAM areas by the startup code).

Summary of Commands

In total, the following commands have been executed to produce the final executable:

```
das -tMCSEN:simple -o mcs0.o mcs0.s das -tMCSEN:simple -o
                mcs1.o mcs1.s das -tMCSEN:simple -o mcs2.o mcs2.s das -tMCSEN:simple
                -Xaccept-caspr-syntax -o mcs3_0.o mcs3_0.mcs das -tMCSEN:simple
                -Xaccept-caspr-syntax -o mcs3_1.o mcs3_1.mcs dld -tMCSEN:simple -o mcs0.x mcs0.o
                mcs0.dld dld -tMCSEN:simple -o mcs1.x mcs1.o mcs1.dld dld -tMCSEN:simple -o mcs2.x
                mcs2.o mcs2.dld dld -tMCSEN:simple -o mcs3.x mcs3_0.o mcs3_1.o mcs3.dld dcc
                -tRH850G4MHFH:simple -c -g -o main.o main.c ./dldgen.py mcs0.x ./dldgen.py mcs1.x
                ./dldgen.py mcs2.x ./dldgen.py mcs3.x dcc -E linkerCmd.dld | grep -v "\#" >
                linkerCmd.prep.dld dcc -tRH850G4MHFH:simple main.o -o main.x -Wm linkerCmd.prep.dl
d
                -Ws crt0.o
```

These commands could obviously be added to a makefile to allow for incremental builds and changes. For example, any change to an MCS source would then lead to re-extraction of the sections, re-creation of the linker command include files, re-preprocessing of the main linker command file, and finally to a re-execution of the final link--all without any manual interaction.

18. LINT FACILITY

18.1. Lint Operation

The lint facility is a powerful tool to find common C programming mistakes at compile time. (For C++, see the -Xsyntax-warning-on entry in the Wind River Diab Compiler Options Reference .)

Lint Use and Functionality

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. (Note that -Xstop-on-warning works with ctoa only, not etoa.)
- Each individual check that -Xlint performs can be turned off by using a bit mask. See the -Xlint entry in the *Wind River Diab Compiler Options Reference* for details.
- For C programs, -Xlint can be used with the -Xforce-prototypes option to warn of a function used before its prototype.

18.2. Lint Usage Example

Learn to use Lint from a simple example.

About This Task

The program **lint.c** contains several errors. The **lint** facility can be used ways to detect those errors.

Procedure

Run **lint** on **lint.c** in the following way:

```
dcc -c -tprocessor_target -Xlint -Xforce-prototypes lint.c
```

The comments in the C program in demonstrate probable defects that will be detected by using -Xlint and -Xforce-prototypes (which causes lint to warn if a function is used before its prototype is declared). These 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.

```
void f1(int);
short f2();
                              /* (-Xlint mask bit disables)          */
static int f4(int i)          /* function never used          (0x10) */
{
   if (i == 0)
        return;                /* missing return expression      (0x20) */
   return i+4;
}

 static int f5(int i);        /* error: function not found               */

 static int i1;               /* variable never used             (0x10) */

 int m(char j, int z1)        /* parameter never used            (0x10) */
   {
     int i, int4;
     char c1;
     unsigned u = 1;          /* variable set but not used       (0x40) */
     int z2;                  /* variable never used             (0x10) */

     c1 = int4;               /* narrowing type conversion      (0x100) */

     if (c1) {
         int4 = 100;
         i    = 0;
     } else {
         c1 = 266;            /* warning: constant out of range (=)      */
         int4 = 101;
     }

     switch(i) {             /* variable i used before being set (0x02) */
              f5(3);         /* statement not reached            (0x80) */
              break;

         case 0:             /* -X force prototype, not lint, warns:    */
              f2(2);         /*    function has no prototype            */
              f3(1);         /*    function not declared                */
              f5(int4);
              break;

         case 4294967297:    /* warning: constant out of range          */
              j=32;          /* variable set but not used        (0x40) */

         deflaut:            /* label not used                   (0x04) */
              c1=f2(99);     /* narrowing type conversion       (0x100) */
              f5(c1);
    }
    goto myLabel;
    f5(42);                  /* statement not reached            (0x80) */

    myLabel: int4 = 42;

    if ( int4<50 )           /* cond. expression always true/false(0x08)*/
    {
         c1=0;

         if ( c1 && int4 )    /* cond. expression always true/false(0x08)*/
         {
              f5(int4);
         }
62:     }
63:  }                           /* missing return expression         (0x20)*/
```

lint produces the following output.

Note: Warnings are not necessarily in line number order because the compiler detects the errors during different internal passes.

```
"src/lint.c", line 7: warning (dcc:1521): missing return expression
"src/lint.c", line 22: warning (dcc:1643): narrowing or signed-to-unsigned
                                type conversion found: int to unsigned char
"src/lint.c", line 28: warning (dcc:1244): constant out of range (=)
"src/lint.c", line 28: warning (dcc:1244): constant out of range (=)
"src/lint.c", line 37: warning (dcc:1500): function f2 has no prototype
"src/lint.c", line 38: warning (dcc:1481): function f3 not declared
"src/lint.c", line 38: warning (dcc:1500): function f3 has no prototype
"src/lint.c", line 42: warning (dcc:1243): constant out of range
"src/lint.c", line 46: warning (dcc:1643): narrowing or signed-to-unsigned
               type conversion found: short to unsigned char
"src/lint.c", line 45: warning (dcc:1251): label deflaut not used
"src/lint.c", line 15: warning (dcc:1516): parameter z1 is never used
"src/lint.c", line 20: warning (dcc:1518): variable z2 is never used
"src/lint.c", line 33: warning (dcc:1522): statement not reached
"src/lint.c", line 50: warning (dcc:1522): statement not reached
"src/lint.c", line 63: warning (dcc:1521): missing return expression
"src/lint.c", line 19: warning (dcc:1604): Useless assignment to variable                         u
.
               Assigned value not used.
"src/lint.c", line 28: warning (dcc:1604): Useless assignment to variable                         c
1.                Assigned value not used.
"src/lint.c", line 43: warning (dcc:1604): Useless assignment to variable                         j
.
               Assigned value not used.
"src/lint.c", line 54: warning (dcc:1606): conditional expression or part
               of it is always true/false
"src/lint.c", line 58: warning (dcc:1606): conditional expression or part
               of it is always true/false
"src/lint.c", line 32: warning (dcc:1607): variable i is used before set
"src/lint.c", line 22: warning (dcc:1607): variable int4 is used before set
"src/lint.c", line 4: warning (dcc:1517): function f4 is never used
"src/lint.c", line 11: error (dcc:1378): function f5 is not found
"src/lint.c", line 13: warning (dcc:1518): variable i1 is never used
```

19. OBJECT AND EXECUTABLE FILE FORMAT

19.1. Executable and Linking Format (ELF)

The compiler supports the Executable and Linking Format (ELF).

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.

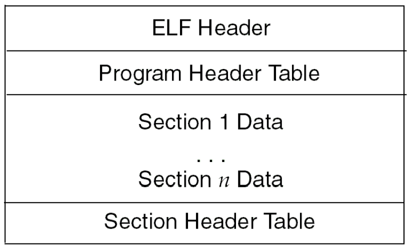
Section data

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:



ELF Headers

The ELF32 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_versio
n;
                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; };
```

The ELF64 header contains general information about the object file and has the following structure from the file **elf.h** (**Elf64_Half** is two bytes, **Elf64_Addr** and **Elf64_Of** are 8 bytes, the other types are four bytes):

```
#define EI_NIDENT 16 typedef struct { unsigned char
                e_ident[EI_NIDENT]; Elf64_Half e_e_type; Elf64_Half e_machine; Elf64_Word e_versio
n;
                Elf64_Addr e_entry; Elf64_Off e_phoff; Elf64_Off e_shoff; Elf64_Word e_flags;
                Elf64_Half e_ehsize; Elf64_Half e_phentsize; Elf64_Half e_phnum; Elf64_Half
                e_shentsize; Elf64_Half e_shnum; Elf64_Half e_shstrndx; };
```

Field	Description	
`e_ident`	Sixteen byte long string with the following content: 4-byte file identification: "\x7FELF"1-byte class: 1 for 32-bit objects1-byte data encoding: little-endian: 1, big-endian: 21-byte version: 1 for current version 9-byte zero padding	
`e_type`	The file type: relocatable: 1, executable: 2	
`e_machine`	Target architecture:	
	20	RH850
`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`	See the RH850 ABI specification.	
`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.	

Field	Description
`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.

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;
```

Program Header Fields

The form *NAME*(*n*) means that the symbolic value *NAME* has the value shown in the parentheses.

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.

Section Headers

There is an 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;
```

The form *NAME* (*n*) means that the symbolic value *NAME* has the value shown in the parentheses.

Table 1. 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 Implementation-Specific C++ Features Reference on page 90).
`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

Field	Description	
`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* clause and the ELF section type and flags assigned to the output section. For information about the *type-spec* clause, see the discussion of the linker command language in the *Wind River Diab Compiler Linker User's Guide.*

Type-spec	Section Type (sh_type)	Section Flags (sh_flags)
BSS	**SHT_NOBITS**	**SHF_ALLOC | SHF_WRITE**
COMMENT	**SHT_PROGBITS**	(none)

Type-spec	Section Type (sh_type)	Section Flags (sh_flags)
CONST	**SHT_PROGBITS**	**SHF_ALLOC**
DATA	**SHT_PROGBITS**	**SHF_ALLOC|SHF_WRITE**
TEXT	**SHT_PROGBITS**	**SHF_ALLOC|SHF_EXECINSTR**

Special Sections

Following are the names of some typical sections and explains their contents:

.text

Machine instruction.

.data

Initialized data.

.sdata

Small initialized data; see the -Xsmall-data option in the *Wind River Diab Compiler Options Reference*,

.bss

Uninitialized variables.

.sbss

Small uninitialized data.

.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.

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, for example:

```
ld.w var@ha[r3],r10
```

Then the 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.

Relocation entries have the following structures from the file **elf.h**:

```
typedef struct { Elf32_Addr r_offset; Elf32_Word r_info;
                Elf32_Sword r_addend; } Elf32_Rela; typedef struct { Elf64_Addr r_offset;
                Elf64_Xword r_info; Elf64_Sxword r_addend; } Elf64_Rela;
```

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 *versionDir* **/include/elf_** *target* **.h**.

For information about relocation types, see the RH850 ABI specification.

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.

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;
```

It has this structure for ELF64:

```
 typedef struct {      Elf64_Word st_name;      unsigned
                char   st_info;      unsigned char   st_other;      Elf64_Half st_shndx;
                Elf64_Addr st_value;      Elf64_Xword     st_size; } Elf64_Sym;
```

Symbol Table Fields

The form *NAME*(*n*) means that the symbolic value *NAME* has the value shown in the parentheses.

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

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.

20. MIGRATION OF CODE TO THE WIND RIVER DIAB COMPILER

20.1. Conversion of Code for Diab Compilation

Working with code originally developed for a different system or toolkit is usually straightforward, given the compatibility options provided by the Diab tools. The compiler accepts many GCC options, and provides other features that facilitate compilation without modifying existing source code. In addition, Wind River provides guidelines for identifying execution problems.

Common Issues

By default, GCC options (in a makefile or from the command line) are parsed and, if possible, automatically translated to equivalent Wind River Diab options.

Compilation issues may be related to things like missing header files, older C/C++ code that does not conform to current standards, and so on. The compiler provides options to address many of these problems without modifying the source code.

Runtime issues can be addressed with **lint**, investigation of conflicts with optimization, and close examination of common code problems.

20.2. GNU Option Translation

If -Xgcc-options-on is enabled (which it is by default), the compiler parses GCC option flags from the command line or makefile and, if possible, translates them to equivalent Wind River Diab Compiler options. Translations are determined by the tables in the file **gcc_parser.conf**. -Xgcc-options-verbose outputs a list of translated options.

Also see the -Xgcc-options-on, -Xgcc-options-off, and -Xgcc-options-verbose entries in the *Wind River Diab Compiler Options Reference*.

20.3. Compilation Issues and Their Resolution

Compilation issues may be related to things like missing header files, older C/C++ code that does not conform to current standards, and so on. The compiler provides options to address many of these problems without modifying the source code.

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 the **-i** entry in the *Wind River Diab Compiler Options Reference* for details).

Older C Code With 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: ...". Or correct the code.

Older C++ Code Written for Older Versions of the Compiler

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. Wind River strongly recommends bringing all source code into compliance with the ANSI standard.

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 the *Wind River Diab Compiler C Library Reference* for more information.

To link the older **iostream** and complex libraries, you must use the -l option (see the *Wind River Diab Compiler Options Reference*) explicitly. If you use the **dcc** driver or invoke **dld** directly, all the old libraries must be specified explicitly. Example:

```
dld -YP,search-path -l:windiss/crt0.o hello.o
        -o hello -ldold -lios -lc version-path/conf/default.dld
```

Note: All the older C++ libraries must be specified explicitly.

To select the old compiler and libraries by default, 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 in this regard, see the *Wind River Diab Compiler Getting Started*.

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 the *Wind River Diab Compiler Options Reference*) 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 Run-time Initialization and Termination on page 174 for more information.

20.4. Execution Issues and Their Resolution

When a program fails to execute properly various measures can be used to determine the problem, including the use of **lint**, cleaning up code that interferes with optimization, fixing improper dynamic memory allocation operations, and so on.

Compile With -Xlint

For C modules only, use -Xlint to enable compile-time checking that detects many non-portable and suspicious programming constructs.

Note: -Xlint does not interfere with optimization, and adds little time to compilation, so it can always be enabled.

See Lint Facility on page 194

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 that the -Xmemory-is-volatile option 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.

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.

The function free( ) automatically detects an attempt to free the same area more than once.

Two special features of malloc( ) and free( ) may help detect errors:

- If global variable _ _**malloc_init** is non-zero, then all memory allocated by malloc( ) will be set to the low-order byte of global integer variable _ _**malloc_val**. For example, before calling malloc( ) for the first time, execute:

  ```
  _ _malloc_init = 1;
  _ _malloc_val  = 0xff;
  ```

  to cause malloc( ) to initialize allocated memory to 0xff.
- If global variable _ _**malloc_check** is non-zero, malloc( ) and free( ) will check the system data for the list of all allocated and free memory blocks on every call.

If any of the above errors are detected, an error message will be written to *device***/dev/lp**. See Character I/O Functions on page 177.

If the "target environment variables" feature is being used (see Program Arguments, Environment Variables, and Predefined Files on page 182), the above global variables may be set with two target environment variables as follows:

DMALLOC_INIT=*value*

Sets _ _**malloc_init** to 1 and _ _**malloc_val** to the given *value* (normally a hex byte, for example, 0xff).

DMALLOC_CHECK

Sets _ _**malloc_check** to 1.

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**.

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)**.

Code That 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 6.48 __packed__ and packed Keywords, p. 67 on methods for accessing byte-swapped data.
- Floating point format.
-
- Sign of plain **char** . By default, the sign of plain **char** is **signed** for this architecture family. Use the option -Xchar-unsigned to force all plain **char** variables to be unsigned.
- Sign of plain **int** bit-fields. By default, bit-fields of type **int** are unsigned. For C modules, use the option -Xbit-fields-signed to be compatible with systems that treat plain **int** bit-fields as signed.