

WIND RIVER DIAB COMPILER GETTING STARTED, 7.0.6



Copyright Notice

Copyright © 2024 Wind River Systems, Inc.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means without the prior written permission of Wind River Systems, Inc.

Wind River, Simics, Tornado, and VxWorks are registered trademarks of Wind River Systems, Inc. Helix, Pulsar, Rocket, Titanium Cloud, Titanium Control, Titanium Core, Titanium Edge, Titanium Edge SX, Titanium Server, and the Wind River logo are trademarks of Wind River Systems, Inc. Any third-party trademarks referenced are the property of their respective owners. For further information regarding Wind River trademarks, please see:

www.windriver.com/company/terms/trademark.html

This product may include software licensed to Wind River by third parties. Relevant notices (if any) are provided for your product on the Wind River download and installation portals:

<https://delivers.windriver.com/>

<https://windshare.usa.windriver.com/>

Wind River may refer to third-party documentation by listing publications or providing links to third-party websites for informational purposes. Wind River accepts no responsibility for the information provided in such third-party documentation.

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

TABLE OF CONTENTS

1. Introduction.	1
1.1. Overview of the Wind River Diab Compiler Tools Suite.	1
1.2. Wind River Compiler Documentation.	2
1.3. Additional Documentation.	2
2. Environment Variables for Compiler Tools and License.	4
2.1. Introduction.	4
2.2. Setting Path and License Variables.	4
2.2.1. Setting Path and License Variables with wrenv in Windows.	4
2.2.2. Setting Path and License Variables with wrenv in Linux.	5
2.2.3. Setting the Path and License Variables Manually.	6
2.3. Verifying the Installation.	7
2.4. Environment Variables.	7
3. Drivers and Subprogram Flow.	9
3.1. Tools and Drivers.	9
3.2. Subprogram Flow.	9
4. Example.	11
4.1. Introduction to the Example.	11
4.2. Selecting the Target and Compilation Environment.	12
4.2.1. Setting the Default Target.	12
4.2.2. Setting the Target Using Environment Variables.	14
4.2.3. Setting the Target Using -t.	14
4.3. Command Lines.	15
4.3.1. Common Command Line-Options.	15
4.3.2. Compiling the Example.	16
4.4. Compiling for Optimization and Debugging.	17
4.4.1. Summary of Optimization Options.	17
4.4.2. Optimizing the Example.	18

4.5. Linking.	18
4.5.1. Introduction to Linking.	18
4.5.2. Linking the Example.	22
5. Basic Troubleshooting Tips.	29
5.1. Diagnosing and Fixing Compilation Problems.	29
6. Glossary.	32
6.1. Diab Compiler Getting Started: Important Terms.	32

1. WIND RIVER DIAB COMPILER GETTING STARTED, 7.0.6

INTRODUCTION

1. Overview of the Wind River Diab Compiler Tools Suite 套件

The Wind River Diab Compiler is a suite of software development tools for device applications, including optimizing C and C++ compilers, an assembler, a linker, an archiver/librarian, and an ANSI standard C and C++ library.

The tool suite includes high-performance C and C++ compilers built on LLVM technology (www.llvm.org). These tools generate fast, compact code, and they are flexible, with many options to control code generation and assist with porting code developed with other tools.

This guide demonstrates how to configure and use the compiler suites, with examples using the ARM architecture. It includes a detailed example, starting from the choice of the target architecture, continuing with the compilation/assembly stage, optional optimizations, and finishing with linking. The process of linking in embedded systems carries a much higher weight and has much more impact on the product than in desktop application development. To this end we discuss the general concept of linking a bit more in detail.

Portability

C

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

Standard C programs can be compiled with a strict-ANSI options combination, `-ansi -pedantic-errors`, 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.

C++

The Diab C++ compiler 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.

The Diab C++ compiler also 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 defaults to supporting C++14. To select a different standard use the `-std` option, e.g. `-std=c++11`.

Note that the standard library headers provided with Diab do not support using `-std` to set a standard earlier than C++11. However in most cases code written for earlier versions of the C++ standard should work without modification.

GCC

The compiler tools provided by this suite are highly compatible with GCC, and in most cases code written for GCC will carry over without modifications.

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.

2. Wind River Compiler Documentation

The Wind River Diab compiler documentation set explains Wind River language extensions and the operation of all tools.

It does not cover programming language basics, or target machine architectures or instruction sets, though it endeavors to explain areas relevant to embedded systems programming, such as special features of linking for embedded systems.

The Diab documentation is available in PDF and HTML form from the Wind River Knowledge Library (<https://docs.windriver.com>).

PDF versions of the Diab documentation are also included in the Diab installation package and are located at **\$WIND_HOME/docs/diab-7.0.2.0**.

3. Additional Documentation

The following programming language references are recommended:

Programming Language 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.

LLVM and clang

The Diab compiler is based on the LLVM compiler infrastructure, a product of the LLVM Project.

- For information on LLVM, go to <https://www.llvm.org>.
- For information on clang, the LLVM compiler frontend, see the [clang page](#) at llvm.org.

Architecture Documentation

For detailed information about microprocessor architecture and instructions, see the following:

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

Assembler and Linker Documentation

For detailed information about assembler and linker architecture and instructions, see the following:

- [Wind River GNU Compiler](#)

2. ENVIRONMENT VARIABLES FOR COMPILER TOOLS AND LICENSE

1. Introduction

This chapter includes information on how to configure environment variables for compilation (either manually or through an application).

This chapter does not cover the actual installation process. See <http://www.windriver.com/licensing/documents/> for information on how to install Wind River products.

2. Setting Path and License Variables

Before using the compiler, you must set the following environment variables:

Procedure

- Set a *path* variable so that your system knows where to look for the compiler and its associated files
- Set a *license* variable so that the compiler can check for a valid user license
- (Linux only) Set a dynamic library path variable (**LD_LIBRARY_PATH**) so that your system knows where to look for library files that are dynamically loaded.

What To Do Next

License and path variables may be set in two ways: by using the **wrenv** program, or manually.

- [Setting Path and License Variables with wrenv in Windows on page 4](#) The **wrenv** program sets up path and license environment variables on your system for compiler use.
- [Setting Path and License Variables with wrenv in Linux on page 5](#) The **wrenv** program sets up path and license environment variables on your system for compiler use.
- [Setting the Path and License Variables Manually on page 6](#) To set up the path and environment variables manually, do the following:

2.1. Setting Path and License Variables with wrenv in Windows

The **wrenv** program sets up path and license environment variables on your system for compiler use.

About This Task

You may also set up your environment manually. See [Setting the Path and License Variables Manually on page 6](#).

The following code example illustrates the use of **wrenv** to set up the environment on Windows hosts; the installation path (*installDir*) is assumed to be **C:\WindRiver**; *<version>* is the release number of the compiler.

Procedure

1. First, use **wrenv** to create the file **env.bat**:

```
C:\> cd WindRiver
C:\WindRiver> wrenv -p diab-<version> -f bat -o print_env > env.bat
C:\WindRiver> type env.bat
set Path=C:\WindRiver\compilers\diab-<version>\WIN64\bin;C:\WindRiver\compilers\llvm-<version>\WIN64\bin;C:\windows\System64;C:\windows;C:\windows\System64\Wbem
set WIND_PREFERRED_PACKAGES=diab-<version>
set WIND_HOME=C:\WindRiver
set WIND_DIAB_PATH=C:\WindRiver\compilers\diab-<version>
set WRSD_LICENSE_FILE=C:\WindRiver\license
set LM_A_APP_DISABLE_CACHE_READ=set
set WIND_TOOLCHAINS=diab
set LD_LIBRARY_PATH=C:\WindRiver\compilers\diab-<version>\WIN64\bin
```

(Although **wrenv** sets **LD_LIBRARY_PATH**, it's not used in Windows.)

This step only needs to be performed once after installation. Note that **Path** is set using the prevailing **Path** at the time **wrenv** is executed.

2. Once **env.bat** has been created, use it to implement the changes created by **wrenv**:

```
C:\WindRiver> env.bat
```

Parent topic: [Setting Path and License Variables on page 4](#)

2.2. Setting Path and License Variables with wrenv in Linux

The **wrenv** program sets up path and license environment variables on your system for compiler use.

About This Task

You may also set up your environment manually. See [Setting the Path and License Variables Manually on page 6](#).

The following code example illustrates the use of **wrenv** to set up the environment on Solaris/Linux hosts using the **sh** shell; the installation path here is given as **/home/user/WindRiver**.

Procedure

1. First, use **wrenv** to create the file **env.sh**:

```
$ cd /home/user/WindRiver
$ wrenv.sh -p diab-<version> -f sh -o print_env > env.sh

$ cat env.sh
WRSD_LICENSE_FILE="/home/user/WindRiver/license"; export WRSD_LICENSE_FILE;
PATH="/home/user/WindRiver/compilers/diab-<version>/LINUX64/bin:/bin:/usr/bin:/usr/sbin:/usr/local/bin"; export PATH;
WIND_PREFERRED_PACKAGES="diab-<version>"; export WIND_PREFERRED_PACKAGES;
WIND_HOME="/home/user/WindRiver"; export WIND_HOME;
```

```
WIND_DIAB_PATH="/home/user/WindRiver/compilers/diab-<version>"; export WIND_DIAB_PATH;
LM_A_APP_DISABLE_CACHE_READ="set"; export LM_A_APP_DISABLE_CACHE_READ;
WIND_TOOLCHAINS="diab"; export WIND_TOOLCHAINS;
LD_LIBRARY_PATH="/home/user/WindRiver/compilers/diab-<version>/LINUX64/bin:/home/user/Wind
River/compilers/llvm-<version>/LINUX64/lib"; export LD_LIBRARY_PATH;
```

This step only needs to be performed once after installation. Note that **PATH** is set using the prevailing **PATH** at the time **wrenv** is executed. **<version>** shows the release number of the compiler.

2. Once **env.sh** has been created, use it to implement the changes created by **wrenv**:

```
$ . env.sh
```

For users working with the C-shell (**csh**) instead of the **sh**, substitute the following line for its counterparts above:

```
% wrenv.sh -p diab-<version> -f csh -o print_env > env.csh
```

3. Once **env.csh** has been created, source it to implement the changes created by **wrenv**:

```
$ source env.csh
```

Parent topic: [Setting Path and License Variables on page 4](#)

2.3. Setting the Path and License Variables Manually

To set up the path and environment variables manually, do the following:

Procedure

1. Define an environment variable called **WRSD_LICENSE_FILE** that points to the directory containing your license file. In most cases, this directory will be *installDir/license*. **<version>** is the release number of the compiler.
2. Add the tools directory to your **PATH** variable:
 - On Linux, add *installDir/compilers/diab-<version>/LINUX64/bin* to **PATH** and **LD_LIBRARY_PATH**.
 - On Windows, add *installDir\compilers\diab-<version>\WIN64\bin* to **PATH**.

In the following examples, the first example shows modifying paths on a Linux machine using the C shell, and the second example shows modifying a path on Windows:

```
setenv PATH ( /home/user/WindRiver/compilers/diab-<version>/LINUX64/bin $PATH )
setenv LD_LIBRARY_PATH ( /home/user/WindRiver/compilers/diab-<version>/LINUX64/bi
n $LD_LIBRARY_PATH )
```

```
PATH=c:\WindRiver\compilers\diab-<version> \WIN64\bin;%PATH%
```

Parent topic: [Setting Path and License Variables on page 4](#)

3. Verifying the Installation

Complete the following task to verify your installation.

Procedure

1. Once your path and environment variables have been set, enter the following commands.

For C:

```
% dcc -tARMV8AMH:simple -VV
```

For C++:

```
% dplus -tARMV8AMH:simple -VV
```

If correctly installed, the compiler will invoke the various subprograms from the front-end through the linker and each will show its version on **stdout**.

2. If you get the following message, check your path:

```
Bad command or file name
```

Ensure that the compiler **bin** directory is in the **PATH**, as described in [Setting Path and License Variables on page 4](#).

3. If you get the following message, and the tools are resident on a different machine from the machine on which you are executing, be sure that your **PATH** variable includes the location of the tools on the machine where they are installed.

```
can't find compiler component
```

4. For Windows, map a drive letter to the directory where the tools reside and use the drive letter to set the path.


4. Environment Variables

The compiler makes use of a set of environment variables that can be used to modify its default behavior.

Default Environment Configuration and Overrides

The configuration information which controls default operation of the tools is usually stored as configuration variables in **default.conf** in the **conf** subdirectory of the *versionDir* directory by the dctrl program. These configuration variables include **DTARGET**, **DFP**, **DOBJECT**, and **DENVIRON**.

However, if an environment variable having the same name as a configuration variable is set, the value of the environment variable will override the value stored in **default.conf**. (This can in turn be overridden by using a **-t** or **-WD** option on the command line when invoking a tool.)

 **Note:** Overriding default settings is for unusual cases. It is usually sufficient to override the default setting by using the **-t** option on a command line when invoking a tool, or to use one of the other methods, all as described in [About Target Configuration](#) in the *Wind River Diab Compiler User's Guide*.

Environment Variables Recognized by the Compiler Tools

DCONFIG

Specifies the configuration file used to define the default behavior of the tools. If neither **DCONFIG** nor the **-WC** option is used, the drivers use **versionDir/conf/dtools.conf**.

DTARGET

DOBJECT

DFP

DENVIRON

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

For more information see the *Wind River Diab Compiler User's Guide*.

DFLAGS

Specifies extra options for the drivers and is a convenient way to specify **-XO**, **-O** or other options with an environment variable (e.g., to avoid changing several makefiles or to override options given in a configuration file). The options in **DFLAGS** are evaluated before the options given on the command line.

DIABTMPDIR

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

3. DRIVERS AND SUBPROGRAM FLOW

1. Tools and Drivers

The primary tools in the Wind River Diab Compiler suite are:

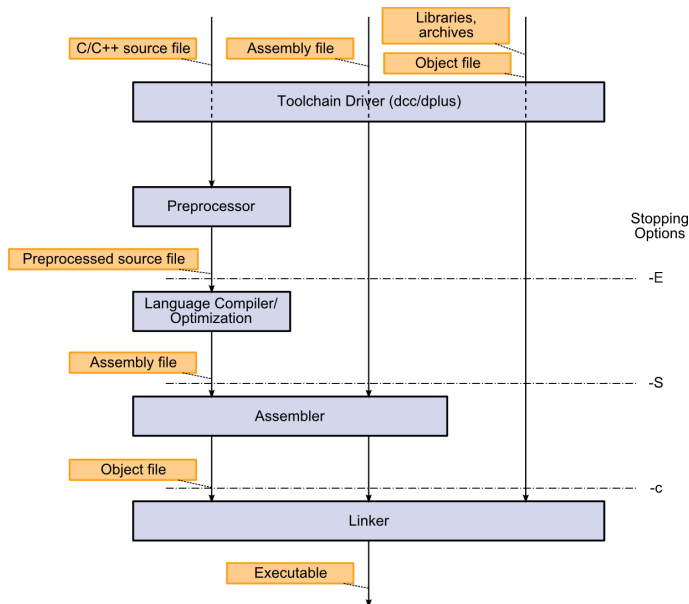
- C compiler (clang, invoked by the **dcc** driver)
- C++ compiler (also clang, invoked by the **dplus** driver)
- Assembler. The clang built-in assembler for standalone assembly files and inline assembly in C/C++. For standalone assembly files, the GNU assembler as<arch> (e.g., asarm or asarm64) can also be used if the -Xgas option is used.
- Linker. By default, the Diab linker dld. The GNU linker ld<arch> (e.g., ldarm or ldarm64) is also supported.

The tools are most easily invoked using the **dcc** and **dplus** driver programs (often referred to simply as drivers). Depending on the input files and options on the command line, the driver may run up to four subprograms: the C preprocessor, the compiler, the assembler, and the linker.

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.

2. Subprogram Flow

The following image shows the subprogram flow.



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

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

Table 1. Driver Subprograms , Default Input and Output Extensions, and Stopping Options

Sub-program	Default Input Extension	Default Output Extension	Function and Stopping Option
clang integrated preprocessor			The preprocessor; takes a C or C++ module as input and processes all # directives. This program is included in the main compiler program. The -E -o option halts the driver after this phase, producing a file with the .i suffix. (The .i file is not produced unless -E -o is used.) Stopping option: -E
clang	.c .cpp .cxx .cc .C (capital, UNIX)	.s	The C- and C++-to-assembly compiler; generates assembly source from preprocessed C and C++ source. Stopping option: -S
as<arch> (e.g., asarm, asarm64)	.s	.o	The GNU (as<arch>) . Stopping option: -c
dld ld<arch> (e.g., ldarm, ldarm64)	.o (object) .a (archive) .dld (commands, Diab) .link (commands, GNU)	a.out	The Diab (dld) or GNU (ld<arch>) linker; generates an executable file from one or more object files and object libraries, as directed by a linker command file. The default output name is a.out if the -o outfile option is not given.


4. EXAMPLE

1. Introduction to the Example

The example program is a simple bubble sort that displays the following features of the tool suite:

- simple commands for compiling, assembling, and linking
- linker command file usage for laying out memory
- locating of individual C and assembler functions in specific areas of memory
- locating of an I/O register at an absolute location during linking
- startup and termination, including copying initial values from read-only memory to random-access memory
- mixing of separate C and assembly-language modules
- using inline assembly within a C module

The source files are installed in sub-directories of *installDir/compilers/diab<version>/example*, in a separate directory for each installed architecture.

 **Note:** This guide uses the 32-bit ARM example. The installation also contains an example for 64-bit ARM.

Example Program Source Files

Source files for the example program are in the following directory, unless otherwise indicated:

installDir/compilers/diab<version>/example/arm

The example uses the following files:

bubble.c	Includes the <code>main()</code> and <code>sort()</code> routines, a small <code>get_short()</code> assembly-language macro function, and usage of a simulated "I/O register."
crt0.s	<p>The "startup" module. The example uses a modified version of <i>installDir/compilers/diab<version>/src/crtarm/crt0.s</i>. The modified version is <i>installDir/compilers/diab<version>/example/arm/crt0.s</i>. It contains architecture-specific assembler code to perform operations such as initializing the stack pointer. It then calls <code>__init_main()</code> in init.c to continue initialization.</p> <p>Each supported architecture has its own crt0.s, in sub-directories of <i>installDir/compilers/diab<version>/src</i>. The modified versions for each architecture for this example are in the sub-directories of <i>installDir/compilers/diab<version>/example/<arch></i>.</p>
init.c	Located in the directory <i>installDir/compilers/diab<version>/src</i> , this program performs basic initialization such as loading data into RAM, then calls the <code>main()</code> routine in bubble.c . init.c is pre-compiled for installed architectures; the object file init.o is loaded from a library archive file.

swap.s	Assembly module containing the swap() routine.
bubble.dld	Linker command file. Includes directive to define a simulated "I/O" register used by bubble.c and crt0.s .
makefile	A makefile suitable for both dmake and gmake to build the example

This guide contains extracts from the source files. Refer to the files themselves for complete details.

2. Selecting the Target and Compilation Environment

Complete this task to select the target and compilation environment.

About This Task

For full information about selecting targets, see [About Target Configuration](#) in the *Wind River Diab Compiler User's Guide*.

Procedure

- To compile a program to run on a particular device, specify the following configuration attributes:
 - the processor running the application
 - the type of object file format to use
 - the type of floating-point support to use, if any
 - the execution environment (i.e., the defaults for **crt0.o**, the linker script, and the library search paths) to use

Together, these attributes constitute the *target*.

- Select the target when invoking the compiler, assembler, or linker in one of three ways:
 - Run **dctrl -t** to change the default target. See [Setting the Default Target on page 12](#). This has the advantage of showing you all of the available target options and, if you don't change targets frequently, removes the need to specify target parameters for each compilation.
Configuring with **dctrl -t** changes the default for all users, and requires write access to the *installDir/conf* directory.
 - Set environment variables. See [Setting the Target Using Environment Variables on page 14](#). Environment variables, if set, override the default configuration.
 - Specify the target using options to the compiler tools. See [Setting the Target Using -t on page 14](#). This is a good approach when different users work with different targets. Command-line options override both the default configuration and environment variables.
- [Setting the Default Target on page 12](#)The **dctrl** command is a utility for configuring the toolchain. The **dctrl -t** option sets the default target attributes.
 - [Setting the Target Using Environment Variables on page 14](#)Override default values using environment variables.
 - [Setting the Target Using -t on page 14](#)Use the **-t of environ** option to explicitly specify the target configuration for any invocation of the compiler tools, and override the default configuration or any environment variables.

2.1. Setting the Default Target

The **dctrl** command is a utility for configuring the toolchain. The **dctrl -t** option sets the default target attributes.

Procedure

1. To run **dctrl**, open a command line and set environment variables, as described in [Setting Path and License Variables on page 4](#).
2. Run **dctrl** from the command line.

The following is an example.

Note that **dctrl** will only display valid configuration options. For example, if a given processor does not support COFF object files, that will not be available as an option.

```
$ dctrl -t
Possible architectures:
  1) ARM
  2) ARM64

Select default architecture? (1-2)[1] 1

Possible devices:
  1) ARM          2) ARMT2          3) ARMV7A          4) ARMV7AT2
  5) ARMV7R       6) ARMV7RT2       7) ARMV8A          8) ARMV8AT2
  9) ARMCORTEXA53 10) ARMCORTEXA53T2 11) ARMV8R         12) ARMV8RT2
 13) ARMCORTEXR52 14) ARMCORTEXR52T2

Select device or core? (1-14)[1] 3

Possible object formats:
  1) ARM AAPCS ABI, little-endian code, big-endian data
  2) ARM AAPCS ABI, little-endian code, little-endian data

Select object format? (1-2)[1] 2

Possible floating point modes:
  1) Software Floating Point
  2) No Floating Point
  3) Hard float: VFPv4-D32 and Advanced SIMD/NEON

Floating point mode "None" can be used to reduce the library footprint
of programs that do not use floating point.

Select float point mode? (1-3)[1] 3

Possible environments:
  1) cross      - Use Ram Disk for I/O
  2) simple     - Only character I/O
  3) other      - Enter text string when requested

The environment controls which libraries are linked by default.
It also specifies an {environ}.conf referred to by dtools.conf.

Select environment? (1-5)[1] 2

Selected default compilation mode is: -tARMV7AMH:simple
```

The compilation mode is displayed in the form **-t~~of~~environ**, which can be used as a command-line option to the compiler tools. See [Setting the Target Using -t on page 14](#).

The configuration is saved in the `installDir/diab/releaseDir/conf/default.conf` file.

Running **dctrl -t** again will write new values to **default.conf**.

This configuration is used for all invocations of the compiler tools, unless it is overridden by setting environment variables or command line options. See [Setting the Target Using Environment Variables on page 14](#) and [Setting the Target Using -t on page 14](#).

Parent topic: [Selecting the Target and Compilation Environment on page 12](#)

2.2. Setting the Target Using Environment Variables

Override default values using environment variables.

About This Task

The default values for target processor, object file format, floating point support, and execution environment are set using the **dctrl -t** command. See [Setting the Default Target on page 12](#).

Procedure

1. Override any of these values using environment variables.
 - To override the target processor, set the **DTARGET** variable.
 - To override the object file format, set the **DOBJECT** variable.
 - To override the floating point support, set the **DFP** variable.
 - To override the execution environment, set the **DENVIRON** variable.
2. To determine valid values for any of these environment variables, run the **dctrl -t** command. For more details, see the *Wind River Diab Compiler User's Guide*.

Parent topic: [Selecting the Target and Compilation Environment on page 12](#)

2.3. Setting the Target Using -t

Use the **-t of:environ** option to explicitly specify the target configuration for any invocation of the compiler tools, and override the default configuration or any environment variables.

Procedure

1. Use the **-t of:environ** option with any of the following commands:
 - `dcc` driver for C
 - `dplus` driver for C++
 - `dld` linker

For example, the following compile command sets the target configuration explicitly:

```
% dcc -tARMCORTEXA53MH:simple test.c
```

The parts of **-t of:environ** are as follows:

- *t* is the target processor, **ARMCORTEXA53** in the example above.
 - *o* is the object file format, **M** (ARM AAPCS ABI, little-endian code, little-endian data) in the example above.
 - *f* is the floating point support, **H** (Hard float: VFPv4-D32 and Advanced SIMD/NEON) in the example above.
 - *environ* is the execution environment, **simple** in the example above.
2. To determine valid *-t* option combinations, invoke the **dctrl -t** command at a command prompt and answer the prompts.
- It ends by showing the *-t* option for those answers. Note that this will change the default values for the installation.
3. To see the current default configuration, run the following command:

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

Parent topic: [Selecting the Target and Compilation Environment on page 12](#)

3. Command Lines

The tools run from the command line. On Windows, open a development shell. On UNIX or Linux, ensure that the environment variables are properly set.

For details, see [Setting Path and License Variables on page 4](#).

The general form for invoking the compiler, assembler, and linker is:

```
tool [options] input-files
```

Options begin with a hyphen "-". For the compiler and linker, options and input files may be mixed in any order on the command line unless one option depends on another (for example, the *-L* option sets a path for use by the *-l* option and so must appear before the *-l* option). For the assembler, *input-files* must be last.

Options are of the form *-c*, *-c name*, *-c value*, or *-c name=value*, where *c* is always a single option letter.

Rules for writing options:

- All characters in an option are case-sensitive: for example, the compiler option *-O* means optimize, while *-o file* sets the output file. This is true even on Windows, except for path and filenames, which are case-insensitive.
- No spaces are allowed within options except immediately after the option letter. For example **-D DEBUG=2** is valid, but **-DDEBUG = 2** is not because there are spaces on either side of '='. Generally, spaces are not used. In this guide, a space may be used for easier reading.
- When an option appears more than once, the final instance is used.

As noted in [Common Command Line-Options on page 15](#), the forms *-@name* and *-@@name* may be used repeatedly on a command line to provide options indirectly through an environment variable or file.

- [Common Command Line-Options on page 15](#) The following command line options may be used with all tools:
- [Compiling the Example on page 16](#) The tools can either be called by the driver program (dcc or (dplus) or they can be called as separate, standalone programs (for example, dld).

3.1. Common Command Line-Options

The following command line options may be used with all tools:

-?

-h

--help

Show a synopsis of command-line options.

-o *file*

Direct the output file to the given file instead of the default for the tool. The file can include an absolute or relative path.

-V

Display the current version number of the tool suite.

-@*name*

Read command-line options from either an environment variable or file. When -@*name* is encountered on the command line, the driver first looks for an environment variable with the given *name* and substitutes its value. If an environment variable is not found then the driver tries to open a file with given *name* and substitutes the contents of the file. The name can include an absolute or relative path.

In a file, arguments to the command may be written one or more per line. Comments are not allowed in the command file.

-@@*name*

Same as -@*name* but also shows all command line options on **stdout**.

-@E=*file*

-@E+*file*

Redirect any output to standard error to the given file. Use "+" instead of "=" to append to the file.

-@O=*file*

-@O+*file*

Redirect any output to standard output to the given file. Use "+" instead of "=" to append to the file.

Parent topic: [Command Lines on page 15](#)

3.2. Compiling the Example

The tools can either be called by the driver program (dcc or dplus) or they can be called as separate, standalone programs (for example, dld).

About This Task

The separate programs will only execute their specific task, whereas the driver will always try to build an executable from all the sources provided to it (unless a certain stop-command is provided—see section [Subprogram Flow on page 9](#)). To this end the driver will use certain defaults if inputs are missing (such as a linker command file or startup code). This allows for either a one-step compile, assemble, and link, or individual single steps for each intermediate file.

The complete executable can be built from the example file **bubble.dld**; in this case, we show the individual steps, rather than a one-step compile.


Procedure

Enter following series of commands

```
% dcc -tARMV7AMH:simple -c -Xpreprocess-assembly crt0.s
% dcc -tARMV7AMH:simple -c -o bubble.o bubble.c
% dcc -tARMV7AMH:simple -c swap.s
% dld -tARMV7AMH:simple -m6 -o bubble.out crt0.o bubble.o swap.o -lc -limpl bubble.dld
```

As a result of running these commands:

1. First, the driver program takes **crt0.s**, preprocesses this file and then passes it to **asarm** to create **crt0.o** out of it (the assembler and compiler output defaults to **name.o** for **name.s** or **name.c**).
2. **bubble.c** is compiled into **bubble.o**. Note the stop command **-c**, provided to the **dcc** driver, which leads to a process stop after compilation (otherwise **dcc** would try to link and create an executable).
3. The assembler creates **swap.o** from **swap.s**.
4. The linker links all the object files into an executable, using the same libraries and the linker command file as in the one-step command.

 Note: Generally, most projects use a tool that generates the required command lines (e.g. **dmake**).

Parent topic: [Command Lines on page 15](#)

4. Compiling for Optimization and Debugging

The compiler implements many distinct optimization techniques, including numerous inter-procedural optimizations, as well as optimizations that are highly specific to each supported target. Advanced global optimizations across multiple functions include partial inlining of functions.

- [Summary of Optimization Options on page 17](#) The following table summarizes the main options used to generate optimized and debuggable code.
- [Optimizing the Example on page 18](#) Run the following example to observe the effects of optimization.

4.1. Summary of Optimization Options

The following table summarizes the main options used to generate optimized and debuggable code.

See [Common Command Line-Options on page 15](#), for more details on these options.

Desired Effect	Options
Optimize for speed	-O -O2 -O3
Optimize for speed and size	-Os
Add debugging	-g

Parent topic: [Compiling for Optimization and Debugging on page 17](#)

4.2. Optimizing the Example

Run the following example to observe the effects of optimization.

About This Task

Due to the simplicity of our example, its optimization potential is limited; however, it's still possible to observe the effects of optimization. Let's compare the resultant size of unoptimized code to the size of optimized code.

Procedure

1. First, check the code size without optimization.

The following command

```
% % dcc -tARMV7AMH:simple -c bubble.c ; sizearm bubble.o
```

gives us the following output (actual numbers may vary):

```
text data bss dec hex filename
655 40 4 699 2bb bubble.o
```

2. Repeat the compilation, this time using the `-Os` option to optimize:

```
% % dcc -tARMV7AMH:simple -c -Os bubble.c ; sizearm bubble.o
```

which yields:

```
text data bss dec hex filename
360 40 4 404 194 bubble.o
```

Results

We can see a decrease in overall code size from 655 bytes initially down to 360 bytes, for a total reduction of 295 bytes, a decrease of about 45%.

Parent topic: [Compiling for Optimization and Debugging on page 17](#)

5. Linking

5.1. Introduction to Linking

The *linker* is a program that combines one or more binary *object modules* produced by compilers and assemblers into one binary *executable file*. It may also write a text map file showing the results of its operation.

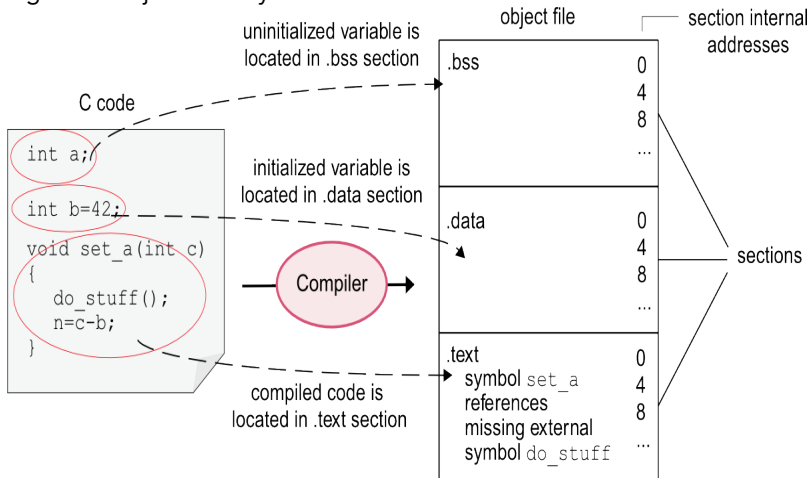
Each object module or object file is the result of one compilation or assembly. Object files are either standalone (typically with the extension `.o`) or are collected in archive libraries, also called libraries (typically with the extension `.a`).

An object module contains sections of code and data, with names such as “.text,” (containing code), “.data” (containing variables having initial values), and “.bss” (containing uninitialized variables), as well as various housekeeping sections containing things like symbol table and debug information. *Data sections* are located in writable memory because their contents can change.

Initialized data must be managed differently than uninitialized data. Values for initialized data must be stored as part of the executable image, then copied to RAM when the program loads. There is no need to store values for uninitialized data, but the memory locations in RAM should be zeroed when the program loads.

Figure 1. Object File Layouts on page 19 depicts how the compiler places variables and the code of a C source file into the object file.

Figure 1. Object File Layouts



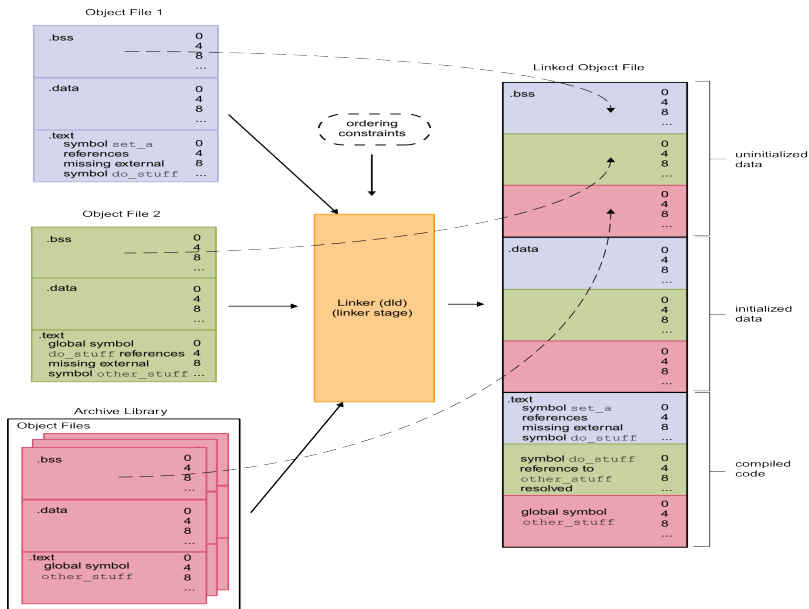
The compiler and assembler create object files with code and data divided into separate sections. These are input sections, used as input into the linker. The linker processes the object files and their sections, performing any necessary relocation, and creates an executable image. Sections in the executable image are output sections because they are the output from the linker. Often an input section will be relocated into an output section with the same name. For example, a **.text** input section normally goes into a **.text** output section.

The linker reads the sections from the object modules input to it, and based on command-line options and a *linker command file*, combines these input sections into output sections, and writes an executable file. (That is the usual case; it's also possible to output a file that can be linked again with other files in a process called *incremental linking*.)

A section may contain a reference to a symbol not defined in it—an *undefined external*. Such an external must be defined as **global** in some other object file. A **global** definition in one object file may be used to satisfy the undefined external in another.

Figure 2. Combining Object Files on page 19 depicts how two object files and a library are used by the linker to form a single object file, demonstrating that the undefined references are resolved. The ordering constraints are provided to the linker in a linker command file. They tell the linker which input sections shall form an output section. In Figure 2. Combining Object Files on page 19 the linker simply takes all input sections with the same name and gathers them into an output section of the same name. This is illustrated using the **.bss** section. You can see that the three different **.bss** sections from the input object files form the **.bss** section of the output object file. Also note that the section-relative internal addressing has not yet been modified. (Some very special symbols, namely the symbols that represent the start and end addresses of sections, are still undefined. They are only defined by the linker once the final relocation has been completed.)

Figure 2. Combining Object Files



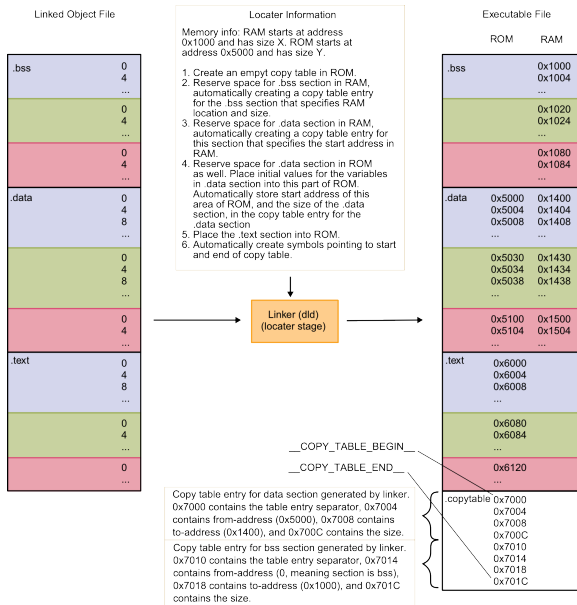
When code is compiled or assembled into an input object file, the first byte of each input section is typically at address 0. (You can see this in both [Figure 1. Object File Layouts on page 19](#) and [Figure 2. Combining Object Files on page 19](#).) Within each section, the addressing restarts at 0.

However, in the real memory, there is just one address 0, and—depending on your memory map—address 0 may or may not point to valid memory. As a consequence, the linker must modify all the addresses within the sections to reflect real-world memory addresses. When doing this, any absolute addressing within a section will become “wrong” (unless the section will really be placed at address 0 in real memory) and will require relocation. The input object file contains sections of relocation information which the linker will use to adjust such absolute references. Relocation information is used to make other similar adjustments as well.

The decision where to locate the section in memory depends on various constraints, such as the memory layout of your embedded system, security requirements, or performance and power considerations. Depending on where in the actual physical memory certain sections are placed, access to the sections can be faster or slower, more or less power consuming, or their protection against unwanted accesses can be better or worse. All these things have to be taken into consideration when defining the memory locations for the section. The linker cannot automatically make such decisions and hence an engineer has to provide the required information as a linker command file.

[Figure 3. Linker Relocations on page 20](#) depicts how the linker does the relocation using the information from the linker command file. In this illustration, the linker command file is intentionally not depicted using linker command language; rather, the figure uses natural language to illustrate what kind of information is provided to the linker. Details on how to encode that information into the linker command language can be found in [Linking the Example on page 22](#). (Note that [Figure 3. Linker Relocations on page 20](#) is a simplification of the real process.)

Figure 3. Linker Relocations



Given the definitions above, in the abstract, the linking process consists of the following steps:

1. Read the command line and linker command file for directions.
2. Read the input object files and combine the input sections into output sections per the directions in the linker command file. Globals in one object file may satisfy undefined externals in another.
3. Search all supplied archive libraries for modules which satisfy any remaining undefined externals.
4. Locate the output sections at specific places in memory per the directions in the linker command file.
5. Use the relocation information in the object files to adjust references now that the absolute addresses for sections are known.
6. If requested, write a link map showing the location of all input and output sections and symbols.

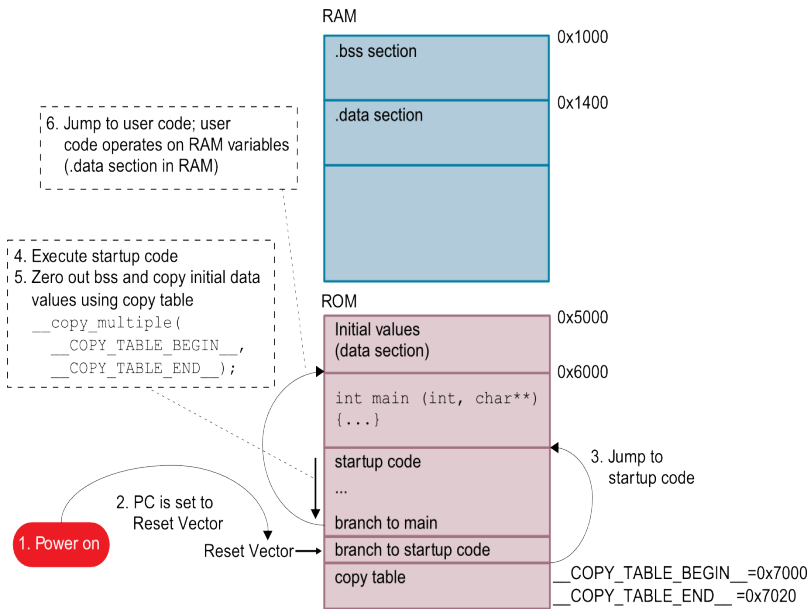
Now, one may ask why the data section exists twice, once in RAM, once in ROM. To explain this, we first need to take a look on how an embedded system gets from being powered off to executing your code.

Figure 4. Powering On an Embedded System on page 21 depicts the general concept of powering on an embedded system: once power is supplied (or the system is reset), the program counter (PC) of the processor jumps to a certain location, the so called reset vector. (In a SMP multicore scenario, there is usually just one processor active at power on, and the others are kept disabled until bootup is complete.) The reset vector usually points to a location in a non-volatile memory, so that the processor can find valid instructions there, even after long power-off phases. At the reset vector, the processor will either find its startup code, or at least a branch instruction to the startup code (depending on the embedded system and the kind of processor used).

So what is the startup code? The startup code is special code, usually provided to you by your board vendor, that is custom-tailored for the board and processor you are using. The startup code configures the minimal set of processor and peripheral configuration registers required to actually execute user code. Such a configuration includes setting the PLL configuration for a proper clock rate, chip selects to allow access to certain off-chip memories, and basic MMU configurations. This startup code is usually small, so that it fits into small non-volatile memories.

The final thing the startup code does is to branch to the entry point of your user code (in C that would be `main()`). The startup code needs to be provided as an object file to the linker, and the linker command file needs to ensure that the code ends up in some non-volatile memory and that the very first instruction of the startup code (or a branch to it) will be mapped to the reset vector. Obviously, the startup code and locator information are closely related and intertwined.

Figure 4. Powering On an Embedded System



Now back to the initial question: Why does the **.data** section exist twice? The reason is that the **.data** section contains variables that have initial values. In case of reset, the values have to be assigned properly so they need to be in a non-volatile memory, which is usually ROM. However, they also need to be changeable during normal operation, so they can't be in ROM; they need to be in RAM. To solve this, the ROM contains the initial values (that is the actual content of the data section), but as shown in [Figure 4. Powering On an Embedded System on page 21](#), the linker reserves the same amount of memory in RAM.

How do the initial values get from ROM to RAM? This is where the startup code comes into play. To actually perform this task, the startup code needs to know where the initial values are, where to put them, and how many bytes it has to copy. To this end, the linker created the aforementioned copy table, so that the startup code can use a compiler provided function named `__copy_multiple()` together with the linker-generated start and end symbols of the copy table. This function processes each entry in the copy table, copying the initial values from the from-address to the to-address stored in the copy table. If the from-address is zero, the RAM is zeroed out (instead of copying) because it is a bss section.

Note that the **data** section is just one example. Similar mechanisms can (and have to) be applied to other sections as well.

Related information

[Linking the Example on page 22](#)

5.2. Linking the Example

Now that we are through the theory, let's look how things really work with the compiler and linker in our fancy bubble example.

Memory Layout

For the purposes of our example, let's give the target device the following characteristics:

- Read-only memory (ROM) starting at address 0x0, with size of 0x40000. This contains the program code and starting values for initialized data. The ROM is divided into two areas (**rom1** and **rom2**). This shows techniques for loading code and data into specific locations in memory. Memory below 0x20000 is reserved for the debug monitor.
- Random-access memory (RAM) starting at address 0x80000, with size of 0x40000. This contains the run-time data, including initialized variables, uninitialized variables, the stack, and the heap.
- An I/O register at address 0xf0000. For this example, we define a symbolic name for a memory mapped I/O register in the linker command file.

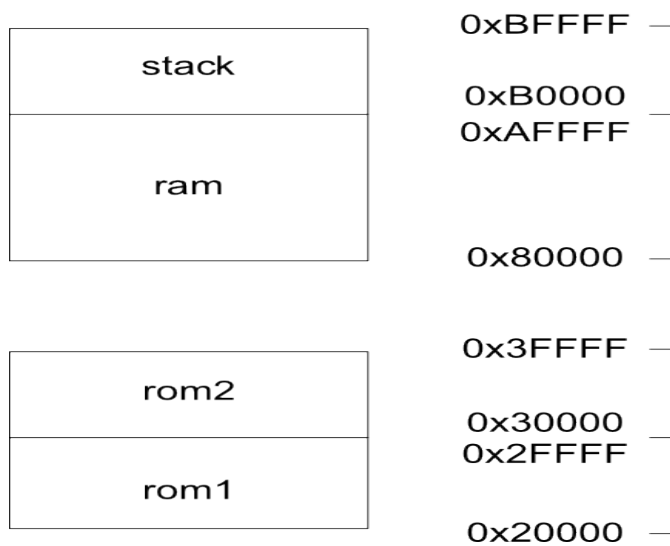
The MEMORY command in the linker command file identifies areas in device memory. Other commands locate code and data into the named memory areas.

For example, **bubble.dld** contains the following:

```
MEMORY
{
    rom1:  org = 0x20000, len = 0x10000 /* 3rd 64KB */
    rom2:  org = 0x30000, len = 0x10000 /* 4th 64KB */
    ram:    org = 0x80000, len = 0x30000 /* 512KB - 703KB */
    stack:  org = 0xb0000, len = 0x10000 /* 7043B - 768KB */
}
```

This identifies four areas in target memory, named **rom1**, **rom2**, **ram**, and **stack**. The command defines the origin and length of each area, creating the memory layout shown in [Figure 5. Memory Layout on page 23](#):

Figure 5. Memory Layout
Memory Areas



Default Text Section

A text section contains executable code or constant data. Text sections are generally located in read-only memory because their contents do not change.

By default, the compiler creates an input section named **.text** that contains code and constant data. For example, in **bubble.c**, the executable code from the `main()` routine and the `get_short()` assembler routine are placed in the default **.text** section.

In **crt0.s**, the **.text** directive also places code from the assembler statements into the **.text** section.

The linker command file **bubble.dld** contains instructions to collect the text sections from the object files. For example, the following instruction combines all input sections labelled **.text**, **.init**, or **.fini** into an output section labelled **.text**.

```
.text : { *(.text) *(.rodata, ".rodata.*") *(.init) *(.fini) }
```

The file also contains the following instructions to collect other read-only sections. Note that the linker generated copy table section (**.copytable**) is included in these sections. The sections are not used directly in the bubble program, but they are used in library routines that are linked with it.

```
.ctors (=TEXT) ALIGN(4) : { ctordtor.o(.ctors) *(.ctors) }
.init_array (=TEXT) ALIGN(4) : { ctordtor.o(.init_array) *(.init_array) }
.dtors (=TEXT) ALIGN(4) : { ctordtor.o(.dtors) *(.dtors) }
.fini_array (=TEXT) ALIGN(4) : { ctordtor.o(.fini_array) *(.fini_array) }
.sdata2 : {}
.copytable : {} /* the copytable */
```

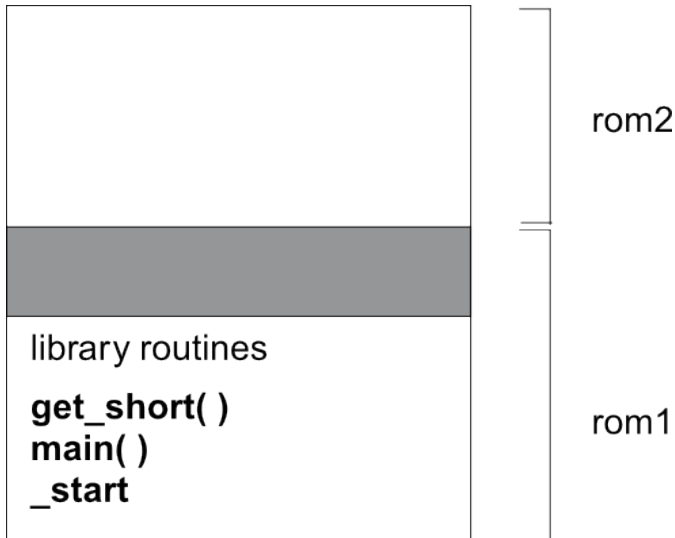
The GROUP command combines the read-only sections into a single group, which it locates in the rom1 memory area. The full command is as follows:

```
GROUP : {
    .text : { *(.text) *(.rodata, ".rodata.*") *(.init) *(.fini) }
    .text : { *(.text) *(.rodata) *(.init) *(.fini) }
    .ctors (=TEXT) ALIGN(4) : { ctordtor.o(.ctors) *(.ctors) }
    .init_array (=TEXT) ALIGN(4) : { ctordtor.o(.init_array) *(.init_array) }
    .dtors (=TEXT) ALIGN(4) : { ctordtor.o(.dtors) *(.dtors) }
    .fini_array (=TEXT) ALIGN(4) : { ctordtor.o(.fini_array) *(.fini_array) }
    .sdata2 : {}
    .copytable : {} /* the copytable */
} > rom1
```

This results in the following being placed in the rom1 area in memory:

- **_start** (from crt0.s)
- main()
- get_short()
- various library routines
- small constants
- the copy table

Figure 6. Routines in the rom1 Area



User-defined Sections

For more control over how memory is used, you can define additional sections, assign selected code or data to the sections, and locate them at specific addresses.

The compiler uses pre-defined or user-defined section classes to determine what section to use for code or data. Two of the pre-defined classes are **CODE**, for executable code, and **DATA**, for static and global variables.

By default, everything in the **CODE** section class is assigned to the **.text** input section. Initialized items in the **DATA** section class are assigned to the **.data** input section, and uninitialized items in the **DATA** section class are assigned to the **.bss** input section.

Section classes also contain *attributes* that define the access mode and addressing mode for items in the section class. See the *Wind River Diab Compiler User's Guide* for details on attributes.

To change the attributes for a section class, use `__attribute__((section("name")))`. For example, the following, from **bubble.c**, assigns generated code to the **.text2** input section:

```
__attribute__((section(".text2")))
```

The `sort()` routine, which has the `__attribute__`, is placed in the **.text2** section.

The following assembler directive, from **swap.s**, performs a similar function. It places the subsequent code into the **.text2** input section and identifies it as code.

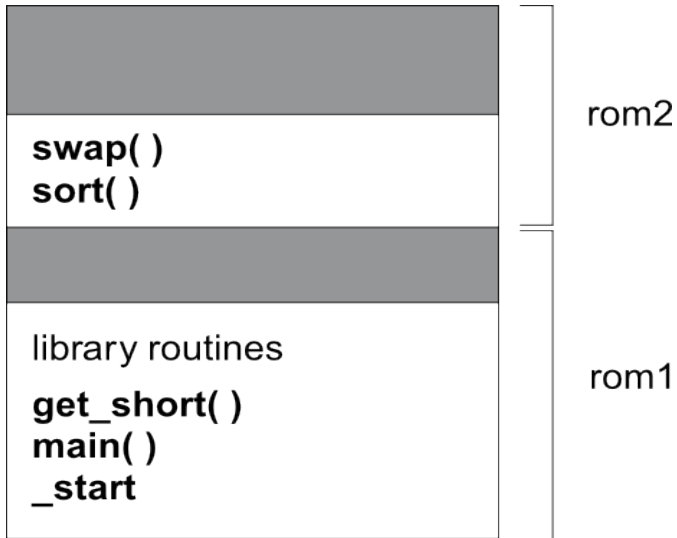
```
.section .text2,"ax",%progbits
```

The linker command file, **bubble.ld**, collects all the input sections named **.text2** into an output section also named **.text2**. This section is located in the **rom2** memory area.

```
.text2 : { *(.text2)
} > rom2
```

This results in the `sort()` and `swap()` routines being located in **rom2**, as shown in [Figure 7. Routines in the rom2 Area on page 25](#).

Figure 7. Routines in the rom2 Area



Initialized Data

The initial values for initialized data are stored in ROM and copied to RAM as part of the start-up sequence.

For example, the following array defined in **bubble.c** has initial values:

```
int array[] = {
10, 9, 8, 7, 6, 5, 4, 3, 2, 1
};
```

By default, the compiler places this in a **.data** input section in the object file.

When it builds the executable file, the linker must perform the following operations:

- Save the initial values for the data into the image.
- Create a copy table entry that contains the location of the initial values in ROM, the data location in RAM, as well as the size of the data section.
- Calculate the relocation values so references to data point to the correct locations in RAM.

On program initialization, the startup program copies the values into the correct locations in RAM.

The following linker commands in **bubble.dld** contain all the required information to perform the tasks (ellipses indicate code eliminated from this example for clarity):

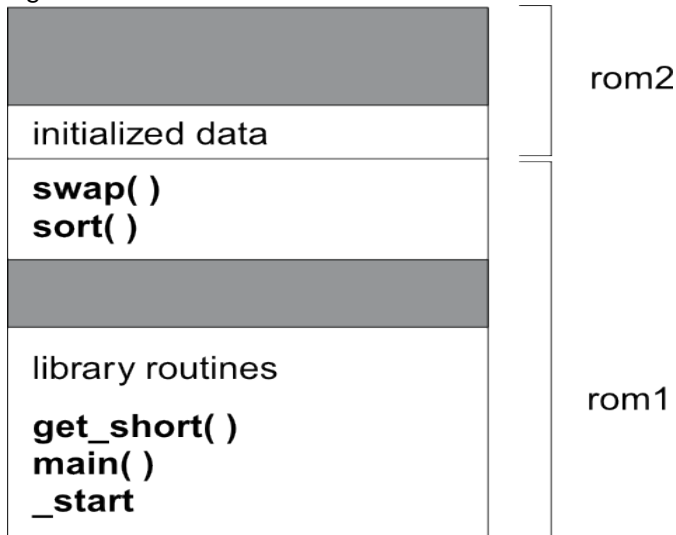
```
GROUP LOAD (>rom2) COPYTABLE : {
    .data (DATA) : {}
    .sdata (DATA) : {}
    ...
} > ram
```

The group is placed in RAM (note the ">ram" as the location for the group.), so references to any symbol within this group point to RAM. The line "LOAD (>rom2) COPYTABLE", means that a copy of each non-bss section in that group will be placed in ROM (that is, into area rom2, and adjacent to whatever has already been placed in rom2), in exactly the same order as in this group. Each non-bss section automatically gets a copy table entry containing the start address of the section copy in ROM, the start address of the section in RAM, and the section size. For a data section, the copy of the section is its original state; that is, it contains the initial values.

When the program loads, the initialization program, **init.c**, copies the data values from ROM to RAM as part of the processing of the copy table:

```
__copy_multiple(__COPY_TABLE_BEGIN__, __COPY_TABLE_END__);
```

Figure 8. Initialized Data in the rom2 Area



Uninitialized Data

For uninitialized data, there is no need to save the values in ROM. In the bubble example, the uninitialized data is part of the same group as the initialized data, but identified as BSS, and so the linker reserves space in RAM and sets symbols to point to the reserved space. It will not, however, create a copy of this section in ROM. The copy table entry contains address zero as the ROM location, which tells `__copy_multiple()` that this is a bss section, which does not need to be copied. Instead, `__copy_multiple()` simply zeros out the complete reserved RAM area.

```
GROUP LOAD (>rom2) COPYTABLE : {
    [...]
    .sbss (BSS) : {}
    .bss (BSS) : {}
} > ram
```

Small Data Area (SDA)

Some architectures, including PowerPC, support the use of a *small data area*. This area can be addressed using a base register and an offset, resulting in single-word instructions for memory access.

The compiler puts data into the small data area depending on the size of the data. The `-Xsmall-data` compiler option sets the size of data elements that can go into the small data area.

The linker automatically defines the symbols **_SDA_BASE_** and **_SDA2_BASE_**. **_SDA_BASE_** contains the value of the base register for small variables and **_SDA2_BASE_** contains the value of the base register for small constants. These values are computed at link-time by adding an offset to the starting address of the **.sdata** (small variable) and **.sdata2** (small constant) sections. The startup code in **crt0.s** stores these values in the appropriate registers.

Note that the small initialized data (**.sdata**) is located at the top of the initialized data section, and the small uninitialized data (**.sbss**) is located at the bottom of the uninitialized data section. This ensures that **.sdata** and **.sbss** are contiguous and can both be addressed using the same base register.

For more details, see the *Wind River Diab Compiler User's Guide* and the **crt0.s** file for your specific architecture.

Stack and Heap

Space for the stack is allocated directly in **bubble.dld**. The MEMORY command defines the stack origin and size:

```
stack: org = 0xb0000, len = 0x10000
```

The initial stack pointer values are also set in **bubble.dld**. The stack grows down, so the initial value is the highest available address.

```
__SP_INIT = ADDR(stack ) + sizeof(stack );
__SP_END = ADDR(stack );
```

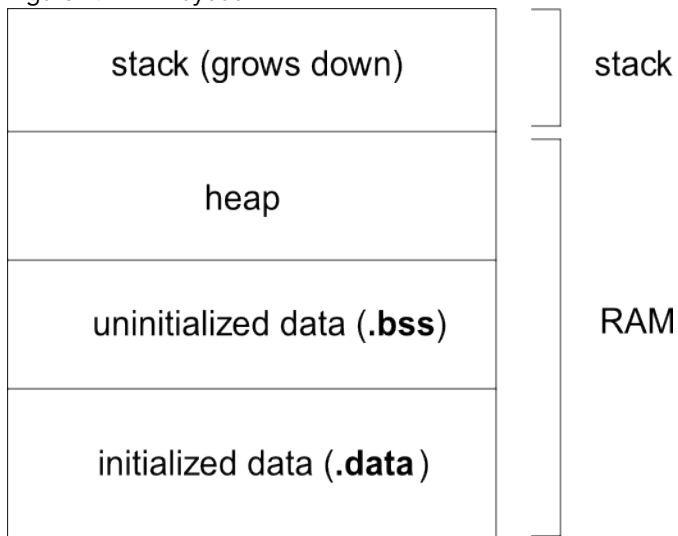
The *heap* occupies any unallocated parts of the ram memory area. The heap start is defined inside the linker command that allocates RAM, following the data sections.

```
__HEAP_START = .;
```

The heap end is the last address in the ram memory area:

```
__HEAP_END = ADDR(ram) + sizeof(ram);
```

Figure 9. RAM Layout



Related information

[Introduction to Linking on page 18](#)

5. BASIC TROUBLESHOOTING TIPS

1. Diagnosing and Fixing Compilation Problems

During software development you will most likely eventually encounter compiler or linker errors. In most cases the resulting error messages, along with the source file and line number that caused the error, will suffice to allow you to fix any problems.

Sometimes, however, error messages seem cryptic or may refer to files or symbols you are not aware of. Here are some general hints to help you in diagnosing and fixing problems.

Retain temporary assembler files

Situation: you are using embedded assembly code and you get an error from the assembler, referring to a file with a name like `"/tmp/dtmpBAAa19739."`

This message refers to the temporary assembly-code file that is created out of the compiled C code and the embedded assembly code. By the time the compiler is finished, the file may not still exist (depending on how your operating system handles its temporary directories). To ensure that the file persists and that it has the same name as the original C file (but with extension `.s`), use the **-Xkeep-assembly-file** switch.

See also [Embed source lines as comments in assembly files on page 29](#).

Embed source lines as comments in assembly files

Situation: you are looking at generated assembly files and you're getting somewhat lost.


A single assembly file can quickly become rather large, and you generally don't want to read it from top to bottom. You want to know what the compiler did to a particular line of code, or you need to see the code before and after some of your embedded assembly code to understand why there is a register clash.

In this case, use the **-Xassembly-listing -g** options, which produce an assembler listing file (**.lst**) with C source lines embedded as comments.

View command-line options and library paths used by the compiler

Situation: you are not sure the compiler is doing what it is supposed to do, or you want to know what options are invoked when a given **-t** option is used.

Start **dcc** with the **-#** option, which shows you the exact command lines used to start **clang** (the C/C++ compiler) and **dld** (the linker). **dcc** displays all the include and library search paths, and everything else that is produced by the **-t** argument.

 **Note:** The **#** notation is used in other contexts, so you may need to escape it with a backwards slash (****). For example, **#** is interpreted as a comment in both makefiles and the **bash** script language. In these cases you need to use **-\\#**.

Check configuration files to look at linker behavior

Situation: you want to know what the linker (**dld**) does when a given **-t** option is used. Actually, the **-t** option does not change linker behavior very much, and what behavior it does change is unlikely to adversely affect compilation output. However, you may still want to examine the effects of using **-t**.

Unfortunately the linker does not offer a `-#` option. If you still want to know what effect a `-t` option has, you will have to examine one or more of the configuration files.

1. First, make sure you know the target type (the `-t` argument) you are interested in. If you don't know your target (which may be because you generally start `dld` without a `-t` argument), look at `install_dir/version/conf/default.conf` to see the default target you are using.
2. Using your preferred editor, open `install_dir/version/conf/dtools.conf` and search for "case dld:". You should see several variables that get configured in response to the `-t` argument. One of the variables is `$DLIBS`. This variable is shared with `dcc`, and to see it, just execute the following command to show the `clang` and `dld` invocation. (You can literally use `foo.c`, because a source file does not need to exist to make `##` work.)

```
$ dcc -tYourTarget -## foo.c
```

3. Check the `dld` invocation and you will see a `-Y P,...` argument. This is the same argument that will be used by `dld` itself. Note the other variables (they all have "LFLAG" in their name).
4. Look for the configuration file `install_dir/version/conf/user.conf`. This configuration file contains user-specific settings, and if the file doesn't exist, then there are none. If the file exists, open it and check to see if it sets one of the flags used by `dld`.
5. Now, back in `dtools.conf`, search for a statement called `switch($DTARGET)`. Go through the list below the `switch` statement and find the first `case` label whose pattern matches your target. Then look through all labels until you find a `break` statement. Directly above that line you can see the target configuration file that will be included by the linker. Open it (you can find the file in `install_dir/version/conf/`).
6. Search that configuration file for "LFLAG." When you find a line containing that string, check the lines above it for the `case` label(s) that the line belongs to. If the label matches your target, you now know at least one piece of the linker flags. (There may be only one.) If you find multiple lines that match your target, then keep in mind that the flags usually stack up (but you can see in the kind of the assignment whether it's an override or an addition).
7. Back in `dtools.conf`, find a statement called `switch($DENVIROn)`. Check the lines below it; if your environment requires a configuration file, note its name, and open it (as always, you can find it in `install_dir/version/conf`). Repeat what you did in the target configuration file (Step [Step 6 on page](#)).
8. Finally, search `dtools.conf` for "LFLAGS" and check if the corresponding assignments apply to your target. You now know what flags `dld` will use when invoked with a certain `-t` option.

Report any actual problems with the Wind River compilation tools

Situation: you found a bug in the compiler or associated tools.

Here is what you should do:

Please include the following information, if applicable, when reporting a problem to Customer Support:

- A description of the problem and error symptoms (crash, invalid code, etc.)
- Details about the host environment
- The version of each tool you are using. Invoke the driver with the `-VV` option to see these version numbers
- The source file or linker command file. For source, prune the file to the minimum necessary to display the error. The preprocessed source file created using the `-P` option. By default, this creates a file with the same basename as the source file and the extension `.i`. This file expands all header files and reflects all options used when invoking the compiler
- The command used to invoke the tool. Invoke the driver (`dcc`) with `##`, which displays a detailed command line for each tool invocation; redirect the output to a file, and send the file.
- For the linker, include the command used to invoke the linker, perhaps in a makefile
- For the linker, include a link map. Use the linker option `-m6` to produce a detailed map and redirect it to a file
- If generated code appears to be in error, please include the assembly file generated by the `-S` or `-Xkeep-assembly-file` option (noting the incorrect instructions), the assembly listings file generated by `-Xassembly-listing`, or an object file.

You can also use the `-save-temps` option and send us the files that it produces (**.i**, **.ii**, **.s**, **.bc**), along with a list of the command-line options you've used.

Ideally, all of the above should be provided. For example, the command line

```
$ dcc -c -Xkeep-assembly-file -Xassembly-listing -g foo.c
```

will create all three files at once.

For more information about contacting Customer Support, see the Release Notes.

6. GLOSSARY

1. Diab Compiler Getting Started: Important Terms

assembler

A tool that takes assembly language input and converts it into object file formats. In contrast to the C compiler it does not translate the programming language, because there is a one on one mapping from assembler mnemonics to machine operation codes. (See [mnemonic on page](#) .) The assembler can apply minor optimizations to the original assembler code for maximum utilization of the chosen target. For example, some pseudo-instructions that have been generated by the compiler or low level optimizers can be replaced by the optimal real instructions, but the assembler does not reorder or remove instructions.

command-line option

An argument passed to the invocation of a given tool.

compiler

A tool to translate a given programming language (e.g. C or C++) into another language. The result is either an object file format, or an intermediary language suitable to be processed into an object file by other tools (e.g., the assembler).

The term "compiler" is sometimes loosely used as a synonym for the entire tool chain used to create an executable. See [tool chain on page](#) .

compiler switch

See [command-line option on page](#) .

data section

A section that usually contains only variables. See [section on page](#) .

double precision

See [single precision on page](#) .

dynamic library

See [shared library on page](#) .

executable file

A file in a well-defined binary format that can be loaded into the memory of a system and run. Depending on the type of executable, it is either loaded by an operating system running on the system, or it is loaded into the memory of the system using ROM- and/or RAM-programming tools. Sometimes simply referred to as an “executable.”

execution environment

The various external factors under which a compilation occurs and which may affect the compilation output. These factors include default libraries, startup code, default linker scripts, environmental variables, and so on.

external

A symbol not defined in the current compilation module. Hence it is “external” from the point of view of the current module.

floating-point mode

A method of representing numbers that cannot be represented as integers, either because they are too large or too small (e.g., 1.234×10^{25}) or because they are irrational (e.g., $2/3$). Some processors offer hardware support for [single precision on page](#) floating-point operations, others for both single- and [double precision on page](#), and others don't offer any hardware support at all. When compiling, you must indicate to the compiler how to handle floating-point operations (e.g., through software or hardware or not at all).

heap

A portion of RAM that contains neither executable code nor predefined variables. It can be used for dynamic memory allocations and is available for any process to use.

incremental linking

The process, executed by the linker, of combining object files and libraries into a [partially linked object file on page 32](#) that can be linked again to allow it to be eventually turned into an executable. Incremental linking allows the resulting object to still have unresolved references.

Instead of linking all object files at once (the usual case), one can create an executable by linking one object files at a time, i.e. incrementally.

input section

See [section on page](#).

library

See [shared library on page](#), [static library on page](#).

linking

A process executed by the linker. During linking, the linker combines input object files into a single output object file. During the process, the linker resolves unresolved references and combines the sections of the input object files into sections of the single output object.

linker

A tool to combine object files into executables or partially linked object files.

linker command file

A file used by the linker that contains ordering constraints on how to combine the sections of the input object files into the single output object file. It also contains information on how to perform [location on page](#) .

location

A process executed by the linker. During location the linker assigns physical addresses to the various symbols contained in a (partially linked) object file. Location must always be preceded by [linking on page](#) .

mnemonic

A human-readable form of an operation code.

object file

A binary file that can be linked into an executable, but is not executable on its own. Typically, several object files may be created from different source code files and then linked together to form an executable.

partially linked object file

A special kind of [object file on page](#) that is the result of performing incremental linking, created by the linker by linking a set of object files. The partially linked object file can still have unresolved references and is not fully located. (See [location on page](#) .)

object module

See [object file on page](#) .

optimization

The technique of modifying the compilation process so as to create executable code that maximally achieves certain desired qualities. Common code optimizations are for performance (speed), runtime memory consumption, and (static) code size. It is usually

difficult to optimize for two or more factors, e.g., for both runtime performance and static code size, and the developer must accept trade-offs—generally by using different compiler command-line options—between different types of optimization.

output section

See [section on page](#) .

reset vector

The address in memory from which the processor will read its very first instruction after power on or reset.

RAM (Random Access Memory)

Memory that can be written to and read from by a processor. The granularity of the RAM access depends on the chosen target. Common granularity levels are words (4 bytes), dwords (8 bytes) and single bytes.

RAM is usually volatile, i.e., the content is lost or corrupted in case of a hard reset or power down.

ROM (Read Only Memory)

Memory that normally can only be read from by a processor. The granularity of the ROM accesses depend on the type of ROM (i.e., NAND-Flash, NOR-Flash, or EEPROM).

Although ROM is normally read-only, there are times, such as during a firmware update, when it can be written to. ROM can be programmed, using special tools or software, to get it into a usable state (e.g., when it is initialized for the first time). ROM is usually non-volatile, i.e., the content is preserved even in case of hard reset or power down.

section

A subset of an object file. One section may contain code, another section may contain initialized variables, another may contain strings, and so on. Default sections are target-dependent; additional sections may be defined by the user.

Commonly, the linker bundles together various *input sections* in object files into *output sections* in an executable. For example, it bundles input sections containing initialized data created from different source files into a single output section of initialized data (which it locates in physical memory) in the executable.

See also [location on page](#) , [object file on page](#) .

shared library

Also known as a *dynamic library*. A package of precompiled routines that are loaded into an application at runtime. Because only the routines that are needed are linked in, the resulting executable is generally smaller than when static libraries are used. Also, many executables can make use of a set of shared libraries, and the libraries themselves may (often) be upgraded without the need to recompile executables. On the other hand, loading and exploiting dynamic libraries may be more complicated than using static libraries, and shared libraries must be distributed along with executables, unlike static libraries.

See also [static library on page](#) .

single precision

A term describing the storage format of a number. Double-precision storage is twice as large as single-precision; that is, double precision occupies two adjacent storage locations in memory. For example, under the IEEE 754 floating-point standard, single precision occupies 32 bits and double precision occupies 64.

small-data area (SDA)

A special kind of data section that only contains variables whose size is below a given threshold (e.g., 8 bytes). The overall size of a small-data area A has a target-dependent upper limit (e.g., 64k), hence the size limitation for the variables in it; otherwise a single massive array could consume the whole area.

The advantage of an SDA is that, if you reserve one of the processor registers for it, the register can contain the fixed address of the middle of the area, so that the variables contained therein can be addressed with a single-address, register-relative, memory-access instruction (as opposed to multiple instructions).

static library

An archive of object files. The linker can use libraries, and the objects they contain, during linkage to resolve unresolved references. Static libraries are compiled and linked directly into an application.

Because they are linked in their entirety, static libraries provide all of their functionality to the application and, unlike shared libraries, do not need to be distributed separately from an executable. However, the entire executable must be replaced when a static library is upgraded; also, using static libraries may waste space if the executable is including unneeded routines or data.

See also [shared library on page](#) .

symbol, symbol reference

An identifier in an object file (often in a symbol table) that refers to a data structure in a source file, such as a function or a variable. Every access to a variable or function becomes a so-called symbol reference. If the symbol and the reference are defined in the same compilation unit (e.g., a C file), the reference is said to be *resolved*. If the symbol is defined in another compilation unit, the reference is unresolved (because the compiler always only sees a single unit at a time) and the linker will have to resolve it. See also [object file on page](#) .

target

The architecture or system(-on-a-chip) the compiler will produce code for.

text section

A section that usually contains only executable code. See [section on page](#) .

tool chain

A set of tools for creating executables out of source code. The minimal Diab tool chain consist of the C and C++ compiler (**clang**), assembler (**as**<arch>), and linker (**dld** or **ld**<arch>). Additional parts of a tool chain can support the user in the process (e.g., **dmake**).

The term “compiler” is sometimes loosely used as a synonym for the whole tool chain, e.g., “the compiler creates an executable”; however, it is the combination of the actual compiler, assembler, and linker that create an executable.