ARM® Compiler

Version 5.04

Getting Started Guide

ARM

ARM DUI0529J

ARM® Compiler

Getting Started Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
A	28 May 2010	Non-Confidential	ARM Compiler v4.1 Release
B	30 September 2010	Non-Confidential	Update 1 for ARM Compiler v4.1
C	28 January 2011	Non-Confidential	Update 2 for ARM Compiler v4.1 Patch 3
D	30 April 2011	Non-Confidential	ARM Compiler v5.0 Release
E	29 July 2011	Non-Confidential	Update 1 for ARM Compiler v5.0
F	30 September 2011	Non-Confidential	ARM Compiler v5.01 Release
G	29 February 2012	Non-Confidential	Document update 1 for ARM Compiler v5.01 Release
H	27 July 2012	Non-Confidential	ARM Compiler v5.02 Release
I	31 January 2013	Non-Confidential	ARM Compiler v5.03 Release
J	27 November 2013	Non-Confidential	ARM Compiler v5.04 Release

Proprietary Notice

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Unrestricted Access is an ARM internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Web Address

www.arm.com

Contents
ARM® Compiler Getting Started Guide

List of Figures
ARM® Compiler Getting Started Guide

List of Tables
ARM® Compiler Getting Started Guide

Preface

This preface introduces the *ARM® Compiler Getting Started Guide*.

It contains the following:

About this book

ARM Compiler Getting Started Guide. This manual provides an overview of the ARM Compiler tools, standards supported, and compliance with the ARM *Application Binary Interface* (ABI). Available as a PDF.

Using this book

This book is organized into the following chapters:

Chapter 1 Overview of ARM® Compiler

Gives general information about ARM® Compiler.

Chapter 2 Getting Started with the Compilation Tools

Describes how to create an application using the tools provided by ARM Compiler.

Appendix A Getting Started Guide Document Revisions

Describes the technical changes that have been made to the Getting Started Guide.

Glossary

The ARM Glossary is a list of terms used in ARM documentation, together with definitions for those terms. The ARM Glossary does not contain terms that are industry standard unless the ARM meaning differs from the generally accepted meaning.

See the *ARM Glossary* for more information.

Typographic conventions

italic
: Introduces special terminology, denotes cross-references, and citations.

bold
: Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`
: Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`mono`space
: Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`
: Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`
: Denotes language keywords when used outside example code.

<and>
: Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS
: Used in body text for a few terms that have specific technical meanings, that are defined in the *ARM glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title.
- The number ARM DUI0529J.
- The page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Other information

- *ARM Information Center*.
- *ARM Technical Support Knowledge Articles*.
- *Support and Maintenance*.
- *ARM Glossary*.

Chapter 1
Overview of ARM® Compiler

Gives general information about ARM® Compiler.

It contains the following:

1.1 About ARM® Compiler

ARM Compiler enables you to build applications for the ARM family of processors from C, C++, or ARM assembly language source.

The toolchain comprises:

armcc
The ARM and Thumb® compiler. This compiles your C and C++ code.

It supports inline and embedded assemblers, and also includes the NEON™ vectorizing compiler, invoked using the command:

```
armcc --vectorize
```

armasm
The ARM and Thumb assembler. This assembles ARM and Thumb assembly language sources.

armlink
The linker. This combines the contents of one or more object files with selected parts of one or more object libraries to produce an executable program.

A 64-bit version of **armlink** is also provided that can access the greater amount of memory available on 64-bit machines. It supports all the features that are supported by the 32-bit version of **armlink** in this release.

If you are using ARM Compiler as a standalone product, then the 32-bit version is used by default.

For ARM Compiler in DS-5, the linker version depends on the host platform. 32-bit tools have the 32-bit linker and 64-bit tools have the 64-bit linker. You do not get both versions.

For the *Microprocessor Developer Kit* (MDK), only the 32-bit linker is provided.

armar
The librarian. This enables sets of ELF object files to be collected together and maintained in archives or libraries. You can pass such a library or archive to the linker in place of several ELF files. You can also use the archive for distribution to a third party for further application development.

fromelf
The image conversion utility. This can also generate textual information about the input image, such as disassembly and its code and data size.

ARM C++ libraries
The ARM C++ libraries provide:

- Helper functions when compiling C++.
- Additional C++ functions not supported by the Rogue Wave library.

ARM C libraries
The ARM C libraries provide:

- An implementation of the library features as defined in the C and C++ standards.
- Extensions specific to the compiler, such as `_fisatty()`, `__heapstats()`, and `__heapvalid()`.
- GNU extensions.
- Common nonstandard extensions to many C libraries.
- POSIX extended functionality.
- Functions standardized by POSIX.

ARM C microlib
The ARM C microlib provides a highly optimized set of functions. These functions are for use with deeply embedded applications that have to fit into extremely small amounts of memory.

Rogue Wave C++ library
The Rogue Wave C++ library provides an implementation of the standard C++ library.

C++ *Standard Template Library* (STL)
An ARM implementation of the C++ Standard Template Library.

Supporting software

You can debug the output from the toolchain with any debugger that is compatible with the ELF, DWARF 2, and DWARF 3 specifications.

Updates and patches to the toolchain are available from the ARM web site as they become available.

Related concepts

1.11 ARM® Compiler support on 64-bit host platforms on page 1-25.
1.5 Licensed features of ARM® Compiler on page 1-17.
1.7 Compliance with the ABI for the ARM Architecture (Base Standard) on page 1-19.
1.10 ARM architectures supported by the toolchain on page 1-24.
1.22 Rogue Wave documentation on page 1-36.

Related references

1.23 Further reading on page 1-37.

Related information

ARM website.
Overview of the Compiler.
Overview of the Assembler.
Overview of the Linker.
The ARM C and C++ Libraries.
The ARM C Micro-Library.
Overview of the ARM Librarian.
Overview of the fromelf Image Converter.

1.2 Host platform support for ARM® Compiler

ARM Compiler supports various Windows, Ubuntu, and Red Hat Enterprise Linux platforms.

Except where stated, the ARM Compiler supports both 32-bit and 64-bit versions of the following OS platforms:

- Windows 8 (64-bit).
- Windows 7 Enterprise Edition SP1.
- Windows 7 Professional Edition SP1.
- Windows XP Professional SP3 (32-bit only).
- Windows Server 2003.
- Windows Server 2008 R2.
- Ubuntu Desktop Edition 10.04 LTS (32-bit only) (deprecated in this release).
- Ubuntu Desktop Edition 12.04 LTS.
- Red Hat Enterprise Linux 5 Desktop and Workstation option, Standard.
- Red Hat Enterprise Linux 6 Desktop and Workstation option, Standard.

Note

You can also use ARM Compiler with Cygwin on the supported Windows platforms. However, Cygwin path translation enabled by `CYGPATH` is only supported on 32-bit Windows platforms, and is not supported on Windows 8.

About running 32-bit ARM Compiler binaries on Red Hat Enterprise Linux 6

Sometimes, Red Hat Enterprise Linux 6 might get installed by default without 32-bit compatibility libraries. To use ARM Compiler on Red Hat Enterprise Linux 6, you must install the 32-bit Compatibility Library support.

Specifically, ARM Compiler requires `glibc` and `libm`. To install them, make sure the **`yum`** command is set up correctly, and enter:

```
# yum groupinstall "Compatibility Libraries"
```

Related concepts

1.21 About specifying Cygwin paths in compilation tools on Windows on page 1-35.
1.1 About ARM® Compiler on page 1-12.

Related information

Cygwin versions supported.

1.3 Changing to the 64-bit linker

You must add the path to the 64-bit version of the executables directory to your PATH environment variable.

Procedure

To set up ARM Compiler to use the 64-bit version of **armlink**, on Windows for example, enter:

```
SET PATH=install_directory\bin64;%PATH%
```

1.4 About the toolchain documentation

ARM Compiler contains a suite of documents that describe how to use the tools and provides information on migration from, and compatibility with, earlier toolchian versions.

The toolchain documentation comprises:

Getting Started Guide (ARM DUI 0529) - this document
This document gives an overview of the toolchain and features.

Software Development Guide (ARM DUI 0471)
This document describes how to use the toolchain to develop software to run on ARM architecture-based processors.

See *Software Development Guide*.

armcc User Guide (ARM DUI 0472)
This document describes how to use the various features of the compiler, **armcc**.

See *armcc User Guide*.

ARM C and C++ Libraries and Floating-Point Support User Guide (ARM DUI 0475)
This document describes the features of the ARM C and C++ libraries, and how to use them. It also describes the floating-point support of the libraries.

See *ARM® C and C++ Libraries and Floating-Point Support User Guide*.

armasm User Guide (ARM DUI 0473)
This document describes how to use the various features of the assembler, **armasm**.

See *armasm User Guide*.

armlink User Guide (ARM DUI 0474)
This document describes how to use the various features of the linker, **armlink**.

See *armlink User Guide*.

armar User Guide (ARM DUI 0476)
This document describes how to use the various features of the librarian, **armar**. It also provides a detailed description of each **armar** command-line option.

See *armar User Guide*.

fromelf User Guide (ARM DUI 0477)
This document describes how to use the various features of the ELF image converter, **fromelf**. It also provides a detailed description of each **fromelf** command-line option.

See *fromelf User Guide*.

Errors and Warning Reference Guide (ARM DUI 0496)
This document describes the errors and warnings that might be generated by each of the build tools in ARM Compiler.

See *Errors and Warnings Reference Guide*.

Migration and Compatibility Guide (ARM DUI 0530)
This document describes the differences you must be aware of in ARM Compiler, when migrating your software from earlier toolchain versions, such as ARM RVCT v4.0.

See *Migration and Compatibility Guide*.

1.5 Licensed features of ARM® Compiler

ARM Compiler requires a license.

If you purchased the toolchain with another ARM product, see the *Getting Started* document of that product for details of the licenses that are included.

Licensing of the ARM development tools is controlled by the FlexNet license management system.

To request a license, go to *ARM Web Licensing* and follow the online instructions.

Related information

ARM DS-5 License Management Guide.

1.6 Standards compliance in ARM® Compiler

ARM Compiler conforms to the ISO C, ISO C++, ELF, DWARF 2, and DWARF 3 standards.

The level of compliance for each standard is:

ar
: **armar** produces, and **armlink** consumes, UNIX-style object code archives. **armar** can list and extract most **ar**-format object code archives, and **armlink** can use an **ar**-format archive created by another archive utility providing it contains a symbol table member.

DWARF 3
: DWARF 3 debug tables (DWARF Debugging Standard Version 3) are supported by the toolchain.

DWARF 2
: DWARF 2 debug tables are supported by the toolchain, and by ELF DWARF 2 compatible debuggers from ARM.

ISO C
: The compiler accepts ISO C 1990 and 1999 source as input.

ISO C++
: The compiler accepts ISO C++ 2003 source as input.

ELF
: The toolchain produces relocatable and executable files in ELF format. The **fromelf** utility can translate ELF files into other formats.

Note

The DWARF 2 and DWARF 3 standards are ambiguous in some areas such as debug frame data. This means that there is no guarantee that third-party debuggers can consume the DWARF produced by ARM code generation tools or that an ARM debugger can consume the DWARF produced by third-party tools.

Related concepts

1.7 Compliance with the ABI for the ARM Architecture (Base Standard) on page 1-19.

Related information

Source language modes of the compiler.
The DWARF Debugging Standard.
International Organization for Standardization.

1.7 Compliance with the ABI for the ARM Architecture (Base Standard)

The ABI for the ARM Architecture (Base Standard) is a collection of standards. Some of these standards are open. Some are specific to the ARM architecture.

The *Application Binary Interface (ABI) for the ARM Architecture (Base Standard)* (BSABI) regulates the inter-operation of binary code and development tools in ARM architecture-based execution environments, ranging from bare metal to major operating systems such as ARM Linux.

By conforming to this standard, objects produced by the toolchain can work together with object libraries from different producers.

The BSABI consists of a family of specifications including:

AADWARF
: *DWARF for the ARM Architecture*. This ABI uses the DWARF 3 standard to govern the exchange of debugging data between object producers and debuggers.

AAELF
: *ELF for the ARM Architecture*. Builds on the generic ELF standard to govern the exchange of linkable and executable files between producers and consumers.

AAPCS
: *Procedure Call Standard for the ARM Architecture*. Governs the exchange of control and data between functions at runtime. There is a variant of the AAPCS for each of the major execution environment types supported by the toolchain.

BPABI
: *Base Platform ABI for the ARM Architecture*. Governs the format and content of executable and shared object files generated by static linkers. Supports platform-specific executable files using post linking. Provides a base standard for deriving a platform ABI.

CLIBABI
: *C Library ABI for the ARM Architecture*. Defines an ABI to the C library.

CPPABI
: *C++ ABI for the ARM Architecture*. Builds on the generic C++ ABI (originally developed for IA-64) to govern interworking between independent C++ compilers.

DBGOVL
: *Support for Debugging Overlaid Programs*. Defines an extension to the *ABI for the ARM Architecture* to support debugging overlaid programs.

EHABI
: *Exception Handling ABI for the ARM Architecture*. Defines both the language-independent and C++-specific aspects of how exceptions are thrown and handled.

RTABI
: *Run-time ABI for the ARM Architecture*. Governs what independently produced objects can assume of their execution environments by way of floating-point and compiler helper function support.

If you are upgrading from a previous toolchain release, ensure that you are using the most recent versions of the ARM specifications.

Related information

Application Binary Interface for the ARM Architecture Introduction and downloads.

Addenda to, and Errata in, the ABI for the ARM Architecture.

Differences between v1 and v2 of the ABI for the ARM Architecture.

ABI for the ARM Architecture Advisory Note: SP must be 8-byte aligned on entry to AAPCS-conforming functions.

1.8 GCC compatibility provided by ARM® Compiler

ARM Compiler provides **gcc** compatibility to aid development using a range of source bases that might have been originally developed to target specific toolchains.

ARM Compiler:

- Can build the vast majority of C and C++ code that is written to be built with **gcc**.
- Is not 100% source compatible in all cases.
- Does not aim to be bug-compatible.

ARM Compiler might emulate specific defects present in **gcc** where the defective behavior is relied on in significant cases.

The level of **gcc** comparability, and **gcc** bug compatibility, might vary over time as updates from EDG are incorporated.

1.9 Toolchain environment variables

Except for ARMLMD_LICENSE_FILE, ARM Compiler does not require any other environment variables to be set. However, there are situations where you might want to set environment variables.

For example, if you want to specify additional command-line options for **armcc**, but you do not want to modify your build scripts, then you can specify the options using ARMCC5_CCOPT.

The environment variables used by the toolchain are:

Table 1-1 Environment variables used by the toolchain

Environment variable	Setting
ARMLMD_LICENSE_FILE	This environment variable must be set, and specifies the location of your ARM license file. See the *License Management Guide* for information on this environment variable. **Note** On Windows, the length of ARMLMD_LICENSE_FILE must not exceed 260 characters.
ARMROOT	Your installation directory root, *install_directory*.
ARMCC5_ASMOPT	An optional environment variable to define additional assembler options that are to be used outside your regular makefile. For example: `--licretry` The options listed appear before any options specified for the **armasm** command in the makefile. Therefore, any options specified in the makefile might override the options listed in this environment variable.
ARMCC5_CCOPT	An optional environment variable to define additional compiler options that are to be used outside your regular makefile. For example: `--licretry` The options listed appear before any options specified for the **armcc** command in the makefile. Therefore, any options specified in the makefile might override the options listed in this environment variable.
ARMCC5_FROMELFOPT	An optional environment variable to define additional **fromelf** image converter options that are to be used outside your regular makefile. For example: `--licretry` The options listed appear before any options specified for the **fromelf** command in the makefile. Therefore, any options specified in the makefile might override the options listed in this environment variable.
ARMCC5_LINKOPT	An optional environment variable to define additional linker options that are to be used outside your regular makefile. For example: `--licretry` The options listed appear before any options specified for the **armlink** command in the makefile. Therefore, any options specified in the makefile might override the options listed in this environment variable.

Table 1-1 Environment variables used by the toolchain (continued)

Environment variable	Setting
ARMCC5INC	The default system include path. That is, the path used by the compiler to search for header filenames enclosed in angle-brackets. The compiler option `-J` overrides this environment variable. The default location of the compiler include files is: *install_directory*`\include`
ARMCC5LIB	The default location of the ARM standard C and C++ library files: *install_directory*`\lib` The compiler option `--libpath` overrides this environment variable. **Note** If you include a path separator at the end of the path, the linker searches that directory and the subdirectories. So for *install_directory*`\lib` the linker searches: *install_directory*`\lib` *install_directory*`\lib\armlib` *install_directory*`\lib\cpplib`
ARMINC	Used only if you do not specify the compiler option `-J` and ARMCC5INC is either not set or is empty. See the description of ARMCC5INC for more information.
ARMLIB	Used only if you do not specify the compiler option `--libpath` and ARMCC5LIB is either not set or is empty. See the description of ARMCC5LIB for more information.
CPATH	Defines additional paths that are used by **armcc** when the GCC emulation mode `--translate_gcc` or `--translate_g++` is specified.
CPLUS_INCLUDE_PATH	Defines additional include paths that are used by **armcc** when the GCC emulation mode `--translate_gcc` or `--translate_g++` is specified.
C_INCLUDE_PATH	Defines additional paths that are used by **armcc** when the GCC emulation mode `--translate_gcc` or `--translate_g++` is specified.
CYGPATH	The location of the **cygpath.exe** file on your system in Cygwin path format. For example: `C:/cygwin/bin/cygpath.exe` You must set this if you want to specify paths in Cygwin format for the compilation tools. **Note** Cygwin path translation enabled by CYGPATH is only supported on 32-bit Windows platforms, and is not supported on Windows 8.

Table 1-1 Environment variables used by the toolchain (continued)

Environment variable	Setting
`TMP`	Used on Windows platforms to specify the directory to be used for temporary files. If `TMP` is not defined, or if it is set to the name of a directory that does not exist, temporary files are created in the current working directory.
`TMPDIR`	Used on Red Hat Linux platforms to specify the directory to be used for temporary files. If `TMPDIR` is not set, a default temporary directory, usually `/tmp` or `/var/tmp`, is used.

Related concepts

1.19 TMP and TMPDIR environment variables for temporary file directories on page 1-33.

1.21 About specifying Cygwin paths in compilation tools on Windows on page 1-35.

Related references

1.16 Methods of specifying command-line options on page 1-30.

Related information

ARM DS-5 License Management Guide.

1.10 ARM architectures supported by the toolchain

ARM Compiler includes support for all ARM architectures from ARMv4™ onwards that are currently supported by ARM, including ARM NEON technology.

All architectures before ARMv4 are obsolete and are no longer supported.

You can specify a target processor or architecture to take advantage of extra features specific to the selected processor or architecture. To do this, use the following command-line options:

- `--cpu=`*name*.
- `--fpu=`*name*.

You can specify the startup instruction set, ARM or Thumb, with the `--arm` or `--thumb` command-line options.

You can force an ARM-only instruction set with the `--arm_only` option.

The compilation tools provide support for mixing ARM and Thumb code. This is known as interworking and enables branching between ARM code and Thumb code.

Related information

Selecting the target CPU at compile time.
NEON technology.
--arm compiler option.
--arm_only compiler option.
--cpu=name compiler option.
--fpu=name compiler option.
--thumb compiler option.
--arm assembler option.
--arm_only assembler option.
--cpu=name assembler option.
--fpu=name assembler option.
--thumb assembler option.
--arm_only linker option.
--cpu=name linker option.
--fpu=name linker option.
Introducing NEON Development Article.
NEON Support in Compilation Tools Development Article.

1.11 ARM® Compiler support on 64-bit host platforms

ARM Compiler provides 32-bit and 64-bit versions of **armlink**.

Although ARM Compiler is supported on certain 64-bit platforms, the tools are 32-bit applications. This limits the virtual address space and file size available to the tools. If these limits are exceeded, **armlink** reports an error message to indicate that there is not enough memory. This might cause confusion because sufficient physical memory is available but the application cannot access it.

A 64-bit version of **armlink** is provided in a separate executables directory in this release.

The 64-bit version of **armlink** can:

- Access the greater amount of memory available to processes on 64-bit operating systems.
- Support all the features that are supported by the 32-bit version of **armlink** in this release.

If you have installed ARM Compiler on a 64-bit machine, then you can use the 64-bit version instead.

——— **Note** ———

By default, your installation is set up to use the standard 32-bit version of **armlink**, even if you are using a 64-bit operating system.

———————————

Related references

1.9 Toolchain environment variables on page 1-21.

1.12 Considerations when using the 64-bit linker

There are some considerations you must be aware of when you are using the 64-bit version of **armlink**.

Be aware of the following:

- In the 64-bit version of the executables directory, **armlink** is the 64-bit executable and all other tools are the 32-bit executables. It might seem redundant to have duplicated 32-bit versions of the executables in the 64-bit executables directory, such as **armcc.exe**. However, this is required by the method that different executables use to call each other. This method dictates that all executables must be in the same directory. The tools call each other in certain circumstances, for example **armcc** calls **armlink** to produce an executable when `-c` is not specified in the command line.
- Cygwin path translation enabled by `CYGPATH` is only supported on 32-bit Windows platforms, and is not supported on Windows 8.

1.13 Special characters on the command line

You can use special characters to select multiple symbolic names in some compilation tools command arguments.

If you are using a special character on Unix platforms, you must enclose the options in quotes to prevent the shell expanding the selection.

- The wildcard character * matches any name.
- The wildcard character ? matches any single character.

For example, enter `'*,~*.*'` instead of `*,~*.*`.

Note

The **armar** command-line options must be preceded by a `-`. This is different from some earlier versions of **armar**, and from some third-party archivers.

Examples

The following examples show the use of these characters:

- **armar** `--create mylib.a *.o`, creates the archive `mylib.a` containing all object files in the current directory.
- **armar** `-t mylib.a s*.o`, lists all object files beginning with s in the `mylib.a`.
- **armlink** `hello.o mylib.a(?.o) -o tst.axf`, links `hello.o` with all object files in `mylib.a` that have a single letter filename.
- **fromelf** `--elf --strip=debug mylib.a(s*.o) --output=my_archive.a`, strips debug information from all object files beginning with s in `mylib.a`, and puts the modified files in `my_archive.a`.

Related information

Assembler command-line syntax.
Compiler command-line syntax.
Compiler command-line options listed by group.
Linker command-line syntax.
armar command-line syntax.
fromelf command-line syntax.

1.14 Rules for specifying command-line options

There are certain rules you must follow when using command-line options. These rules depend on the type of option.

The following rules apply:

Single-letter options

All single-letter options, including single-letter options with arguments, are preceded by a single dash `-`. You can use a space between the option and the argument, or the argument can immediately follow the option. For example:

`-J` *`directory`*

`-J`*`directory`*

Keyword options

All keyword options, including keyword options with arguments, are preceded by a double dash `--`. An `=` or space character is required between the option and the argument. For example:

```
--depend=file.d
```

```
--depend file.d
```

Compilation tools options that contain non-leading `-` or `_` can use either of these characters. For example, `--force_new_nothrow` is the same as `--force-new-nothrow`.

To compile files with names starting with a dash, use the POSIX option `--` to specify that all subsequent arguments are treated as filenames, not as command switches. For example, to compile a file named `-ifile_1`, use:

```
armcc -c -- -ifile_1
```

In some Unix shells, you might have to include quotes when using arguments to some command-line options, for example:

```
--keep='s.o(vect)'
```

Related information

Assembler command-line syntax.
Compiler command-line syntax.
Compiler command-line options listed by group.
Linker command-line syntax.
armar command-line syntax.
fromelf command-line syntax.

1.15 Order of options on the command line

In general, command-line options can appear in any order. However, the effects of some options depend on how they are combined with other related options.

Where options override other options on the same command line, the options that appear closer to the end of the command line take precedence. Where an option does not follow this rule, this is noted in the description for that option.

Use the `--show_cmdline` option to see how the command line is processed. The commands are shown normalized.

Related information

--show_cmdline compiler option.
--show_cmdline assembler option.
--show_cmdline linker option.
--show_cmdline fromelf option.
--show_cmdline armar option.

1.16 Methods of specifying command-line options

You can specify command-line options directly. Some operating systems restrict the length of the command line, so you can also specify command-line options in environment variables or in a text file.

You can:

- Specify the commands directly on the command line. However, the number of options you can specify is limited by the command length supported by your operating system.
- Specify the commands in a text file, called a *via file*. A separate via file is required for each tool. You can use a via file to overcome the command length limitation of your operating system.
- Use tool-specific environment variables. These are:
  - `ARMCC5_ASMOPT` for the assembler.
  - `ARMCC5_CCOPT` for the compiler.
  - `ARMCC5_FROMELFOPT` for the fromelf image converter.
  - `ARMCC5_LINKOPT` for the linker.

  The syntax is identical to the command-line syntax. The compilation tool reads the value of the environment variable and inserts it at the front of the command string. This means that you can override options specified in the environment variable by arguments on the command line.

Related information

--via compiler option.
--via assembler option.
--via linker option.
--via fromelf option.
--via armar option.

1.17 Precedence of command-line options when using them in a text file

The compilation tools read command-line options from a specified text file and combine them with any additional options you have specified for the tool. Some options might take precedence over other options.

The precedence given to a command-line option depends on:

- The command-line option.
- The position of the `--via` option on the command line.

To see a command line equivalent to the result of combining the options, specify the `--show_cmdline` option. For example, if `armcc.txt` contains the options `--debug --cpu=ARM926EJ-S`:

- **armcc** `-c --show_cmdline --cpu=ARM7TDMI --via=armcc.txt hello.c [armcc --show_cmdline --debug -c --cpu=ARM926EJ-S hello.c]`

  In this case, `--cpu=ARM7TDMI` is not used because `--cpu=ARM926EJ-S` is the last instance of `--cpu` on the command-line.
- **armcc** `--via=armcc.via -c --show_cmdline --cpu=ARM7TDMI hello.c [armcc --show_cmdline --debug -c hello.c]`

  In this case, `--cpu=ARM926EJ-S` is not used because `--cpu=ARM7TDMI` is the last instance of `--cpu` on the command line. In addition, `--cpu=ARM7TDMI` is not shown in the output, because this is the default option for `--cpu`.

1.18 Portability of source files between hosts

There are guidelines you can follow to assist portability of source files between hosts.

The guidelines are:

- Ensure that filenames do not contain spaces. If you have to use path names or filenames containing spaces, enclose the path and filename in double (") or single (') quotes.
- Make embedded path names relative rather than absolute.
- Use forward slashes (/) in embedded path names, not backslashes (\).

1.19 TMP and TMPDIR environment variables for temporary file directories

The compilation tools use a temporary directory when processing files.

The environment variable name used to refer to this directory depends on the platform:

- `TMP` on Windows platforms. If `TMP` is not defined, or if it is set to the name of a directory that does not exist, temporary files are created in the current working directory.
- `TMPDIR` on Red Hat Linux platforms. If `TMPDIR` is not set, a default temporary directory, usually `/tmp` or `/var/tmp`, is used.

`TMP` and `TMPDIR` are typically set up by a system administrator. However, it is permissible for you to change them.

Related references

1.9 Toolchain environment variables on page 1-21.

1.20 Autocompletion of command-line options

You can specify a shortened version of a command-line option with the autocompletion feature.

To use the autocompletion feature, insert a full stop (`.`) after the characters to be autocompleted.

The following rules apply to the autocompletion feature:

- You must separate arguments from the full stop by an equals (`=`) character or a space character.
- You cannot use autocompletion for the arguments to an option.
- You must include sufficient characters to make the autocompleted option unique.

For example, use `--diag_su.=223` to specify `--diag_suppress=223` on the command line.

Specifying `--diag.=223` is not valid, because `--diag.` does not identify a single unique command-line option.

Related information

Compiler command-line options listed by group.

1.21 About specifying Cygwin paths in compilation tools on Windows

You must use an environment variable to specify Cygwin paths for compilation tools on Windows.

By default on Windows, the compilation tools require path names to be in the Windows DOS format, for example, `C:\myfiles`. If you want to use Cygwin path names, then set the `CYGPATH` environment variable to the location of the **`cygpath.exe`** file on your system. For example:

```
set CYGPATH=C:/cygwin/bin/cygpath.exe
```

You can now specify file locations in the compilation tools command-line options using the Cygwin path format. The paths are translated by **`cygpath.exe`**. For example, to compile the file `/cygdrive/h/main.c`, enter the command:

```
armcc -c --debug /cygdrive/h/main.c
```

You can still specify paths that start with:

- A drive letter, for example `C:\` or `C:/`.
- UNC, for example, `\\computer`.

The compilation tools do not translate these paths because the paths are already in a form that Windows understands.

Limitations of CYGPATH

Be aware of the following limitations with `CYGPATH`:

- Cygwin path translation enabled by `CYGPATH` is only supported on 32-bit Windows platforms, and is not supported on Windows 8.
- When using a Cygwin style path with spaces or other special terminal characters, the path must be double quoted:
  - The use of single quotes or escaping characters is not supported.
  - The use of literal doublequote characters in path names is not supported.

Related references

1.9 Toolchain environment variables on page 1-21.

Related information

Cygwin versions supported.

Cygwin home page.

1.22 Rogue Wave documentation

The documentation for the Rogue Wave Standard C++ Library used by ARM Compiler is available from the ARM website. It is also installed with some ARM products.

The manuals for the Rogue Wave Standard C++ Library used by the compilation tools are:

- *Standard C++ Library Class Reference*.
- *Standard C++ Library User's Guide - OEM Edition*.

These manuals might be installed with the documentation of your ARM product. If they are not installed, you can view them at *Rogue Wave Standard C++ Library Documentation*

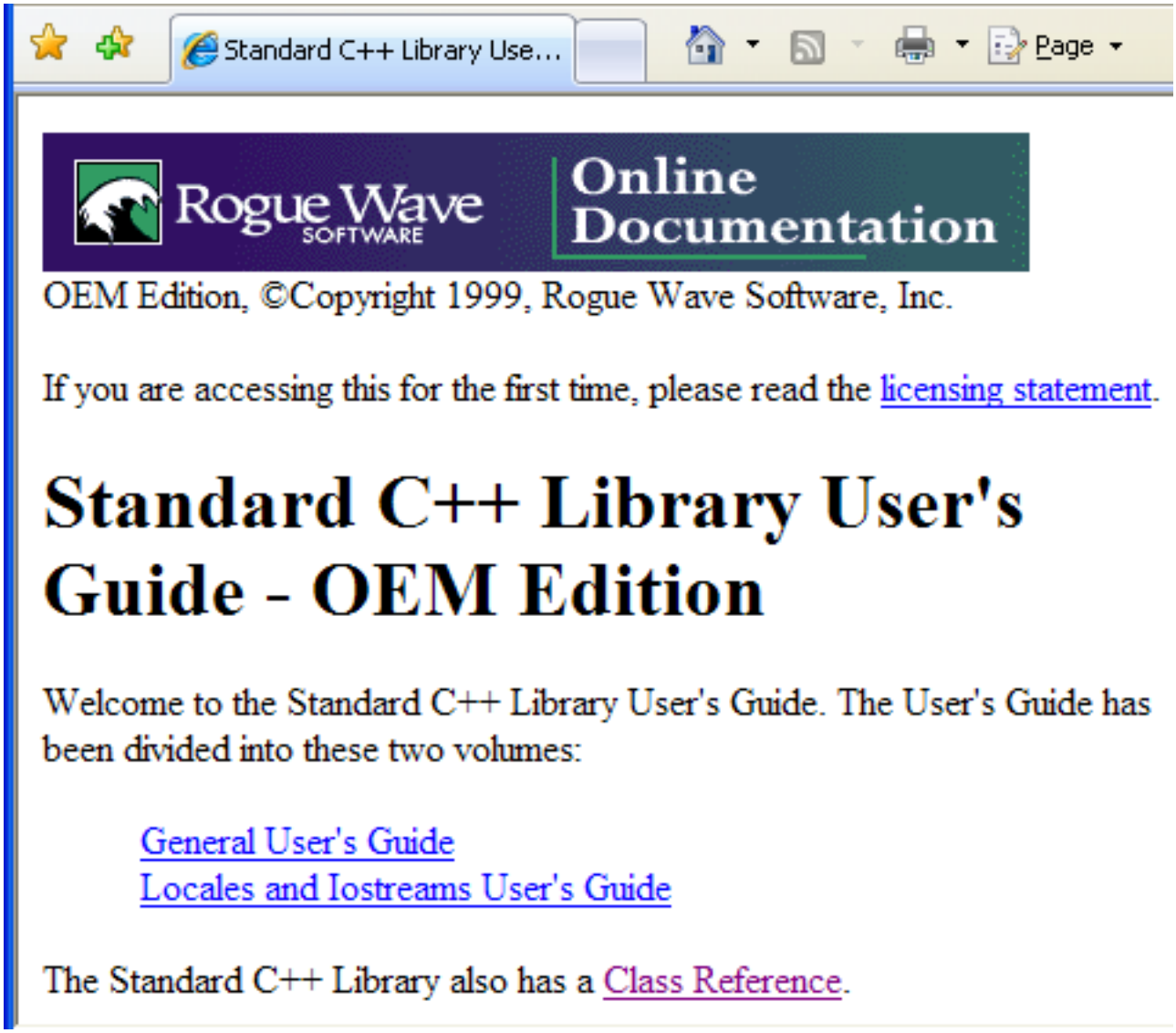

Figure 1-1 Rogue Wave HTML documentation

1.23 Further reading

Additional information on developing code for the ARM family of processors is available from both ARM and third parties.

ARM publications

ARM periodically provides updates and corrections to its documentation. See *ARM Infocenter* for current errata sheets and addenda, and the ARM Frequently Asked Questions (FAQs).

For full information about the base standard, software interfaces, and standards supported by ARM, see *Application Binary Interface (ABI) for the ARM Architecture*.

In addition, see the following documentation for specific information relating to ARM products:

- *ARM Architecture Reference Manuals*.
- *Cortex-A series processors*.
- *Cortex-R series processors*.
- *Cortex-M series processors*.
- *ARM11 processors*.
- *ARM9 processors*.
- *ARM7 processors*.

Other publications

This ARM Compiler tools documentation is not intended to be an introduction to the C or C++ programming languages. It does not try to teach programming in C or C++, and it is not a reference manual for the C or C++ standards. Other publications provide general information about programming.

The following publications describe the C++ language:

- *ISO/IEC 14882:2003, C++ Standard*.
- Stroustrup, B., *The C++ Programming Language* (3rd edition, 1997). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-88954-4.

The following publications provide general C++ programming information:

- Stroustrup, B., *The Design and Evolution of C++* (1994). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-54330-3.

  This book explains how C++ evolved from its first design to the language in use today.
- Vandevoorde, D and Josuttis, N.M. *C++ Templates: The Complete Guide* (2003). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-73484-2.
- Meyers, S., *Effective C++* (1992). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-56364-9.

  This provides short, specific guidelines for effective C++ development.
- Meyers, S., *More Effective C++* (2nd edition, 1997). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-92488-9.

The following publications provide general C programming information:

- ISO/IEC 9899:1999, *C Standard.*

  The standard is available from national standards bodies (for example, AFNOR in France, ANSI in the USA).
- Kernighan, B.W. and Ritchie, D.M., *The C Programming Language* (2nd edition, 1988). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-110362-8.

This book is co-authored by the original designer and implementer of the C language, and is updated to cover the essentials of ANSI C.

- Harbison, S.P. and Steele, G.L., *A C Reference Manual* (5th edition, 2002). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-089592-X.

  This is a very thorough reference guide to C, including useful information on ANSI C.
- Plauger, P., *The Standard C Library* (1991). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-131509-9.

  This is a comprehensive treatment of ANSI and ISO standards for the C Library.
- Koenig, A., *C Traps and Pitfalls,* Addison-Wesley (1989), Reading, Mass. ISBN 0-201-17928-8.

  This explains how to avoid the most common traps in C programming. It provides informative reading at all levels of competence in C.

See *The DWARF Debugging Standard web site* for the latest information about the *Debug With Arbitrary Record Format* (DWARF) debug table standards and ELF specifications.

The following publications provide information about the *European Telecommunications Standards Institute* (ETSI) basic operations:

- ETSI Recommendation G.191: *Software tools for speech and audio coding standardization*.
- *ITU-T Software Tool Library 2005 User's manual*, included as part of ETSI Recommendation G.191.
- ETSI Recommendation G723.1: *Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s*.
- ETSI Recommendation G.729: *Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear prediction (CS-ACELP)*.

These publications are all available from the telecommunications bureau of the *International Telecommunication Union (ITU) web site*.

Publications providing information about Texas Instruments compiler intrinsics are available from *Texas Instruments web site*.

The Wireless MMX Technology Developer Guide is available from Intel.

Chapter 2
Getting Started with the Compilation Tools

Describes how to create an application using the tools provided by ARM Compiler.

It contains the following:

2.1 About the ARM compilation tools

The compilation tools allow you to build executable images, partially linked object files, and shared object files, and to convert images to different formats.

A typical application development might involve the following:

- Compiling C/C++ source code for the main application (**armcc**).
- Assembling ARM assembly source code for near-hardware components, such as interrupt service routines (**armasm**).
- Linking all objects together to generate an image (**armlink**).
- Converting an image to flash format in plain binary, Intel Hex, and Motorola-S formats (**fromelf**).

The following figure shows how the compilation tools are used for the development of a typical application.

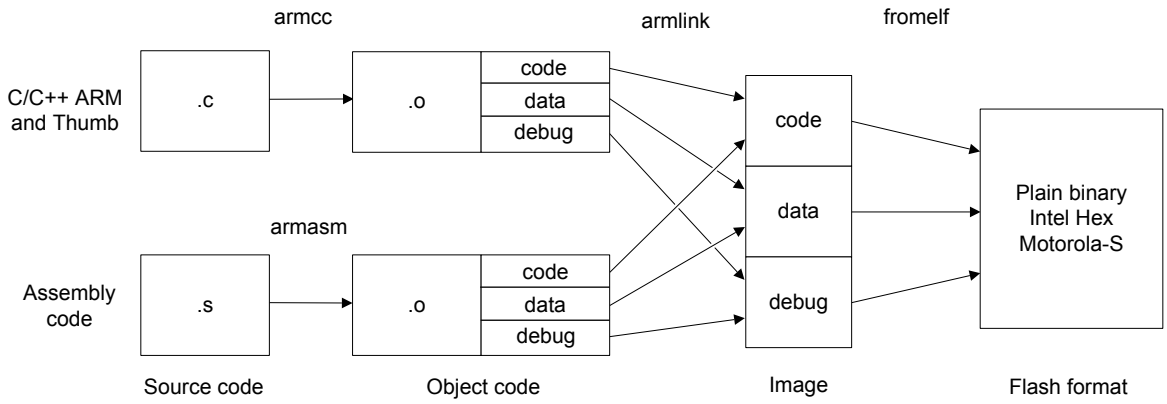

Figure 2-1 A typical tool usage flow diagram

Related concepts

2.2 The ARM compiler command on page 2-41.
2.3 The ARM linker command on page 2-43.
2.4 The ARM assembler command on page 2-44.
2.5 The fromelf image converter command on page 2-45.

2.2 The ARM compiler command

The compiler, **armcc**, can compile C and C++ source code into ARM and Thumb code.

Typically, you invoke the compiler as follows:

armcc [*options*] *file_1* … *file_n*

You can specify one or more input files. The compiler produces one object file for each source input file.

Building an example image from C++ source

To compile a C++ file called `shapes.cpp`:

1. Compile the C++ file `shapes.cpp` with the following command:

```
armcc -c --cpp --debug -O1 shapes.cpp -o shapes.o
```

2. The following options are commonly used:

   -c
   Tells the compiler to compile only, and not link.

   --cpp
   Tells the compiler that the source is C++.

   --debug
   Tells the compiler to add debug tables for source-level debugging.

   -O1
   Tells the compiler to generate code with restricted optimizations applied to give a satisfactory debug view with good code density and performance.

   -o *filename*
   Tells the compiler to create an object file with the specified *filename*.

   Note

   Be aware that `--arm` is the default compiler option.

3. Link the file:

```
armlink shapes.o --info totals -o shapes.axf
```

4. Use an ELF, DWARF 2, and DWARF 3 compatible debugger to load and run the image.

Command-line options for compiling ARM code

The following compiler options generate ARM code:

--arm
Tells the compiler to generate ARM code in preference to Thumb code. However, `#pragma thumb` overrides this option.

This is the default compiler option.

--arm_only
Forces the compiler to generate only ARM code. The compiler behaves as if Thumb is absent from the target architecture. Any `#pragma thumb` declarations are ignored.

Command-line options for compiling Thumb code

The following compiler option generates Thumb code:

`--thumb`

Tells the compiler to generate Thumb code in preference to ARM code. However, `#pragma arm` overrides this option.

Related concepts

2.3 The ARM linker command on page 2-43.

Related information

--arm compiler option.
--arm_only compiler option.
#pragma thumb compiler option.
--thumb compiler option.
#pragma arm compiler option.

2.3 The ARM linker command

The linker combines the contents of one or more object files with selected parts of one or more object libraries to produce an image or object file.

Typically, you invoke the linker as follows:

armlink [*options*] *file_1* … *file_n*

Linking an example object file

To link the object file `shapes.o`, enter:

```
armlink shapes.o --info totals -o shapes.axf
```

`-o`
: Specifies the output file as `shapes.axf`.

`--info totals`
: Tells the linker to display totals of the Code and Data sizes for input objects and libraries.

2.4 The ARM assembler command

The assembler, **armasm**, can assemble ARM assembly code into ARM and Thumb code.

Typically, you invoke the assembler as follows:

armasm [*options*] *inputfile*

Building an example from assembly source

To build the assembler program `word.s`:

1. Assemble the source file:

```
armasm --debug word.s
```

2. Link the object file:

```
armlink word.o -o word.axf
```

3. Use an ELF, DWARF 2, and DWARF 3 compatible debugger to load and run the image. Step through the program and examine the registers to see how they change.

Related information

Assembler command-line syntax.

2.5 The fromelf image converter command

fromelf allows you to convert ELF files into different formats and display information about them.

The features of the **fromelf** image converter include:

- Converting an executable image in ELF executable format to other file formats.
- Controlling debug information in output files.
- Disassembling either an ELF image or an ELF object file.
- Protecting *intellectual property* (IP) in images and objects that are delivered to third parties.
- Printing information about an ELF image or an ELF object file.

Examples

The following examples show how to use **fromelf**:

```
fromelf --text -c -s --output=outfile.lst infile.axf
```

Creates a plain text output file that contains the disassembled code and the symbol table of an elf image.

```
fromelf --bin --16x2 --output=outfile.bin infile.axf
```

Creates two files in binary format (`outfile0.bin` and `outfile1.bin`) for a target system with a memory configuration of a 16-bit memory width in two banks.

The output files in the last example are suitable for writing directly to a 16-bit Flash device.

Related information

fromelf command-line syntax.

Appendix A
Getting Started Guide Document Revisions

Describes the technical changes that have been made to the Getting Started Guide.

It contains the following:

- *A.1 Revisions for Getting Started Guide on page Appx-A-47.*

A.1 Revisions for Getting Started Guide

The following technical changes have been made to the Getting Started Guide.

Table A-1 Differences between Issue I and Issue J

Change	Topics affected
Updated the host platform support	*1.2 Host platform support for ARM® Compiler on page 1-14*
Moved the topic on avoiding the `BLX` (immediate) instruction issue on ARM 1176 processors to the *armlink User Guide*.	*Avoiding the BLX (immediate) instruction issue on an ARM1176JZ-S or ARM1176JZF-S processor*

Table A-2 Differences between issue H and issue I

Change	Topics affected
NEON vectorizing compiler no longer requries a separate license, so removed the statement.	• *1.1 About ARM® Compiler on page 1-12* • *1.5 Licensed features of ARM® Compiler on page 1-17*
Added note about CYGPATH not being supported by the 64-bit linker.	• *1.12 Considerations when using the 64-bit linker on page 1-26* • *1.9 Toolchain environment variables on page 1-21* • *1.21 About specifying Cygwin paths in compilation tools on Windows on page 1-35*
Updated the host platforms supported, and added information about 32-bit compatibility libraries on Red Hat Enterprise Linux 6.	*1.2 Host platform support for ARM® Compiler on page 1-14*
Removed the reference to `--ltcg` from the note, and enhanced the description.	*1.3 Changing to the 64-bit linker on page 1-15*
Removed the reference to *Building Linux Applications with ARM Compiler toolchain and GNU Libraries*.	*1.4 About the toolchain documentation on page 1-16*
Added a topic for GCC compatibility provided by ARM Compiler.	*1.8 GCC compatibility provided by ARM® Compiler on page 1-20*

Table A-3 Differences between Issue F and Issue H

Change	Topics affected
Updated the Windows 7 platform support.	*1.2 Host platform support for ARM® Compiler on page 1-14*

Table A-4 Differences between Issue E and Issue F

Change	Topics affected
Updated the list of environment variables to the new version numbering scheme, for example `ARMCC5INC`.	• *1.9 Toolchain environment variables on page 1-21* • *1.16 Methods of specifying command-line options on page 1-30*.

Table A-5 Differences between Issue D and Issue E

Change	Topics affected
Added a section on avoiding the `BLX` (immediate) instruction issue on ARM 1176 processors.	Topic moved to the *armlink User Guide* in revision J.

Table A-6 Differences between Issue C and Issue D

Change	Topics affected
Added Windows Server 2008 R2 and Ubuntu 10.04 LTS 32/64 to the list of platforms supported, and included a note about Cygwin support. Removed Windows Vista and Solaris from the list of platforms supported.	*1.2 Host platform support for ARM® Compiler on page 1-14*
Added a description of how to set up the DS-5 environment to use the 64-bit linker.	*1.3 Changing to the 64-bit linker on page 1-15*
Updated the list of environment variables. Removed `ARMCC`*`nn`*`BIN`, because it is not used by any of the tools.	*1.9 Toolchain environment variables on page 1-21*

Table A-7 Differences between Issue A and Issue B

Change	Topics affected
Added details about the 64-bit version of `armlink` to list of tools.	*1.1 About ARM® Compiler on page 1-12*
Added a topic on how to change to using the 64-bit linker.	*1.3 Changing to the 64-bit linker on page 1-15*
Added the environment variables required for using the 64-bit linker.	*1.9 Toolchain environment variables on page 1-21*
Added a note about the 64-bit version of `armlink`.	*1.11 ARM® Compiler support on 64-bit host platforms on page 1-25*