ARM® Compiler toolchain v4.1 for µVision

Introducing the ARM Compiler toolchain

ARM®

ARM DUI 0592A (ID061811)

ARM Compiler toolchain v4.1 for µVision
Introducing the ARM Compiler toolchain

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Confidentiality	Change
June 2011	A	Non-Confidential	Release for ARM Compiler toolchain v4.1 for µVision

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.

Product Status

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

Web Address

`http://www.arm.com`

Contents
ARM Compiler toolchain v4.1 for µVision Introducing the ARM Compiler toolchain

Chapter 1
Conventions and feedback

The following describes the typographical conventions and how to give feedback:

Typographical conventions

The following typographical conventions are used:

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

`mono`space Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

`monospace italic`

Denotes arguments to commands and functions where the argument is to be replaced by a specific value.

`monospace bold`

Denotes language keywords when used outside example code.

italic Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

bold Highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate, and for ARM® processor signal names.

Feedback on this product

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

- your name and company

- the serial number of the product
- details of the release you are using
- details of the platform you are using, such as the hardware platform, operating system type and version
- a small standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem
- the version string of the tools, including the version number and build numbers.

Feedback on content

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

- the title
- the number, ARM DUI 0592A
- if viewing online, the topic names to which your comments apply
- if viewing a PDF version of a document, the page numbers to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

ARM periodically provides updates and corrections to its documentation on the ARM Information Center, together with knowledge articles and *Frequently Asked Questions* (FAQs).

Other information

- ARM Product Manuals, `http://www.keil.com/support/man_arm.htm`
- Keil Support Knowledgebase, `http://www.keil.com/support/knowledgebase.asp`
- Keil Product Support, `http://www.keil.com/support/`
- ARM Glossary, `http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html`.

Chapter 2
Overview of the ARM Compiler toolchain

The following topics provide general information about the ARM Compiler toolchain:

Tasks

Concepts

Reference

- *Further reading* on page 2-26.

2.1 About the ARM Compiler toolchain

The ARM Compiler toolchain 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.

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.

armar The librarian. This enables sets of ELF format 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.

C++ libraries

The ARM C++ libraries provide:

- helper functions when compiling C++
- additional C++ functions not supported by the Rogue Wave library.

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()`
- common nonstandard extensions to many C libraries.
- POSIX extended functionality
- functions standardized by POSIX.

C micro-libraries

The ARM C *micro-library* (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 library provides an implementation of the standard C++ library.

2.1.1 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.

2.1.2 See also

Concepts

- *Host platform support for ARM Compiler toolchain* on page 2-5

- *Licensed features of the toolchain* on page 2-8
- *Standards compliance in the toolchain* on page 2-9
- *Compliance with the ABI for the ARM Architecture (Base Standard)* on page 2-10.
- *ARM architectures supported by the toolchain* on page 2-14.
- *Toolchain support on 64-bit host platforms* on page 2-15.
- *Further reading* on page 2-26.

Using the Compiler:

- Chapter 2 *Overview of the compiler*.

Using the Assembler:

- Chapter 2 *Overview of the Assembler*.

Using the Linker:

- Chapter 2 *Overview of the linker*.

Using ARM® C and C++ Libraries and Floating Point Support:

- Chapter 2 *The ARM C and C++ libraries*
- Chapter 3 *The ARM C micro-library*.

Creating Static Software Libraries with armar:

- Chapter 2 *Overview of the ARM Librarian*.

Using the fromelf Image Converter:

- Chapter 2 *Overview of the fromelf image converter*.

Other information

- ARM website, `http://www.arm.com`.

2.2 Host platform support for ARM Compiler toolchain

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

- Windows 7 Enterprise Edition
- Windows 7 Professional Edition
- Windows XP Professional SP3 (32-bit only)
- Windows Server 2003
- Windows Server 2008 R2

2.2.1 See also

Tasks

- *Specifying Cygwin paths in compilation tools on Windows* on page 2-25.

Concepts

- *About the ARM Compiler toolchain* on page 2-3.

2.3 About the toolchain documentation

The toolchain documentation comprises:

Introducing the ARM® Compiler toolchain (ARM DUI 0592) - this document

This document gives an overview of the toolchain and features.

Using the Compiler (ARM DUI 0375)

This document describes how to use the various features of the compiler, `armcc`.

Compiler Reference (ARM DUI 0376)

This document provides reference information for the various features of the compiler, `armcc`. It also provides a detailed description of each compiler command-line option.

Using ARM® C and C++ Libraries and Floating-Point Support (ARM DUI 0378)

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.

ARM® C and C++ Libraries and Floating-Point Support Reference (ARM DUI 0589)

This document provides reference information for the various features of the ARM C and C++ libraries.

Using the Assembler (ARM DUI 0379)

This document describes how to use the various features of the assembler, `armasm`.

Assembler Reference (ARM DUI 0588)

This document provides reference information for the various features of the assembler, `armasm`. It also provides a detailed description of each assembler command-line option.

Using the Linker (ARM DUI 0377)

This document describes how to use the various features of the linker, `armlink`.

Linker Reference (ARM DUI 0458)

This document provides reference information for the various features of the linker, `armlink`. It also provides a detailed description of each linker command-line option.

Creating Static Software Libraries with armar (ARM DUI 0590)

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.

Using the fromelf Image Converter (ARM DUI 0459)

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.

Errors and Warning Reference (ARM DUI 0591)

This document describes the errors and warnings that might be generated by each of the build tools in ARM Compiler v4.1.

Migration and Compatibility (ARM DUI 0593)

This document describes the differences you must be aware of in the ARM Compiler toolchain v4.1 for µVision, when migrating your software from earlier toolchain versions, such as ARM RVCT v4.0 for µVision.

2.4 Licensed features of the toolchain

The toolchain requires a toolchain 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 for µVision is controlled by the Keil licensing system, see *Keil Product Licensing*, http://www.keil.com/license/ for details.

2.5 Standards compliance in the toolchain

The toolchain conforms to the following standards. In each case, the level of compliance is noted:

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 standard is 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.

2.5.1 See also

Concepts

- *Compliance with the ABI for the ARM Architecture (Base Standard)* on page 2-10.

Using the Compiler:

- *Source language modes of the compiler* on page 2-3.

Other information

- The DWARF Debugging Standard, `http://dwarfstd.org/`
- International Organization for Standardization, `http://www.iso.org/iso/home.htm`.

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

The *Application Binary Interface (ABI) for the ARM Architecture (Base Standard)* (BSABI) is a collection of standards. Some of these standards are open. Some are specific to the ARM architecture. They regulate the inter-operation of binary code and development tools in ARM architecture-based execution environments.

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*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0040-/index.html. This ABI uses DWARF 3 standard to govern the exchange of debugging data between object producers and debuggers.

AAELF *ELF for the ARM Architecture*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0044-/index.html. 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*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042-/index.html. 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*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0037-/index.html. 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 that is used to derive a platform ABI.

CLIBABI *C Library ABI for the ARM Architecture*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0039-/index.html. Defines an ABI to the C library.

CPPABI *C++ ABI for the ARM Architecture*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0041-/index.html. Builds on the generic C++ ABI (originally developed for IA-64) to govern interworking between independent C++ compilers.

DBGOVL *Support for Debugging Overlay Programs*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0049-/index.html. Defines an extension to the *ABI for the ARM Architecture* to support debugging overlaid programs.

EHABI *Exception Handling ABI for the ARM Architecture*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0038-/index.html. Defines both the language-independent and C++-specific aspects of how exceptions are thrown and handled.

RTABI *Run-time ABI for the ARM Architecture*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0043-/index.html. 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 to ARM Compiler v4.1 for µVision from a previous toolchain release, ensure that you are using the most recent versions of the ARM specifications.

2.6.1 See also

Other information

- *Application Binary Interface for the ARM Architecture Introduction and downloads*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0036-/index.html
- *Addenda to, and Errata in, the ABI for the ARM Architecture*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0045-/index.html
- *Differences between v1 and v2 of the ABI for the ARM Architecture*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0047-/index.html
- *ABI for the ARM Architecture Advisory Note: SP must be 8-byte aligned on entry to AAPCS-conforming functions*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0046-/index.html.

2.7 Toolchain environment variables

The ARM Compiler toolchain does not require 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 `ARMCC*nn*_CCOPT`.

The environment variables used by the toolchain are:

Table 2-1 Environment variables used by the toolchain

Environment variable [a]	Setting
`ARMROOT`	Your installation directory root (*`install_directory`*). This documentation assumes that *`install_directory`* is `C:\Keil\ARM`.
`ARMCC*nn*_ASMOPT`	An optional environment variable to define additional assembler options that are to be used outside your regular makefile. 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.
`ARMCC*nn*_CCOPT`	An optional environment variable to define additional compiler options that are to be used outside your regular makefile. 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.
`ARMCC*nn*_FROMELFOPT`	An optional environment variable to define additional `fromelf` image converter options that are to be used outside your regular makefile. 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.
`ARMCC*nn*_LINKOPT`	An optional environment variable to define additional linker options that are to be used outside your regular makefile. 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.
`ARMCC*nn*INC`	The default system include path. That is, the path used 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`*`\RV31\INC`

`ARMCC*nn*LIB`	The default location of the ARM standard C and C++ library files:
	`install_directory``\RV31\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`*`\RV31\LIB` the linker searches:
	`LIB`
	`LIB\armlib`
	`LIB\cpplib`

`ARMINC`	Used only if you do not specify the compiler option `-J` and `ARMCC*nn*INC` is either not set or is empty. See the description of `ARMCC*nn*INC` for more information.

Table 2-1 Environment variables used by the toolchain (continued)

Environment variable [a]	Setting
ARMLIB	Used only if you do not specify the compiler option --libpath and ARMCC*nn*LIB is either not set or is empty. See the description of ARMCC*nn*LIB for more information.
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.
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.

a. Replace *nn* with the version of the toolchain you are using. For example, ARMCC41INC if you are using ARM Compiler toolchain v4.1.

2.7.1 See also

Concepts

- *TMP environment variable for temporary file directories* on page 2-23
- *Specifying command-line options with an environment variable* on page 2-24
- *Specifying Cygwin paths in compilation tools on Windows* on page 2-25.

Using the Assembler:

- Chapter 2 *Overview of the Assembler*.

Using the Compiler:

- Chapter 2 *Overview of the compiler*.

Using the Linker:

- Chapter 2 *Overview of the linker*.

Using ARM® C and C++ Libraries and Floating Point Support:

- Chapter 2 *The ARM C and C++ libraries*.

Using the fromelf Image Converter:

- Chapter 2 *Overview of the fromelf image converter*.

Creating Static Software Libraries with armar:

- Chapter 2 *Overview of the ARM Librarian*.

Reference

Assembler Reference:

- Chapter 2 *Assembler command line options*.

Compiler Reference:

- Chapter 3 *Compiler Command-line Options*.

Linker Reference:

- Chapter 2 *Linker command-line options*.

Using the fromelf Image Converter:

- Chapter 4 *fromelf command reference*.

2.8 ARM architectures supported by the toolchain

The toolchain includes support for all ARM architectures from ARMv4™ onwards that are currently supported by ARM. 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. Also, 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.

2.8.1 See also

Tasks

Using the Compiler:

- *Selecting the target CPU at compile time* on page 5-8.

Reference

Compiler Reference:

- *--arm* on page 3-11
- *--arm_only* on page 3-11
- *--cpu=name* on page 3-20
- *--fpu=name* on page 3-44
- *--thumb* on page 3-90.

Assembler Reference:

- *--arm* on page 2-6
- *--arm_only* on page 2-6
- *--cpu=name* on page 2-8
- *--fpu=name* on page 2-13
- *--thumb* on page 2-23.

Linker Reference:

- *--arm_only* on page 2-9
- *--cpu=name* on page 2-27
- *--fpu=name* on page 2-57.

2.9 Toolchain support on 64-bit host platforms

Although the toolchain 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.

2.9.1 See also

Concepts

- *About the ARM Compiler toolchain* on page 2-3.

2.10 Using special characters on the compilation tools command-line

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

- the wildcard character * can be used to match any name
- the wildcard character ? can be used to match any single character.

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

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.

2.10.1 See also

Reference

Using the Assembler:

- *Assembler command line syntax* on page 7-2
- *Assembler commands listed in groups* on page 7-3.

Using the Compiler:

- *Compiler command-line syntax* on page 3-3
- *Compiler command-line options listed by group* on page 3-4.

Using the Linker:

- *Linker command-line syntax* on page 2-3
- *Linker command-line options listed in groups* on page 2-4.

Creating Static Software Libraries with armar:

- *armar command-line syntax* on page 2-4
- *armar command-line options listed in groups* on page 2-5.

Using the fromelf Image Converter:

- *fromelf command-line syntax* on page 2-6
- *fromelf command-line options listed in groups* on page 2-7.

2.11 Compilation tools command-line option rules

You can control many aspects of the compilation tools operation with command-line options.

The following rules apply, depending on the type of option:

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)'
```

2.11.1 See also

Reference

Using the Assembler:

- *Assembler command line syntax* on page 7-2
- *Assembler commands listed in groups* on page 7-3.

Using the Compiler:

- *Compiler command-line syntax* on page 3-3
- *Compiler command-line options listed by group* on page 3-4.

Using the Linker:

- *Linker command-line syntax* on page 2-3
- *Linker command-line options listed in groups* on page 2-4.

Creating Static Software Libraries with armar:

- *armar command-line syntax* on page 2-4
- *armar command-line options listed in groups* on page 2-5.

Using the fromelf Image Converter:

- *fromelf command-line syntax* on page 2-6
- *fromelf command-line options listed in groups* on page 2-7.

2.12 About ordering the compilation tools command-line options

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.

2.12.1 See also

Reference

Compiler Reference:

- *--show_cmdline* on page 3-85.

Assembler Reference:

- *--show_cmdline* on page 2-22.

Linker Reference:

- *--show_cmdline* on page 2-112.

Using the fromelf Image Converter:

- *--show_cmdline* on page 4-47.

Creating Static Software Libraries with armar:

- *--show_cmdline* on page 6-28.

2.13 Autocompletion of compilation tools command-line option

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.

2.13.1 See also

Reference

Using the Assembler:

- *Assembler commands listed in groups* on page 7-3.

Using the Compiler:

- *Compiler command-line options listed by group* on page 3-4.

Using the Linker:

- *Linker command-line options listed in groups* on page 2-4.

Creating Static Software Libraries with armar:

- *armar command-line options listed in groups* on page 2-5.

Using the fromelf Image Converter:

- *fromelf command-line options listed in groups* on page 2-7.

2.14 Using a text file to specify command-line options

Some operating systems restrict the length of the command line. You can either:

- specify options that extend beyond this limit by including them in a text file
- place all of your command-line options in a text file.

To use a text file to specify command-line options:

1. Create a text file containing the required command-line options. The options must be specified on a single line. For example:

   ```
   --debug --cpu=ARM926EJ-S
   ```

2. Use the `--via` command-line option to specify the file location containing the required options. For example:

   ```
   armcc --via myoptions.txt
   ```

   You can use any filename extension, or no filename extension.

2.14.1 Priority of command-line options when using them in a text file

The compiler reads the command-line options from the specified file and combines them with any additional options you have specified on the command-line. The priority given to a command-line option depends on:

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

To see the priority of 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`.

2.14.2 See also

Reference

Assembler Reference:

- *--show_cmdline* on page 2-22
- *--via=file* on page 2-24.

Compiler Reference:

- *--show_cmdline* on page 3-85
- *--via=filename* on page 3-95
- Appendix B *Via File Syntax*.

Linker Reference:

- *--show_cmdline* on page 2-112
- *--via=file* on page 2-145.

Creating Static Software Libraries with armar:

- *--show_cmdline* on page 6-28
- *--via=file* on page 6-35.

Using the fromelf Image Converter:

- *--show_cmdline* on page 4-47
- *--via=file* on page 4-53.

2.15 Portability of source files between hosts

To assist portability of source files between hosts, use the following guidelines:

- 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 (\).

2.15.1 See also

Concepts

- Chapter 2 *Overview of the ARM Compiler toolchain*.

2.16 TMP environment variable for temporary file directories

The compilation tools use the environment variable TMP 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.

TMP is typically set up by a system administrator. However, it is permissible for you to change it.

2.16.1 See also

Concepts

- Chapter 2 *Overview of the ARM Compiler toolchain*.

Reference

- *Toolchain environment variables* on page 2-12.

2.17 Specifying command-line options with an environment variable

You can specify command-line options by setting the value of a tool-specific environment variable:

- `ARMCC`*nn*`_ASMOPT` for the assembler
- `ARMCC`*nn*`_CCOPT` for the compiler
- `ARMCC`*nn*`_FROMELFOPT` for the fromelf image converter
- `ARMCC`*nn*`_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.

2.17.1 See also

Reference

- *Toolchain environment variables* on page 2-12.

2.18 Specifying Cygwin paths in 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:/`
- a UNC, for example, `\\computer`.

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

2.18.1 See also

Reference

- *Toolchain environment variables* on page 2-12.

Assembler Reference:

- Chapter 2 *Assembler command line options*.

Compiler Reference:

- Chapter 3 *Compiler Command-line Options*.

Linker Reference:

- Chapter 2 *Linker command-line options*.

Creating Static Software Libraries with armar:

- Chapter 6 *armar command reference*.

Using the fromelf Image Converter:

- Chapter 4 *fromelf command reference*.

2.19 Further reading

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

2.19.1 ARM publications

ARM periodically provides updates and corrections to its documentation. See ARM Infocenter, http://infocenter.arm.com/help/index.jsp 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*, http://infocenter.arm.com/help/topic/com.arm.doc.subset.swdev.abi/index.html.

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

- ARM Architecture Reference Manuals, http://infocenter.arm.com/help/topic/com.arm.doc.subset.arch.reference/index.html
- Cortex-R series processors, http://infocenter.arm.com/help/topic/com.arm.doc.set.cortexr/index.html
- Cortex-M series processors, http://infocenter.arm.com/help/topic/com.arm.doc.set.cortexm/index.html
- ARM9 processors, http://infocenter.arm.com/help/topic/com.arm.doc.set.arm9/index.html
- ARM7 processors, http://infocenter.arm.com/help/topic/com.arm.doc.set.arm7/index.html
- Vector floating-point coprocessors, http://infocenter.arm.com/help/topic/com.arm.doc.set.vfp/index.html.

2.19.2 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, `http://www.dwarfstd.org` 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, `http://www.itu.int`.

Publications providing information about Texas Instruments compiler intrinsics are available from Texas Instruments web site, `http://www.ti.com`.

Chapter 3
Creating an application

The following topics describe how to create an application using the toolchain:

Tasks

3.1 Using the compilation tools

A typical application development might involve the following:

- C/C++ source code for the main application (`armcc`)
- assembly source code for near-hardware components (`armasm`), such as interrupt service routines
- 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.

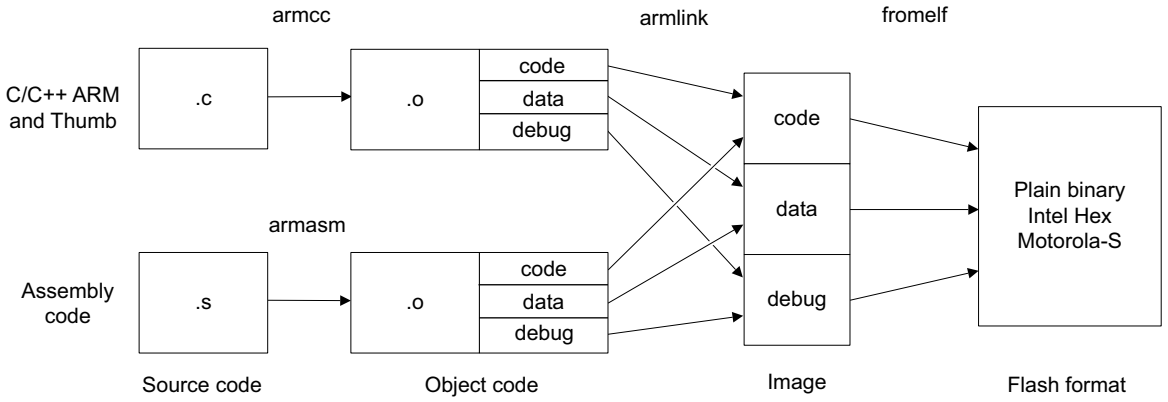

Figure 3-1 A typical tool usage flow diagram

3.1.1 See also

Tasks

- *Using the compiler* on page 3-3
- *Using the linker* on page 3-5
- *Using the assembler* on page 3-6
- *Using the fromelf image converter* on page 3-7.

3.2 Using the compiler

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

Typically, you invoke the compiler as follows:

armcc [*options*] *ifile_1 ... ifile_n*

You can specify one or more input files.

3.2.1 Compiling example

To compile the C++ example source file shapes.cpp:

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

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

   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.

2. Link the file:

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

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

See the readme.txt file that accompanies the example for more information.

3.2.2 Compiling for 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.

3.2.3 Compiling for Thumb code

To build a Thumb version use:

```
armcc --thumb ...
```

where:

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

3.2.4 See also

Tasks

- *Using the linker* on page 3-5.

Using the Compiler:

- Chapter 3 *Getting started with the Compiler*.

Reference

Compiler Reference:

- *Command-line options* on page 3-6
- *--arm* on page 3-11
- *--arm_only* on page 3-11
- *-c* on page 3-17
- *--debug, --no_debug* on page 3-24
- *-g* on page 3-47
- *-o filename* on page 3-69
- *-Onum* on page 3-71
- *--thumb* on page 3-90
- *#pragma arm* on page 5-47
- *#pragma thumb* on page 5-60.

3.3 Using the linker

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
```

3.3.1 Linking example

To link the object file `shapes.o`, use the command:

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

where:

`-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.

3.3.2 See also

Tasks

Using the Linker:

- Chapter 3 *Linking models supported by armlink*.

Reference

Linker Reference:

- Chapter 2 *Linker command-line options*.

3.4 Using the assembler

The basic syntax to use the assembler (`armasm`) is:

```
armasm [options] inputfile
```

For example, to assemble the code in a file called `myfile.s`, and to include debugging information in the resulting object file, type:

```
armasm --debug myfile.s
```

This generates an object file called `myfile.o`.

3.4.1 Building an example from assembler source

To build an assembler program, for example `word.s`:

1. Assemble the source file using the command:
   ```
   armasm --debug word.s
   ```
2. Link the file using the command:
   ```
   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.

3.4.2 See also

Tasks

Using the Assembler:

- *Assembler command line syntax* on page 7-2.

Reference

Assembler Reference:

- Chapter 2 *Assembler command line options*.

3.5 Using the fromelf image converter

The features of the `fromelf` image converter utility 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.

3.5.1 Examples of using fromelf

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 16-bit Flash device.

3.5.2 See also

Tasks

Using the fromelf Image Converter:

- Chapter 3 *Using fromelf*.

Reference

Using the fromelf Image Converter:

- Chapter 4 *fromelf command reference*.