ARM® Compiler

Version 6.00

Getting Started Guide

ARM

ARM DUI0741A

ARM® Compiler

Getting Started Guide

Release information

Document History

Issue	Date	Confidentiality	Change
A	14 March 2014	Non-Confidential	ARM Compiler v6.0 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

http://www.arm.com

Contents
ARM® Compiler Getting Started Guide

Chapter 2 Creating an Application

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*.

This section contains the following subsections:

About this book

The ARM Compiler Getting Started Guide provides general information for ARM Compiler 6 users.

Using this book

This book is organized into the following chapters:

Chapter 1 Overview of the ARM Compiler 6 Toolchain

Gives general information about ARM® Compiler 6.

Chapter 2 Creating an Application

Describes how to create an application using ARM Compiler.

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.

`monospace`
: 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 DUI0741A.
- The page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note

ARM tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Chapter 1
Overview of the ARM Compiler 6 Toolchain

Gives general information about ARM® Compiler 6.

It contains the following sections:

1.1 About ARMv8 terminology

ARMv8 introduces a number of terms to describe the state of the integer register bank and supported instruction sets.

ARMv8 introduces the following terms to describe the state of the integer register bank:

AArch32
The state in which the integer registers are 32-bit. There are 16 unbanked registers available in this state.

AArch64
The state in which the integer registers are 64-bit. There are 31 unbanked registers (0 to 30) available in this state, with register number 31 being a special case.

The unbanked registers can be accessed as either 32-bit registers (bits 0 to 31 only) or 64-bit registers.

Register number 31 represents:

Zero Register
In most cases register number 31 reads as zero when used as a source register, and discards the result when used as a destination register.

Stack Pointer
When used as a load/store base register, and in a small selection of arithmetic instructions, register number 31 provides access to the current stack pointer.

The PC is never accessible as a named register.

ARMv8 introduces the following terms to describe the instruction sets supported:

A32
An alias for the ARM instruction set that uses 32-bit encoded instructions. It is available in AArch32 state.

T32
An alias for the Thumb instruction set that uses 16-bit and 32-bit encoded instructions. It is available in AArch32 state. T32 support incorporates Thumb-2 technology.

A64
The instruction set available in AArch64 state. It uses 32-bit encoded instructions.

——— **Note** ———

The terms ARM and Thumb are used for features that are equivalent in A32 and T32 respectively. If a feature has no equivalent, then A32 or T32 is used as required.

———————————

——— **Note** ———

Detailed information about the ARMv8 architecture is available under license. Contact your ARM Account Representative for details.

———————————

Related information

ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

1.2 About ARM® Compiler

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

ARM Compiler supports the following architectures:

- ARMv8™ bare metal targets.

The toolchain comprises:

armclang
The compiler. This compiles C, C++, and GNU assembly language sources.

The compiler is based on LLVM and Clang technology.

LLVM is a set of open-source components that allow the implementation of optimizing compiler frameworks.

Clang is a compiler front end for LLVM, providing support for the C and C++ programming languages.

armasm
The legacy assembler. This assembles A32, A64, and T32 ARM syntax assembly code.

Only use `armasm` for legacy ARM syntax assembly code. Use the `armclang` assembler and GNU syntax for all new assembly files.

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 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.
- Common nonstandard extensions to many C libraries.
- POSIX extended functionality.
- Functions standardized by POSIX.

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

Supporting software

You can debug the output from the toolchain with any debugger that is compatible with the ELF and DWARF 4 specifications, such as ARM DS-5.

Check the ARM web site for updates and patches to the toolchain.

1.3 Host platform support for ARM® Compiler

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

The ARM Compiler supports 64-bit versions of the following OS platforms:

- Windows 7 Enterprise Edition SP1.
- Windows 7 Professional Edition SP1.
- Windows 8 Professional Edition.
- Windows Server 2012 Standard.
- Ubuntu Desktop Edition 12.04 LTS. (32-bit compatibility libraries are required.)
- Red Hat Enterprise Linux 5 Desktop with Workstation option. (32-bit compatibility libraries are required.)
- Red Hat Enterprise Linux 6 Workstation. (32-bit compatibility libraries are required.)

Related concepts

1.11 ARM® Compiler and virtual address space on page 1-23.

1.4 About the toolchain documentation

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

The toolchain documentation comprises:

Getting Started Guide (ARM DUI 0741) - this document
This document gives an overview of the toolchain.

Migration and Compatibility Guide (ARM DUI 0742)
This document describes the differences you must be aware of in ARM Compiler 6, when migrating your software from older versions of ARM compiler.

Software Development Guide (ARM DUI 0773)
This document contains information to help you develop code for various ARM architecture-based processors.

armclang Reference Guide (ARM DUI 0774)
This document provides user information for the ARM compiler, `armclang`. `armclang` is an optimizing C and C++ compiler that compiles Standard C and Standard C++ source code into machine code for ARM architecture-based processors.

armasm User Guide (ARM DUI 0801)
This document describes how to use the various features of the legacy ARM assembler, `armasm`.

armasm Reference Guide (ARM DUI 0802)
This document provides reference information for the various features of the legacy ARM assembler, `armasm`. It also provides a detailed description of each assembler command-line option.

armlink User Guide (ARM DUI 0803)
This document describes how to use the various features of the linker, `armlink`.

armlink Reference Guide (ARM DUI 0804)
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.

armar User Guide (ARM DUI 0806)
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.

fromelf User Guide (ARM DUI 0805)
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 Warnings Reference Guide (ARM DUI 0807)
This document describes the errors and warnings that might be generated by each of the build tools in ARM Compiler toolchain.

ARM C and C++ Libraries and Floating-Point Support User Guide (ARM DUI 0808)
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 Guide (ARM DUI 0809)
This document provides reference information for the various features of the ARM C and C++ libraries.

Related concepts

1.16 Rogue Wave documentation on page 1-29.
1.17 Clang and LLVM documentation on page 1-30.

Related references

1.18 Further reading on page 1-31.

1.5 ARM® Compiler toolchain licensing

ARM Compiler requires a license.

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

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 4
: DWARF 4 debug tables (DWARF Debugging Standard Version 4) are generated by the compiler.

DWARF 3
: DWARF 3 debug tables are supported by the rest of the tools in the toolchain.

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.

Related concepts

1.7 Compliance with the ABI for the ARM Architecture (Base Standard) on page 1-17.

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:

AADWARF64
: *DWARF for the ARM 64-bit Architecture (AArch64)*. This ABI uses the DWARF 3 standard to govern the exchange of debugging data between object producers and debuggers. It also gives additional rules on how to use DWARF 3, and how it is extended in ways specific to the ARM 64-bit architecture.

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.

AAELF64
: *ELF for the ARM 64-bit Architecture (AArch64)*. This specification provides the processor-specific definitions required by ELF for AArch64-based systems. It builds on the generic ELF standard to govern the exchange of linkable and executable files between producers and consumers.

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.

AAPCS64
: *Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*. 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.

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.

CPPABI64
: *C++ ABI for the ARM Architecture*. This specification 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 concepts

1.6 Standards compliance in ARM® Compiler on page 1-16.

1.8 GCC compatibility provided by ARM® Compiler 6

The compiler in ARM Compiler 6 is based on Clang and LLVM technology. As such, it provides a high degree of compatibility with GCC.

ARM Compiler 6 can build the vast majority of C code that is written to be built with GCC. However, ARM Compiler is not 100% source compatible in all cases. Specifically, ARM Compiler does not aim to be bug-compatible with GCC. That is, ARM Compiler does not replicate GCC bugs.

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.

The environment variables used by the toolchain are described in the following table.

Where an environment variable is identified as GCC compatible, the GCC documentation provides full information about that environment variable. Search for "Environment Variables Affecting GCC" on the GCC web site, `gcc.gnu.org`.

Table 1-1 Environment variables used by the toolchain

Environment variable	Setting
`ARMCOMPILER6_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.
`ARMCOMPILER6_CLANGOPT`	An optional environment variable to define additional `armclang` options that are to be used outside your regular makefile. The options listed appear before any options specified for the `armclang` command in the makefile. Therefore, any options specified in the makefile might override the options listed in this environment variable.
`ARMCOMPILER6_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.
`ARMCOMPILER6_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.
`ARMROOT`	Your installation directory root, *`install_directory`*.
`ARMLMD_LICENSE_FILE`	This environment variable must be set, and specifies the location of your ARM license file. See the *ARM® DS-5™ License Management Guide* for information on this environment variable. **Note** On Windows, the length of `ARMLMD_LICENSE_FILE` must not exceed 260 characters.
`C_INCLUDE_PATH`	GCC compatible environment variable. Adds the specified directories to the list of places that are searched to find included C files.

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

Environment variable	Setting
COMPILER_PATH	GCC compatible environment variable. Adds the specified directories to the list of places that are searched to find subprograms.
CPATH	GCC compatible environment variable. Adds the specified directories to the list of places that are searched to find included files regardless of the source language.
CPLUS_INCLUDE_PATH	GCC compatible environment variable. Adds the specified directories to the list of places that are searched to find included C++ files.
TMP	Used on Windows platforms to specify the directory to be used for temporary files.
TMPDIR	Used on Red Hat Linux platforms to specify the directory to be used for temporary files.

1.10 ARM architectures supported by the toolchain

ARM Compiler supports both the AArch64 and AArch32 states of the ARMv8 architecture.

When compiling code, the compiler needs to know which architecture to target in order to take advantage of features specific to that architecture.

To specify a target architecture with `armclang`, use the `--target` command-line option:

`--target=`*`arch-vendor-os-env`*

Supported targets are as follows:

`aarch64-arm-none-eabi`
: The AArch64 state of the ARMv8 architecture. This is the default target.

`armv8a-arm-none-eabi`
: The AArch32 state of the ARMv8 architecture.

Note

The `--target` option is an `armclang` option. For all of the other tools, such as `armasm` and `armlink`, use the `--cpu`, `--fpu`, and `--device` options to specify target processors and architectures.

1.11 ARM® Compiler and virtual address space

ARM Compiler provides 64-bit versions of `armclang` and `armlink`.

`armasm`, `armar` and `fromelf` are only available as 32-bit applications. This limits the virtual address space and file size available to those tools. If these limits are exceeded, the tool 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.

The 64-bit `armclang` and `armlink` tools can utilize the greater amount of memory available on 64-bit host machines.

Related references

1.3 Host platform support for ARM® Compiler on page 1-13.

1.12 Compilation tools command-line option rules

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

`armclang` option rules

`armclang` follows the same syntax rules as GCC. Some options are preceded by a single dash `-`, others by a double dash `--`. Some options require an `=` character between the option and the argument, others require a space character.

`armasm`, `armar`, `amlink`, and `fromelf` command-line syntax rules

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:

```
armar -r -a obj1.o mylib.a obj2.o
```

```
armar -r -aobj1.o mylib.a obj2.o
```

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:

```
armlink myfile.o --feedback=fb.txt
```

```
armlink myfile.o --feedback fb.txt
```

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

Command-line syntax rules common to all tools

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 link a file named `-ifile_1`, use:

```
armlink -- -ifile_1
```

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

```
armlink obj1.o --keep='s.o(vect)'
```

Migration tools command-line syntax

The ARM Compiler Source Compatibility Checker has a small number of keyword options, all of which are preceded by a double dash `--`.

The command-line translation wrapper conforms to the syntax rules for older compiler versions.

1.13 ARM® Compiler migration tools

Migration tools help users migrate from older versions of ARM Compiler to ARM Compiler 6.

ARM provides the following migration tools:

- The ARM Compiler Source Compatibility Checker lets you check whether C or C++ source code that compiles with previous versions of the ARM Compiler is compatible with ARM Compiler 6.

  The ARM Compiler Source Compatibility Checker is available separately from ARM.
- The command-line translation wrapper automatically converts some of the most common command-line options from older `armcc` compiler versions to the equivalent command line options for ARM Compiler 6 compiler (`armclang`).

  The wrapper invokes `armclang` with the translated options, allowing you to invoke `armclang` with `armcc` options, as a help to migrate projects from older compiler versions.

  The wrapper is provided as a user-customizable Python script so that you can modify the translation rules as needed.

Related information

Migration and Compatibility Guide.

1.14 ARM® Compiler package structure

The structure of the ARM Compiler distribution package is as follows.

```
ARMCompiler6.00/
 |- bin/
 |- documents/
 |- include/
 |- lib/
 |- sw/
    |- info/
    |- mappings/
    |- migration/
    |- python2.7
```

`bin/`
: Contains all binary tools.

`documents/`
: Contains documentation. This document can be found in this directory.

`include/`
: Contains C/C++ header files.

`lib/`
: Contains all library files.

`sw/info/`
: Contains the README and information about third-party software licenses.

`sw/mappings/`
: Contains license mapping files.

`sw/migration/`
: Contains support tools for migration from older compiler versions to ARM Compiler 6.

`sw/python2.7/`
: Contains the Python 2.7 distribution.

1.15 Compiler command-line options

Describes the most common ARM Compiler command-line options.

ARM Compiler provides many command-line options, including most Clang command-line options as well as a number of ARM-specific options. Additional information about command-line options is available:

- The *armclang Reference Guide* provides more detail about a number of command-line options.
- For a full list of Clang command-line options, consult the Clang and LLVM documentation.

Table 1-2 Compiler command-line options

Option	Description
`-c`	Performs the compilation step, but not the link step.
`-xc -std=c90`	Enables the compilation of C90 source code. These are positional arguments and only affect subsequent input files on the command line.
`-xc -std=c99`	Enables the compilation of C99 source code. These are positional arguments and only affect subsequent input files on the command line.
`-xc++ -std=c++98`	Enables the compilation of C++ source code. These are positional arguments and only affect subsequent input files on the command line.
`--target=`*`arch-vendor-os-env`*	Enables code generation for the selected ARM architecture. Valid values for `--target` are `aarch64-arm-none-eabi` to target the AArch64 state of the ARMv8 architecture, or `armv8a-arm-none-eabi` to target the AArch32 state of the ARMv8 architecture.
`-marm`	Requests that the compiler targets the A32 instruction set when compiling for AArch32 state, `--target=armv8a-arm-none-eabi --marm` for example. The `-marm` option is only valid with AArch32 targets, for example `--target=armv8a-arm-none-eabi`. The compiler ignores the `-marm` option and generates a warning with AArch64 targets.
`-mthumb`	Targets the T32 instruction set when compiling for AArch32 state, `--target=armv8a-arm-none-eabi -mthumb` for example. The `-mthumb` option is only valid with AArch32 targets, for example `--target=armv8a-arm-none-eabi`. The compiler ignores the `-mthumb` option and generates a warning with AArch64 targets.
`-D`	Defines a preprocessing macro.
`-E`	Executes only the preprocessor step.

Table 1-2 Compiler command-line options (continued)

Option	Description
`-I`	Adds the specified directories to the list of places that are searched to find included files.
`-finline-functions`	Enables inlining of functions.
`-Xlinker`	Specifies command-line options to pass to the linker when a link step is being performed after compilation.
`-M`	Instructs the compiler to produce a list of makefile dependency lines suitable for use by a make utility.
`-o`	Specifies the name of the output file.
`-O`*num*	Specifies the level of optimization to be used when compiling source files.
`-Oz / -Os`	Performs optimizations to reduce image size at the expense of a possible increase in execution time. `-Os` balances code size against code speed. `-Oz` optimizes for code size. By default, the compiler performs optimizations to reduce execution time at the expense of a possible increase in image size.
`-S`	Outputs the disassembly of the machine code generated by the compiler.
`-v`	Shows how the compiler processes the command line. The commands are shown normalized, and the contents of any via files are expanded.
`-fvectorize`	Enables the generation of Advanced SIMD vector instructions directly from C or C++ code.
`-g`	Generates DWARF debug tables.
`--version`	Displays version information and license details.

1.16 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*

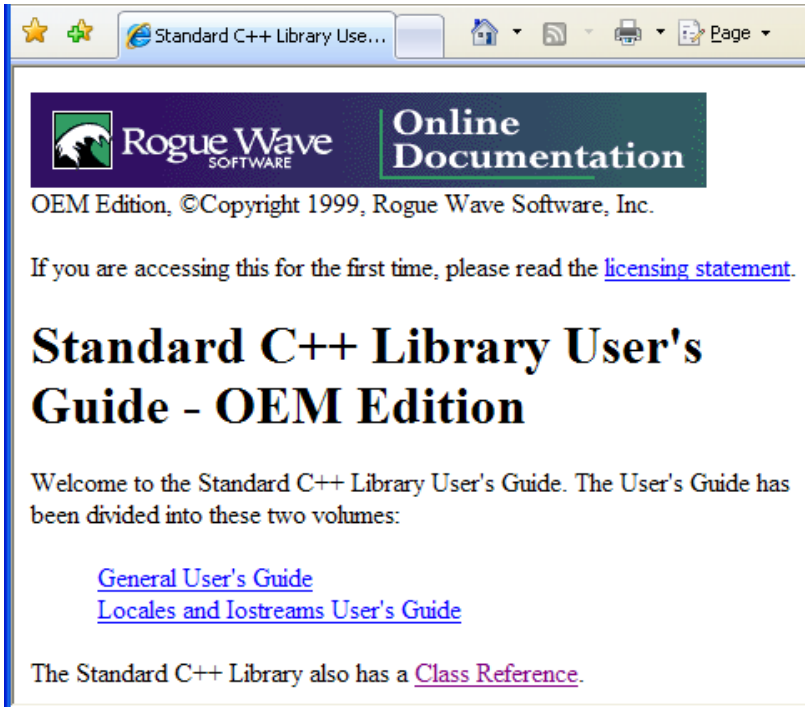

Figure 1-1 Rogue Wave HTML documentation

Related concepts

1.17 Clang and LLVM documentation on page 1-30.

Related references

1.4 About the toolchain documentation on page 1-14.

1.18 Further reading on page 1-31.

1.17 Clang and LLVM documentation

ARM Compiler is based on Clang and LLVM compiler technology.

The *Clang Compiler User's Manual* provides documentation for Clang. Search for "Clang compiler user manual" on the LLVM Compiler Infrastructure Project web site, `llvm.org`.

Related concepts

1.16 Rogue Wave documentation on page 1-29.

Related references

1.4 About the toolchain documentation on page 1-14.

1.18 Further reading on page 1-31.

Related information

The LLVM Compiler Infrastructure Project.

1.18 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*.

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.

Related concepts

1.16 Rogue Wave documentation on page 1-29.

1.17 Clang and LLVM documentation on page 1-30.

Related references

1.4 About the toolchain documentation on page 1-14.

Chapter 2
Creating an Application

Describes how to create an application using ARM Compiler.

It contains the following sections:

2.1 Introduction to 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:

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

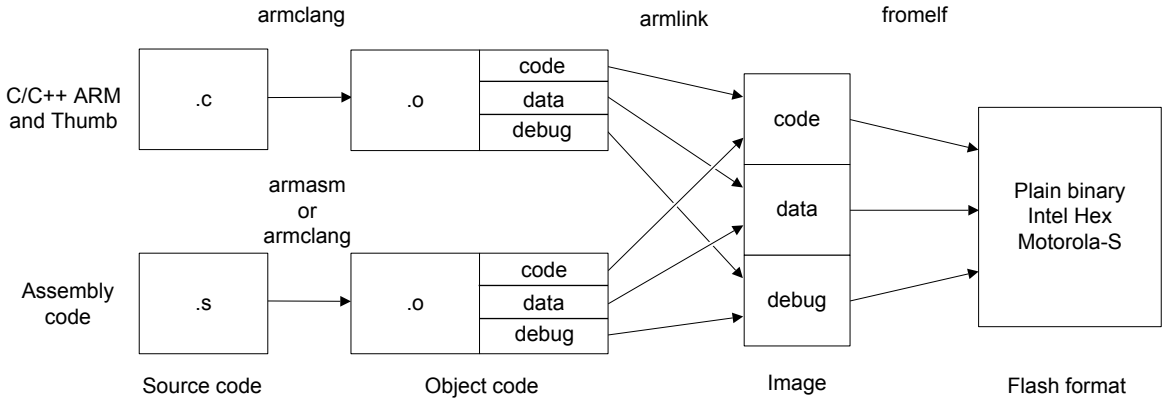

Figure 2-1 A typical tool usage flow diagram

Note

`armasm` is only supported for backwards compatibility. Use `armclang` for any new assembly code.

2.2 The ARM compiler command

The compiler, `armclang`, can compile C and C++ source code into A32 instructions and T32 instructions for AArch32 targets, or A64 instructions for AArch64 state targets.

Typically, you invoke the compiler as follows:

`armclang [`*`options`*`] `*`file_1 … file_n`*

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

2.3 Building an image from C source

This example shows how to build an image from C code with `armclang` and `armlink`.

To compile a C source file `hello_world.c`:

Procedure

1. Compile the C source with the following command:

```
armclang -c -O1 -o hello_world.o -xc -std=c90 -g hello_world.c
```

The following options are commonly used:

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

`-xc`
Tells the compiler that the source is C code.

Note

The compiler infers the source code language from the filename extension. The `-x` option overrides the inferred language setting. The `-x` option is not required for this example, it is provided for illustrative purposes only.

`-std=c90`
Tells the compiler that the source is ISO C90 C code.

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

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

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

Note

Be aware that the default target is `aarch64-arm-none-eabi`, with the compiler generating A64 instructions.

2. Link the file:

```
armlink hello_world.o --force_scanlib --info totals -o hello_world.axf
```

The `--force_scanlib` option tells `armlink` to link with the ARM libraries. The compiler does not include `$$Lib$Request` symbols when building objects, so by default (that is, if you do not specify `--force_scanlib`) `armlink` does not automatically link with the ARM libraries, resulting in an `L6411E` error.

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

Related concepts

The ARM compiler command.

Related tasks

2.5 Linking an object file (armclang) on page 2-38.

2.4 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`

2.5 Linking an object file (armclang)

This example shows how to link an object file with `armlink`.

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

```
armlink --force_scanlib --info totals -o hello_world.axf hello_world.o
```

where:

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

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

`--force_scanlib`
: Link with the ARM libraries.

  The compiler does not include `$$Lib$Request` symbols when building objects, so by default `armlink` does not automatically link with the ARM libraries, resulting in the following messages:

```
Warning: L6665W: Neither Lib$$Request$$armlib Lib$$Request$$cpplib defined, not
searching ARM libraries.

Error: L6411E: No compatible library exists with a definition of startup symbol __main.
```

Related concepts

The ARM linker command.

2.6 The ARM assembler commands

ARM Compiler provides two assembler commands: `armclang` for GNU syntax assembly code, and `armasm` for legacy ARM syntax assembly code.

- `armclang` includes an assembler for GNU syntax assembly language source code. Use GNU syntax for all new assembly source code, and use `armclang` to assemble these source files.

  Typically, you invoke the `armclang` assembler as follows:

  `armclang [options] file_1.s … file_n.s`

  You can specify one or more `.s` input files. `armclang` automatically identifies from the `.s` suffix that the input file contains assembly code, and assembles the source files. `armclang` produces one object file for each source input file.
- `armasm` assembles ARM assembly code. You should only use `armasm` for legacy assembly files.

  Typically, you invoke the `armasm` assembler as follows:

  `armasm [options] file_1.s … file_n.s`

Related tasks

2.8 Building an image from legacy ARM syntax assembly code on page 2-41.

2.7 Building an image from GNU syntax assembly code

This example shows how to build GNU syntax assembly language code with `armclang`.

Note

Use GNU syntax for all new assembly source code, and use `armclang` to assemble these source files. Only use ARM syntax and `armasm` for legacy assembly files.

To build an assembly program, for example `hello_world.s`:

Procedure

1. Assemble the source file:

   ```
   armclang hello_world.s
   ```

   `armclang` assembles the source file and automatically calls the linker to produce an executable image, `a.out`.
2. Use an ELF and DWARF 4 compatible debugger to load and run the image.

   Step through the program and examine the registers to see how they change.

2.8 Building an image from legacy ARM syntax assembly code

This example shows how to build ARM assembly language code with `armasm`.

Note

Only use `armasm` for legacy assembly files. Use `armclang` to assemble new assembly files.

To build an assembly program, for example `hello_world.s`:

Procedure

1. Assemble the source file:

```
armasm --debug hello_world.s
```

2. Link the object file:

```
armlink hello_world.o -o hello_world.axf
```

3. Use an ELF and DWARF 4 compatible debugger to load and run the image.
   Step through the program and examine the registers to see how they change.

Related concepts

2.6 The ARM assembler commands on page 2-39.

2.9 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.
