# ARM® Compiler toolchain

**Version 5.02**

# Building Linux Applications with the ARM Compiler toolchain and GNU Libraries

ARM®


ARM DUI 0483H (ID070912)

# ARM Compiler toolchain
## Building Linux Applications with the ARM Compiler toolchain and GNU Libraries



### Release Information

The following changes have been made to this book.

**Change History**

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

### 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 Building Linux Applications with the ARM Compiler toolchain and GNU Libraries

# 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 0483H
- 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 Information Center, http://infocenter.arm.com/help/index.jsp
- ARM Technical Support Knowledge Articles, http://infocenter.arm.com/help/topic/com.arm.doc.faqs/index.html
- ARM Support and Maintenance, http://www.arm.com/support/services/support-maintenance.php
- ARM Glossary, http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html.

# Chapter 2
# About building Linux applications with the ARM Compiler toolchain and GNU libraries

The following topics give an overview of building a Linux application or library with the ARM Compiler toolchain, and describe limitations and requirements:

**Concepts**

## 2.1 About the ARM Compiler toolchain and GNU libraries

The ARM Compiler toolchain enables you to create dynamic images that can run under Linux using the header files and libraries from the GNU C library (`glibc`).

The instructions assume that you are familiar with ARM Compiler toolchain, GNU toolchain, and Linux.

### 2.1.1 See also

**Tasks**

- Chapter 3 *Using the ARM Compiler toolchain to build a Linux application or library*.

**Concepts**

- *Expected use cases for developing Linux applications* on page 2-3
- *Limitations when building Linux applications* on page 2-4
- *Target processor requirements for building Linux applications* on page 2-6
- *Build requirements for Linux applications* on page 2-7
- *About the ARM Application Binary Interface* on page 2-8
- *Interactions between mixed-ABI components* on page 2-9.

**Other information**

- GNU ARM toolchain, `http://www.gnuarm.com`
- GCC, the GNU Compiler Collection, `http://gcc.gnu.org`
- GNU Operating System, `http://www.gnu.org`.

## 2.2 Expected use cases for developing Linux applications

The following are the expected use cases for developing Linux applications and libraries with the ARM Compiler toolchain and GNU libraries:

- building a standalone Linux application with the ARM Compiler toolchain
- building static and shared libraries with the ARM Compiler toolchain, and linking these to an application built with the ARM Compiler toolchain
- building a static or shared library with the ARM Compiler toolchain, and linking this to an application built with the GNU toolchain
- migrating an existing Linux application build using the ARM Compiler toolchain, retaining explicit search paths on the command line
- migrating an existing Linux application build using the ARM Compiler toolchain, using a standard configuration of system search paths and libraries
- using `armcc` and `armlink` as drop-in replacements for GCC and GNU ld using command-line translation.

### 2.2.1 See also

**Reference**

*Compiler Reference*:

- *--arm_linux* on page 3-16
- *--arm_linux_paths* on page 3-21
- Chapter 3 *Compiler Command-line Options*.

*Linker Reference*:

- *--arm_linux* on page 2-13
- Chapter 2 *Linker command-line options*.

## 2.3 Limitations when building Linux applications

There are several limitations on interoperation between the GNU tools and libraries and the ARM Compiler toolchain:

**GNU tool and library compatibility with the ARM Compiler toolchain**

Be aware of the following compatibility requirements:

- CodeSourcery 2008q1 is the earliest recommended release.
- CodeSourcery 2005q3 is the earliest release that provides basic interoperation with the ARM Compiler toolchain, however, releases earlier than 2008 have known interoperability issues and you might encounter problems using them.
- If libraries from a distribution or mainline GNU toolchain build are used, their sources tend to be behind CodeSourcery releases in terms of bug fixes, so a more recent version might be required to avoid interoperation problems.
- You must use the 2005-q3-2 release of the CodeSourcery tools (or a later release). Because of updates in the ARM *Application Binary Interface* (ABI) ELF specification, the binary utilities (binutils) from this CodeSourcery release cannot consume object files built with the ARM Compiler toolchain. Support for the new ELF ABI revision is in the 2006-q1 and later releases.
- There are slight implementation differences in the way C++ exceptions are handled between the ARM Compiler toolchain and GCC. Because of these differences, the GNU C/C++ library prior to the CodeSourcery 2007-q1-10 release did not support code generated by the ARM Compiler toolchain that used C++ exceptions. Therefore, to use C++ exceptions you must use the CodeSourcery 2007-q1-10 release or later. This includes using these libraries on the filesystem of your target.

**The ARM Compiler toolchain cannot be used for building the Linux kernel or kernel-based code, such as device drivers or other kernel modules**

This is because a significant portion of the kernel code is written in assembly language using the *GNU assembler* (GAS) syntax. This is incompatible with `armasm`, and there is no performance gain to be made from rebuilding such code with a different assembler.

Also, the function interfaces for the kernel code prior to version 2.6.16 have not been written to comply with the ABI. This means that drivers and other kernel modules cannot be compiled using the ARM Compiler toolchain because there are no guarantees that calls would be made correctly between the kernel and the driver code. You must use the GNU toolchain when building the kernel and kernel modules.

**ARM architecture v4T is not fully supported**

See *Target processor requirements for building Linux applications* on page 2-6.

### 2.3.1 Unsupported GCC features

The following GCC features are not supported:

- assembly source, both inline assembly and separate *GNU assembler* (GAS) source files
- nested functions
- frame pointers.

### 2.3.2 See also

**Other information**

- GNU ARM toolchain, http://www.gnuarm.com
- GCC, the GNU Compiler Collection, http://gcc.gnu.org

## 2.4 Target processor requirements for building Linux applications

Building Linux applications is aimed at *ARM architecture v5TE* (ARMv5TE™) or later processors, such as the ARM926EJ-S™ and ARM1176JZ-S™ processors. This is because the ARM ABI uses ARMv5TE as its base architecture, and earlier architecture versions are not fully covered by the ABI.

You might be able to use the ARM Compiler toolchain to build Linux applications for *ARM architecture v4T* (ARMv4T™) processors, such as the ARM720T™ and ARM920T™ processors. This is, however, entirely at your own risk and is not supported. In particular, you cannot use Thumb code built for ARMv4T in shared libraries. ARM recommends that you only use the GNU toolchain when building Linux applications for ARMv4T targets.

The filesystem on your target must contain the ABI-compliant library binaries that are included in the CodeSourcery GNU toolchain releases.

Also, the target must be running a Linux kernel with:

- support for the *Native POSIX Threading Library* (NPTL), that is the more recent mechanism for supporting multithreaded code under Linux with the GNU C library
- thread-local storage (TLS).

For the mainstream kernel source, this means that your target must be running version 2.6.12 (or later) of the Linux kernel. Your Linux distribution, however, might have applied the appropriate patches to its release of an earlier kernel. For more details, contact your Linux distributor.

Prebuilt binary images of the Linux kernel configured for the ARM development boards can be found on the ARM website.

### 2.4.1 See also

**Reference**

- *Build requirements for Linux applications* on page 2-7
- *About the ARM Application Binary Interface* on page 2-8.

**Other information**

- Linux Support for the ARM Architecture, `http://www.arm.com/community/software-enablement/linux.php`.

## 2.5 Build requirements for Linux applications

All information in *Building Linux Applications with the ARM Compiler toolchain and GNU Libraries* relates to the use of the ARM Compiler toolchain.

The CodeSourcery 2005-q1 release was the first to permit *Embedded Application Binary Interface* (EABI) compliant interoperation between ARM Compiler toolchain and the GNU toolchain. However, several enhancements and fixes have been made since the CodeSourcery 2005-q1 release. Therefore, ARM recommends that you only use the CodeSourcery 2008q1 or later releases for interoperating with the ARM Compiler toolchain, and the instructions in this document relate only to these releases.

Therefore, the instructions in this document relate only to the CodeSourcery 2006-q1-6 and later releases, because it is now simpler and safer to link with a newer release.

Your ARM Linux distribution might already use the CodeSourcery toolchain or have the appropriate patches applied. For more details, contact your ARM Linux distributor.

### 2.5.1 See also

**Other information**

- Application Binary Interface (ABI) for the ARM Architecture, http://infocenter.arm.com/help/topic/com.arm.doc.subset.swdev.abi
- CodeSourcery binary and source packages for the GNU toolchains, http://www.codesourcery.com/gnu_toolchains/arm

## 2.6 About the ARM Application Binary Interface

The *Application Binary Interface* (ABI) for the ARM Architecture is a collection of standards, some open and some specific to the ARM architecture. These standards regulate the interoperation of binary code, development tools, and a spectrum of ARM processors-based execution environments from bare metal to platform operating systems such as ARM Linux.

A third-party toolchain such as the GNU tools must comply with the standards given in the ABI for its objects to link and interoperate correctly with those produced by the ARM Compiler toolchain. The CodeSourcery release of the GNU tools is specifically tailored to fully support the ARM ABI and permit objects produced using both the ARM Compiler toolchain and the GNU toolchain to work together successfully.

### 2.6.1 See also

**Other information**

- Application Binary Interface (ABI) for the ARM Architecture, http://infocenter.arm.com/help/topic/com.arm.doc.subset.swdev.abi
- CodeSourcery binary and source packages for the GNU toolchains, http://www.codesourcery.com/gnu_toolchains/arm

## 2.7 Interactions between mixed-ABI components

If you are not using an *Application Binary Interface* (ABI) compliant kernel, you might need to build a mixed ABI system. Kernels before version 2.6.16 can only be built using the legacy GNU ABI (use GCC option `-mabi=apcs-gnu` when using the CodeSourcery toolchain). This includes all kernel modules and device drivers.

This can cause problems when your applications or libraries must interface directly with kernel structures or functions (system calls), including through the use of a shared header file describing kernel structures. In this case, you must use assembly code or modified descriptions of the structures to translate between the two ABIs when calling kernel functions or manipulating kernel data structures in your applications or libraries.

From kernel 2.6.16 onwards, you can build the Linux kernel using the new ARM *Embedded Application Binary Interface* (EABI). This enables easier integration of applications and libraries to form a completely EABI-compliant system.

### 2.7.1 See also

**Other information**

- Application Binary Interface (ABI) for the ARM Architecture, http://infocenter.arm.com/help/topic/com.arm.doc.subset.swdev.abi

# Chapter 3
# Using the ARM Compiler toolchain to build a Linux application or library

The following topics describe how to use the ARM Compiler toolchain to build a Linux application or library:

**Tasks**



**Concepts**

- *Additional headers from the ARM Compiler toolchain* on page 3-18.

## 3.1 About using the ARM Compiler toolchain to build a Linux application or library

There are several possible routes to producing a Linux application or library with the ARM Compiler toolchain, depending on the requirements of your build. In most cases, you must configure the tools based on an existing GNU toolchain or by providing an alternate location for system header files and libraries. When the tools are configured, you can use the ARM Compiler toolchain in one of these ways:

- Use the tools directly to produce an application using the standard configuration, but with normal ARM Compiler toolchain command-line options.
- Use ARM Compiler toolchain as a drop-in replacement for GCC and the GNU linker.
- Migrate a build from RVCT to use the new features in the ARM Compiler toolchain.

### 3.1.1 See also

**Tasks**

- *Building for ARM Linux using normal ARM Compiler toolchain options* on page 3-8
- *Using the ARM Compiler toolchain as a drop-in replacement for GCC and GNU Id* on page 3-9
- *Migrating a build from an earlier version of the ARM tools* on page 3-13.

**Concepts**

- *Configuration of the ARM Compiler toolchain for Linux applications* on page 3-4.

## 3.2 Configuration of the ARM Compiler toolchain for Linux applications

When building for ARM Linux, your library configuration determines:

- the paths you must use to the appropriate system header files and libraries (for example, `glibc` and `libstdc++`)
- the appropriate standard options and object files (such as those containing the application entry point function and C library initialization code).

Configuration of the ARM Compiler toolchain can be performed:

- automatically with `armcc`
- manually by specifying particular paths that are used for finding header files and libraries.

**Automatic configuration**

The ARM compiler, `armcc`, can obtain the configuration information automatically from an existing GNU toolchain. The paths and options used remain the same unless you modify your libraries. Consequently, this configuration information is obtained once and stored in a configuration file that you specify on the command-line. The information in this configuration file is then re-used when compiling or linking for ARM Linux.

**Manual configuration**

You can configure the ARM Compiler toolchain manually by specifying particular paths that are used for finding header files and libraries. To configure ARM Compiler toolchain manually, you must specify:

- the `sysroot` path
- the path to the C++ header files.

The `sysroot` path is the root of the tree directory where header files and libraries are normally installed. For a native Linux filesystem, this is the root of the directory tree `/`. If you are configuring against a CodeSourcery distribution, or another self-contained cross-compilation GNU toolchain, this is typically the root of the directory tree where `glibc` was installed. For recent CodeSourcery releases, this is the `arm-none-linux-gnueabi/libc` subdirectory. If you are configuring against the filesystem of the target (for example, to pick up new libraries as they are built and installed into the target filesystem tree) the `sysroot` is the root of this -filesystem.

The C++ header file path is the path of the directory containing the header files from `libstdc++`. These are usually installed in a separate subdirectory from those normally searched relative to the `sysroot`, and must also be provided as part of the library configuration. In a CodeSourcery distribution, this is typically the `arm-none-linux-gnueabi/include/c++/`*`version`* subdirectory, where *`version`* is the GCC version. In a typical Linux filesystem installation, these might be installed in `/usr/include/c++/version`.

The configuration produced by this process is written to a configuration file that is used later when building for ARM Linux.

To specify the location of the configuration file, use the `--arm_linux_config_file=`*`path`* compiler option, where *`path`* is the filename of the configuration file. The *`path`* must include the full path to the configuration file if the file is not located in the same directory as `armcc`. You must specify this both when producing the configuration file and when using the configuration during compilation or linking.

### 3.2.1 See also

**Tasks**

- *Configuring the ARM Compiler toolchain automatically* on page 3-6
- *Configuring the ARM Compiler toolchain manually* on page 3-7.

**Reference**

*Compiler Reference*:

- *--arm_linux_config_file=path* on page 3-18.

**Other information**

- ARM Linux Internet Platform, `http://linux.onarm.com`
- GNU ARM toolchain, `http://www.gnuarm.com`.

## 3.3 Configuring the ARM Compiler toolchain automatically

If GCC is in a directory listed in the `PATH` environment variable, you can configure the tools using the command:

```
armcc --arm_linux_configure --arm_linux_config_file=config_file_path
```

If GCC is not on your system path, you can specify this explicitly:

```
armcc --arm_linux_configure --arm_linux_config_file=config_file_path
--configure_gcc=path_to_gcc
```

where *`path_to_gcc`* is the path and filename of the GCC driver binary, that is, the actual `gcc` executable (with `.exe` suffix on Windows). For a cross-compiler the filename is, for example, `arm-none-linux-gnueabi-gcc` (with `.exe` suffix on Windows).

During configuration, the compiler also determines the location of the GNU linker used by GCC and queries the linker for additional information. If this cannot be determined, or you want to override the normal path to the GNU linker, you can specify this using the `--configure_gld=`*`path_to_gld`* option, where *`path_to_gld`* is the complete path and filename of the GNU ld binary.

You can also manually:

- override the `sysroot` path
- override the location of the C++ header files
- specify additional search paths for header files and libraries.

### 3.3.1 See also

**Tasks**

- *Configuring the ARM Compiler toolchain manually* on page 3-7.

**Concepts**

- *Configuration of the ARM Compiler toolchain for Linux applications* on page 3-4.

**Reference**

*Compiler Reference*:

- *--arm_linux_config_file=path* on page 3-18
- *--arm_linux_configure* on page 3-19
- *--configure_gld=path* on page 3-45.

## 3.4 Configuring the ARM Compiler toolchain manually

To configure the tools manually, use:

```
armcc --arm_linux_configure --configure_sysroot=sysroot_path
--configure_cpp_headers=headers_path --arm_linux_config_file=filename
```

You can also specify additional header search paths and library search paths as a comma-separated list using the `--configure_extra_includes=list` and `--configure_extra_libraries=list` options.

To manually configure against a CodeSourcery distribution, you must provide extra library paths for the GCC support libraries, because these are not packaged in the `glibc` sysroot. For example, you can use a command similar to the following:

```
armcc --arm_linux_configure --arm_linux_config_file=filename
--configure_sysroot=codesourcery_root/arm-none-linux-gnueabi/libc
--configure_cpp_headers=codesourcery_root/arm-none-linux-gnueabi/include/c++/gcc_versio
n
--configure_extra_libraries=codesourcery_root/lib/gcc/arm-none-linux-gnueabi/gcc_versio
n,codesourcery_root/arm-none-linux-gnueabi/lib
```

### 3.4.1 See also

**Tasks**

- *Configuring the ARM Compiler toolchain automatically* on page 3-6.

**Concepts**

- *Configuration of the ARM Compiler toolchain for Linux applications* on page 3-4.

**Reference**

*Compiler Reference*:

- *--arm_linux_config_file=path* on page 3-18
- *--arm_linux_configure* on page 3-19
- *--configure_cpp_headers=path* on page 3-39
- *--configure_extra_includes=paths* on page 3-40
- *--configure_extra_libraries=paths* on page 3-41
- *--configure_sysroot=path* on page 3-46.

**Other information**

- CodeSourcery, `http://www.codesourcery.com`
- GNU ARM toolchain, `http://www.gnuarm.com`.

## 3.5 Building for ARM Linux using normal ARM Compiler toolchain options

After you have configured ARM tools, you can use the configuration to build code for ARM Linux using the `--arm_linux_paths` compiler option. This follows the typical GCC usage model where the compiler driver is used to control linking and selection of standard system object files and libraries. You must also specify the location of the configuration file with `--arm_linux_config_file=`*`filename`*. Using these options, you can build application code directly. For example, to build the Hello World example:

```
armcc --arm_linux_paths --arm_linux_config_file=filename -o hello hello.c
```

To create a shared library, compile and link your code using `--apcs=/fpic --shared`. The compiler provides the `--shared` option to select variants of the system object files and libraries from the configuration that are suitable for linking into a shared library.

For example, to compile a source file `source.c` suitable for use in a shared library:

```
armcc --arm_linux_paths --arm_linux_config_file=filename --apcs=/fpic -c source.c
```

To link two object files `obj1.o` and `obj2.o` into a shared library `libexample.so`:

```
armcc --arm_linux_paths --arm_linux_config_file=filename --shared -o libexample.so obj1.o
obj2.o source.o
```

**Note**

When linking a C++ application with `--arm_linux_paths`, you must specify the `--cpp` option to the compiler driver so that it passes the appropriate C++ libraries to the linker.

### 3.5.1 See also

**Reference**

*Compiler Reference*:

- *--apcs=qualifer...qualifier* on page 3-11
- *--arm_linux_config_file=path* on page 3-18
- *--arm_linux_paths* on page 3-21
- *-o filename* on page 3-154
- *--shared* on page 3-188.

**Other information**

- ARM Linux Internet Platform, `http://linux.onarm.com`
- GNU ARM toolchain, `http://www.gnuarm.com`.

## 3.6 Using the ARM Compiler toolchain as a drop-in replacement for GCC and GNU ld

You can use the ARM Compiler toolchain as a replacement for GCC and GNU ld. In GCC emulation mode, `armcc` accepts command lines intended for GCC and GNU ld and translates these internally into standard `armcc` and `armlink` command lines.

### 3.6.1 See also

**Concepts**

- *GCC emulation mode in armcc* on page 3-10
- *Passing normal armcc options in GNU emulation mode* on page 3-11
- *Differences in behavior and limitations between GCC and armcc emulation mode* on page 3-12

## 3.7 GCC emulation mode in armcc

The ARM compiler, `armcc`, supports a GCC emulation mode. In this mode, `armcc` accepts command lines intended for GCC and GNU ld and translates these internally into standard `armcc` and `armlink` command lines. This follows the typical GCC usage model of using the compiler driver to direct linking, rather than invoking the linker directly. However, `armcc` does provide support for being invoked as if emulating GNU ld directly, and reports itself as the linker if invoked in GCC emulation mode with `--print-prog-name=ld`. This is primarily intended to support a limited number of cases where the linker is invoked directly by existing build scripts targeting the GNU tools, for example in a partial link step.

To enable emulation of GCC, invoke `armcc` with one of the following options:

- `--translate_gcc` to emulate `gcc`
- `--translate_g++` to emulate `g++`
- `--translate_gld` to emulate GNU ld.

You must also provide `--arm_linux_config_file=`*`filename`* to give a location for the configuration file.

**Note**

If you do not provide a configuration file with the `--arm_linux_config_file` option when in translation mode, the compiler performs translation of options but does not set any defaults for ARM Linux, including ABI defaults such as enum size. This mode of operation is provided for convenience and is not intended for building Linux applications.

### 3.7.1 See also

**Reference**

*Compiler Reference*:

- *--arm_linux_config_file=path* on page 3-18
- *--translate_g++* on page 3-198
- *--translate_gcc* on page 3-200
- *--translate_gld* on page 3-202.

**Other information**

- ARM Linux Internet Platform, `http://linux.onarm.com`.

## 3.8 Passing normal armcc options in GNU emulation mode

To take advantage of features specific to `armcc`, you can pass normal compilation tools options to the compiler when in GCC emulation mode. To do this, use `-Warmcc,option,...` This is a fake GCC-like option that accepts a comma-separated list of `armcc` options. These options are passed verbatim to the compiler, and are appended to the translated command line so that they can override any translation options.

If you want to pass options to `armlink` from `armcc`, use the `-L` option:

```
armcc -Warmcc -Loption1 -Loption2
```

### 3.8.1 See also

**Reference**

*Compiler Reference*:

- *-Lopt* on page 3-127
- *-Warmcc,option[,option,...]* on page 3-221.

*Linker Reference*:

- Chapter 2 *Linker command-line options*.

## 3.9 Differences in behavior and limitations between GCC and armcc emulation mode

There are some differences in behavior between GCC and the emulation mode supported by `armcc`:

- If no optimization level is specified, the GCC default (`-O0`) is used.
- If a GCC numeric optimization level (`-O0` through `-O3`) is used, this is translated into `-On -Otime` for `armcc`. The GCC `-Os` option translates as `-O3 -Ospace`.

  To force the use of a specific `armcc` optimization level, include the `-Warmcc` option. For example:

  ```
  -Warmcc,-O1,-Ospace
  ```
- Support for diagnostic control is limited. In particular, warnings are suppressed by default (similar to GCC) and are re-enabled with `-W` or `-Wall`. The `-w` option (lower case `-w`) is supported to suppress warnings, for example to override a `-Wall` earlier on the command line. Other GCC `-W…` options are ignored. If you require control of individual messages then you can use the normal compilation tools options, such as `-Warmcc,--diag_suppress` and `-Warmcc,--diag_error`.
- Many GCC options do not have an equivalent in `armcc`. These include, for example, many of the `-f...` GCC options that control optimization phases that are specific to the GCC code generator, and are not applicable to `armcc`. Any GCC options that do not have an equivalent in `armcc` are silently ignored. To see the GCC options that are ignored, specify the options `-Warmcc --remarks`.

### 3.9.1 See also

**Reference**

*Compiler Reference*:

- *--diag_error=tag[,tag,...]* on page 3-70
- *--diag_suppress=optimizations* on page 3-74
- *-Onum* on page 3-156
- *-Ospace* on page 3-160
- *-Otime* on page 3-161
- *--remarks* on page 3-181
- *-W* on page 3-220
- *-Warmcc,option[,option,...]* on page 3-221.

## 3.10 Migrating a build from an earlier version of the ARM tools

Existing code for ARM Linux that builds successfully using RVCT v3.0 or RVCT v3.1 can work without changes. However, you can take advantage of the new features in ARM Compiler v4.1, and later, to simplify your makefiles or other build scripts. For example, use:

- `--arm_linux` to configure a set of options with defaults that are suitable for ARM Linux compilation
- `--arm_linux_paths` to take advantage of the configuration capabilities in the ARM Compiler toolchain.

In ARM Compiler v4.1, and later, you do not have to link with the helper libraries, such as `h_5.1`. If you are recompiling an entire project from source, the required functions are generated in the object files by the compiler. If you are linking with legacy object files compiled using a previous version of the ARM Compiler toolchain, you must still link with an appropriate helper library.

### 3.10.1 See also

**Concepts**

- *Minimal migration path without using a configuration file* on page 3-14
- *Migration using a configuration* on page 3-16.

**Reference**

*Compiler Reference*:

- *--arm_linux* on page 3-16
- *--arm_linux_paths* on page 3-21.

*Linker Reference*:

- *--arm_linux* on page 2-13.

**Other information**

- ARM Linux Internet Platform, `http://linux.onarm.com`.

## 3.11 Minimal migration path without using a configuration file

The compiler and linker both provide the `--arm_linux` command-line option for building ARM Linux applications. This does not require a configuration file, and enables a set of default configuration options, for example *Application Binary Interface* (ABI) variant options such as `--enum_is_int`. This permits you to simplify the compiler options used in existing makefiles while retaining full and explicit control over the header and library search paths used. When migrating a build from an earlier version of the ARM compilation tools, you can remove these standard switches from the list of those supplied to the compiler and linker with the single `--arm_linux` option.

The `--arm_linux` option in the compiler enables the following command-line options:

```
--gnu_defaults --enum_is_int --library_interface=aeabi_glibc --apcs=/interwork
--preinclude=linux_armcc.h
```

**Note**

Be aware that if you specify `--preinclude` manually, the compiler does not search the subdirectories of the default include path, or the path specified in `ARMCC`*n*`INC`. Therefore, to enable the compiler to locate the header file in the `arm_linux` subdirectory, you must also include the `--arm_linux` command-line option.

The `--gnu_defaults` option (implied by `--arm_linux`) in the compiler enables the following command-line options:

```
--gnu --wchar32 --no_hide_all --signed_bitfields --no_debug_macros --allow_null_this
--no_implicit_include
```

The `--arm_linux` option in the linker enables the following command-line options:

```
--sysv --no_startup --no_ref_cpp_init --no_scanlib --keep=*(.init) --keep=*(.fini)
--keep=*(.init_array) --keep=*(.fini_array)
--linux_abitag=2.6.12--diag_suppress=6332,6318,6319,6765,6747,6420
```

### 3.11.1 See also

**Reference**

*Compiler Reference*:

- *--apcs=qualifer...qualifier* on page 3-11
- *--arm_linux* on page 3-16
- *--enum_is_int* on page 3-85
- *--gnu_defaults* on page 3-108
- *--hide_all, --no_hide_all* on page 3-113
- *--library_interface=lib* on page 3-128
- *--preinclude=filename* on page 3-172.

*Linker Reference*:

- *--arm_linux* on page 2-13
- *--diag_suppress=tag[,tag,...]* on page 2-47
- *--keep=section_id* on page 2-89
- *--linux_abitag=version_id* on page 2-101
- *--ref_cpp_init, --no_ref_cpp_init* on page 2-130
- *--scanlib, --no_scanlib* on page 2-141
- *--startup=symbol, --no_startup* on page 2-155.

*Introducing the ARM Compiler toolchain*:

- *Toolchain environment variables* on page 2-14.

**Other information**

- ARM Linux Internet Platform, http://linux.onarm.com.

## 3.12 Migration using a configuration

If you want to take advantage of the configuration capabilities in the ARM Compiler toolchain, you can create a configuration file.When this configuration file is created, you can modify an existing build by replacing the list of standard options and search paths with the `--arm_linux_paths` compiler option.

### 3.12.1 See also

**Tasks**

- *Building for ARM Linux using normal ARM Compiler toolchain options* on page 3-8.

**Concepts**

- *Configuration of the ARM Compiler toolchain for Linux applications* on page 3-4.

**Reference**

*Compiler Reference*:

- *--arm_linux_paths* on page 3-21.

**Other information**

- ARM Linux Internet Platform, `http://linux.onarm.com`.

## 3.13 Typical assembler command-line options

When using assembly code in your application or library, you usually have to specify the following `armasm` command-line options:

`--apcs=/interwork`

This instructs the assembler to set the build attributes in the object file to indicate that the code is ARM and Thumb interworking-safe.

`--no_hide_all`

This indicates that the assembler must use dynamic import and export for all global symbols.

ARM also recommended that you specify a CPU or architecture with the `--cpu` option that at least conforms to *ARM Architecture v5TE* (ARMv5TE), for ease of interoperation with the GNU tools.

### 3.13.1 See also

**Reference**

*Assembler Reference*:

- *--apcs=qualifier...qualifier* on page 2-7
- *--cpu=name* on page 2-19
- *--no_hide_all* on page 2-60.

## 3.14 Additional headers from the ARM Compiler toolchain

Some of the standard ARM Compiler toolchain headers must be used when building for ARM Linux. These headers define some implementation-specific macros that are dependent on the compiler rather than the C library used. The files are provided in the `arm_linux` and `arm_linux_compat` subdirectories of the ARM Compiler toolchain header files. The `arm_linux_compat` directory must be given before other system include directories in the include path list, and the `arm_linux` directory must appear at the end of the include path list. If the `ARMCC`*`n`*`INC` environment variable is set, then these paths are automatically used by the `--arm_linux` and `--arm_linux_paths` options and by GCC emulation mode.

An additional header file, `linux_armcc.h`, is also provided. This defines a number of macros for compatibility with GCC and the Linux environment. This is automatically included (equivalent to using `--preinclude=linux_armcc.h`) when using `--arm_linux` or `--arm_linux_paths`. When using the ARM Compiler toolchain to emulate GCC, these macros are defined internally in the compiler to permit preprocessing of files other than C or C++ source without automatically including the file.

If you want to use the DSP or NEON® intrinsics available in the ARM Compiler toolchain, these are also provided in the `arm_linux` subdirectory for convenience, for example `#include <arm_neon.h>`. However, both `dspfns.h` and `math.h` include a definition `round()`. You must rename one definition if you want to use both versions of these functions. For example:

```
#define round dsp_round#include <dspfns.h>#undef round
```

### 3.14.1 See also

**Reference**

*Introducing ARM® Compiler toolchain*:

- *Toolchain environment variables* on page 2-14

*Compiler Reference*:

- *--arm_linux* on page 3-16
- *--arm_linux_paths* on page 3-21
- *--preinclude=filename* on page 3-172.

## 3.15 Building a shared library with the ARM Compiler toolchain

When building dynamic shared libraries, all of the library code must be compiled and linked to be position-independent. To do this when using `armcc` options, use the `--apcs=/fpic` command-line option. In GCC emulation mode, use `-shared -fPIC`.

`armlink` supports the creation of dynamic shared libraries. This requires some additional options.

`--shared` Instructs the linker to create a dynamic shared library and not a static library.

`--soname <name>`
Specifies the shared object name (`SONAME`) for the library.

`--fpic` Enables you to link position-independent code that is compiled with `--apcs/fpic`.

For example, to link `libfunc.o` and `asmfunc.o` into a dynamic shared library `libdynamic.so`, you can use the following linker command line:

```
armlink --arm_linux --fpic --shared --soname libdynamic.so -o libdynamic.so libfunc.o
asmfunc.o libc.so.6
```

When using GCC emulation mode, if `-shared` is passed to the compiler driver this automatically passes `--shared --fpic` to `armlink`. You can still specify the shared object name or other options, for example using `-Wl`, `-soname`, `libexample.so`.

### 3.15.1 See also

**Reference**

*Compiler Reference*:

- *--apcs=qualifer...qualifier* on page 3-11.

*Linker Reference*:

- *--arm_linux* on page 2-13
- *--fpic* on page 2-74
- *--shared* on page 2-146
- *--soname=name* on page 2-151.

## 3.16 Using shared libraries in your application

You use shared libraries with `armlink` in the same way as normal libraries by specifying them on the linker command line. References to the shared library are added to the image and resolved to the library by the dynamic loader at runtime.

References to libraries are resolved in the order they are specified on the command line. This is also the order that the dependencies are resolved by the dynamic linker. You can specify the runtime location of libraries using the `--rpath` GNU ld option, that is an accepted alias of the `--runpath` linker option.

Unlike GNU ld, `armlink` repeatedly searches libraries in command-line order until either all references are resolved or no further references can be resolved by the given libraries. That is, `armlink` behaves similarly to:

```
ld --start-group lib1.a lib2.a lib3.a … --end-group
```

`armlink` supports a `--library=name` option similar to the `-l option` in GNU ld. This can search for libraries named as `libname.so` or `libname.a` depending on whether dynamic library searching is enabled at that point on the command line. The searching of dynamic libraries is controlled by the `--[no_]search_dynamic_libraries` option, as shown in the last two lines of the following example:

```
gcc -shared -fPIC -Wl,-Bdynamic -lfoo -Wl,-Bstatic -lbar

armcc --arm_linux -L--shared -L--fpic \-L--search_dynamic_libraries -L--library=foo
\-L--no_search_dynamic_libraries -L--library=bar
```

These two command lines perform a link searching for `libfoo.so` before `libfoo.a`, but only searching for `libbar.a`.

### 3.16.1 See also

**Reference**

*Linker Reference*:

- *--fpic* on page 2-74
- *--library=name* on page 2-97
- *--search_dynamic_libraries, --no_search_dynamic_libraries* on page 2-144
- *--runpath=pathlist* on page 2-138
- *--shared* on page 2-146.

*Compiler Reference*:

- *--arm_linux* on page 3-16
- *-Lopt* on page 3-127.

# Chapter 4
# Frequently-asked questions and troubleshooting

The following topics provide answers to common questions and additional information on building Linux applications with the ARM Compiler toolchain and GNU libraries:

**General information**



**Interoperation errors**



**Diagnosing common problems**

## 4.1 Where can I find more information on building Linux applications?

The recommended starting point for getting more information is the CodeSourcery toolchain FAQ.

You can also look at ARM and Linux forums and newsgroups, or at mailing list archives. The ARM Linux Project and the ARM Linux Wiki provide resources relating specifically to ARM Embedded Linux.

**Note**

ARM does not provide support on the use of the GNU tools. For more information, see the GNU Compiler Collection.

### 4.1.1 See also

**Other information**

- GNU ARM toolchain, http://www.gnuarm.com
- ARM Linux Internet Platform, http://linux.onarm.com
- Linux on ARM Wiki, http://www.linux-arm.org
- GNU Compiler Collection, http://gcc.gnu.org.

## 4.2 How do I build an EABI-compliant Linux kernel?

Prior to kernel version 2.6.16 an *Embedded Application Binary Interface* (EABI) compliant kernel could not be built. However, this is only an issue for applications and libraries that directly access kernel structures or functions. This is because the EABI-compliant GNU C library translates calls appropriately from EABI-compliant applications to the non EABI-compliant kernel system calls.

From kernel version 2.6.16, you can build an EABI kernel. However, you must still use the GNU toolchain.

### 4.2.1 See also

**Other information**

- GNU ARM toolchain, http://www.gnuarm.com
- ARM Linux Internet Platform, http://linux.onarm.com
- Linux on ARM Wiki, http://www.linux-arm.org
- GNU Compiler Collection, http://gcc.gnu.org.

## 4.3 Can I build the Linux kernel using the ARM Compiler toolchain?

The Linux kernel has a large amount of assembly code that is written in *GNU assembler* (GAS) syntax. The ARM assembler does not support the GAS syntax and therefore cannot be used to build the Linux kernel.

Because the most critical parts of the kernel are written in assembly and not C, you are unlikely to see a significant improvement if you use ARM Compiler toolchain to build the kernel.

### 4.3.1 See also

**Other information**

- GNU ARM toolchain, http://www.gnuarm.com
- ARM Linux Internet Platform, http://linux.onarm.com
- Linux on ARM Wiki, http://www.linux-arm.org
- GNU Compiler Collection, http://gcc.gnu.org.

## 4.4 Which kernel version must I use?

The binary form of the CodeSourcery toolchain is built to use *Native POSIX Thread Library* (NPTL) and it expects to have *thread-local storage* (TLS) support in the kernel. Recent CodeSourcery binary releases have a dependency on kernel version 2.6.16 or later because of the requirement for EABI support in the kernel.

**Note**

Your Linux distributor might have already applied the appropriate patches to their kernel build. Contact your Linux distributor for more information.

### 4.4.1 See also

**Other information**

- GNU ARM toolchain, http://www.gnuarm.com.

## 4.5 Can I use EABI-compliant and non EABI-compliant applications together?

To use EABI-compliant and non EABI-compliant applications together, you must:

1. Place the libraries and the dynamic linker in a different directory to the normal libraries.
2. Use `/libeabi` for the *Embedded Application Binary Interface* (EABI) compliant libraries, and leave the original, non EABI-compliant libraries in `/lib`.
3. Set the library search path for EABI applications using the environment variable `LD_LIBRARY_PATH=/libeabi` or by using the `--rpath` linker option.
4. Rebuild all applications to use the EABI in your final system because the extra libraries take up a significant amount of space in the filesystem.

### 4.5.1 See also

**Other information**

- ARM GNU/Linux ABI Supplement, http://www.codesourcery.com/gnu_toolchains/arm/arm_gnu_linux_abi.pdf

## 4.6 GNU tools report EABI version differences between source object and target

The GNU tools report the following error when used on objects generated by the ARM Compiler toolchain:

```
ERROR: Source object … has EABI version 5, but target … has EABI version 4
```

The ARM Compiler toolchain generates ELF files conforming to revision 5 of the ARM ABI ELF (AAELF) specification. However, the CodeSourcery 2005-q3-2 release only supports revision 4 of the AAELF specification, and does not consume objects produced by the ARM Compiler toolchain. Support for the new ABI revision is included in the 2006-q1-3 and later releases of the CodeSourcery toolchain.

### 4.6.1 See also

**Other information**

- ARM GNU/Linux ABI Supplement, http://www.codesourcery.com/gnu_toolchains/arm/arm_gnu_linux_abi.pdf

## 4.7 GNU linker or armlink report conflicts between `wchar_t` types

You see the following errors:

- GNU linker error:

  ```
  ld: ERROR: … : Conflicting definitions of wchar_t
  ```

- `armlink` error:

  ```
  Error: L6242E: Cannot link object dummy.o as its attributes are incompatible with
  the image attributes ... wchart-16 clashes with wchart-32.
  ```

This is because the linker has detected a mismatch between the `wchar_t` types used. The CodeSourcery document for building Linux applications specifies that `wchar_t` must be 32 bits.

A similar error exists for incompatible sizes of enumeration types. For ARM Linux, an **`enum`** must be 32 bits wide.

To resolve these errors, ensure that all of your code is compiled for 32-bit `wchar_t` and 32-bit enums, for example using the `--wchar32` and `--enum_is_int` `armcc` options. This is done automatically if `--arm_linux` is used.

Alternatively, `armlink` supports the options `--no_strict_wchar_size` and `--no_strict_enum_size` that avoid these errors. Be aware, however, that binary compatibility might be broken between the objects with differing attributes if they pass data of enum or `wchar_t` types between each other and this might lead to runtime failures.

### 4.7.1 See also

**Reference**

*Linker Reference*:

- *--arm_linux* on page 2-13
- *--strict_enum_size, --no_strict_enum_size* on page 2-157
- *--strict_wchar_size, --no_strict_wchar_size* on page 2-163.

*Compiler Reference*:

- *--enum_is_int* on page 3-85
- *--wchar32* on page 3-225.

**Other information**

- ARM GNU/Linux ABI Supplement, `http://www.codesourcery.com/gnu_toolchains/arm/arm_gnu_linux_abi.pdf`

## 4.8 Using hardware VFP instructions

ARM Linux uses software floating-point linkage, where floating-point arguments are passed in integer registers even if functions themselves perform operations in hardware VFP registers.

When building ARM Linux applications with the `--arm_linux` or `--arm_linux_paths` compiler command-line options, the default is always software floating-point linkage even if you specify a CPU that implies an FPU.

For example, if you specify a `--cpu` that implies an FPU, such as ARM1136JF-S or Cortex-A9, the compiler defaults to `--fpu=softvfp+vfp` rather than `--fpu=vfp`. `--fpu=softvfp+vfp` is equivalent to the GCC `-mfloat-abi=softfp` command-line option.

If you specify a `--cpu` that does not imply an FPU, you must explicitly specify `--fpu=softvfp+vfp` to use VFP.

You can override these explicitly to use hardware or software floating point variants of the Procedure Call Standard by specifying `--apcs=/hardfp` or `--apcs=/softfp` respectively. If using GCC emulation, the corresponding options are:

- `-mfloat-abi=hard`, to compile for hardware FPU with hardware linkage
- `-mfloat-abi=softfp`, to compile for hardware FPU but with software linkage
- `-mfloat-abi=soft`, to compile without hardware FPU instructions being used.

### 4.8.1 See also

**Reference**

*Compiler Reference*:

- *--apcs=qualifer...qualifier* on page 3-11
- *--arm_linux* on page 3-16
- *--arm_linux_paths* on page 3-21
- *--fpu=name* on page 3-100.

*Assembler Reference*:

- *--apcs=qualifier...qualifier* on page 2-7
  *--fpu=name* on page 2-39.

## 4.9 Can I use the ARM libraries in a Linux application?

In general, you must not use the ARM libraries when building a Linux application. The libraries provided with the ARM Compiler toolchain are targeted at standalone applications running directly on the target hardware, that is, without an OS. They contain semihosting calls and memory handling that is not suitable for use under an operating system like Linux. It is sometimes possible to use small, self-contained portions of the ARM library code, however you must take care to retarget any semihosted I/O functions and signal handling. Also, the ARM libraries can only be statically linked into an application or shared library.

### 4.9.1 See also

**Tasks**

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

- *Using the libraries in a nonsemihosting environment on page 2-36*.

**Concepts**

- *How can I see what libraries are being used?* on page 4-11
- *What to do about segmentation faults* on page 4-14.

*Developing Software for ARM® Processors*:

- Chapter 3 *Embedded Software Development*.

## 4.10 How can I see what libraries are being used?

The linker provides an option `--info=libraries` that lists the libraries it uses. For information on which library functions are being used, you can request verbose output from the linker with `--verbose` and redirect this to a file with `--list=filename.txt`.

When using `--arm_linux_paths` or the GCC emulation mode, the configuration file provides the list of system paths and standard libraries to link to. This file is in XML format, and you can examine this file in a text editor to check the libraries that are used by the tools.

### 4.10.1 See also

**Reference**

*Linker Reference*:

- *--info=topic[,topic,...]* on page 2-80
- *--list=filename* on page 2-102
- *--verbose* on page 2-184.

*Compiler Reference*:

- *--arm_linux_paths* on page 3-21.

## 4.11 How can I have greater control over which libraries are linked into my application?

If you require explicit control over the libraries that are linked with your application, this can be done with a manual link step by passing the `--arm_linux` linker option. The `--arm_linux` option sets the `--no_scanlib` option to disable searching of system library paths. You can then provide your own list of search paths with `--userlibpath`, and a list of libraries to use.

### 4.11.1 See also

**Reference**

*Linker Reference*:

- *--arm_linux* on page 2-13
- *--scanlib, --no_scanlib* on page 2-141
- *--userlibpath=pathlist* on page 2-179.

## 4.12 Common problems with running an application

The following table lists some common problems with running an application.

**Table 4-1 Common problems with running applications**

| Problem | Solution |
|---|---|
| Cannot find the application | • Check that the application is on the path, or you are running it with `./program` in the current directory.<br>• The dynamic loader may not be the same as specified at link time. In this case, use */path-to-linker/dynamic-loader program-path/program*. For example:<br>`/libeabi/ld-linux.so.3 /opt/bin/eabi/hello`<br>**Note**<br>You can specify an alternative dynamic loader for an application by passing, for example, `--dynamiclinker=/libeabi/ld-linux.so.3` |
| Permission denied | Check that you have set the executable flag for the program (use `chmod +x program`). |
| "GLIBC_2.4 not found" error<br>"unable to find library XXX.so.X" | This is the dynamic linker reporting that it cannot use the libraries found on its default path. You can use the `LD_LIBRARY_PATH` environment variable to access the correct libraries. For example:<br>`LD_LIBRARY_PATH=/libeabi ./helloworld`<br>Alternatively, you can use the `--rpath` linker option. |
| "Illegal instruction" error before `main()` | This indicates that the image has been built for the incorrect architecture (for example, ARMv6 code running on an ARMv5TE core), or that the kernel has been built without NPTL support.<br>Check that you have built the image for the correct ARM architecture and check that you are using either a 2.6.12 (or later) Linux kernel or one with the appropriate patches applied as part of your distribution.<br>Also ensure that the system call interface matches between the Linux kernel and the GNU C library you are using.<br>**Note**<br>The binary libraries from recent CodeSourcery releases are built for the new system call interface introduced with EABI support in kernel version 2.6.16.<br>When at `main()` this is likely to be an actual undefined instruction in the application. |

### 4.12.1 See also

**Reference**

*Linker Reference*:

- *--dynamic_linker=name* on page 2-51
- *--runpath=pathlist* on page 2-138.

**Other information**

- GNU ARM toolchain, `http://www.gnuarm.com`.

.

## 4.13 What to do about segmentation faults

There are many possible causes of segmentation faults. They might be caused by problems with your application. You must also ensure that:

- When you use a manual link step, pass the `--no_scanlib` or `--arm_linux` switches to the linker. This ensures that the linker does not search the ARM libraries and accidentally link in semihosted I/O functions. If you are explicitly linking with any portions of the ARM libraries, ensure that any semihosted I/O and signal handling functions are retargeted appropriately.
- When you create a dynamic library, compile and link as position-independent code (use `--apcs/fpic` for the compiler and `--fpic` for the linker).
- When you create an application using a manual link, use either the two linker switches `--no_startup` and `--entry _start`, or the linker switch `--arm_linux`.
- When you use C++ exceptions, link with libraries from an appropriate CodeSourcery release (2007-q1-10 or later) and use these libraries on your target filesystem.

### 4.13.1 See also

**Concepts**

- *Can I use the ARM libraries in a Linux application?* on page 4-10.

**Reference**

*Compiler Reference*:

- *--apcs=qualifer...qualifier* on page 3-11.

*Linker Reference*:

- *--arm_linux* on page 2-13
- *--entry=location* on page 2-58
- *--fpic* on page 2-74
- *--scanlib, --no_scanlib* on page 2-141
- *--startup=symbol, --no_startup* on page 2-155.

## 4.14 Image sizes and stripping debug data

Both the GNU and ARM toolchains add a significant amount of information to an image that is generally only of use for debugging.

For production systems, you can strip the debugging data from your applications and shared libraries. With the ARM Compiler toolchain, use the `--no_debug` switch at the link stage or run `fromelf` on the linked image. In addition, you can use `fromelf` to remove the `.comment` sections and symbols from the file. For example:

```
fromelf --strip debug,comment,symbols --elf -output stripped.axf image.axf
```

In addition, the data sizes in images built with the ARM Compiler toolchain can be slightly larger than those in GNU images. This is typically because some ZI data (BSS) is moved into the RW data area for performance reasons on bare-metal systems. You can move this data to ZI sections using the compiler switch `--bss_threshold=0`. This is enabled by default when using GCC emulation.

### 4.14.1 See also

**Reference**

*Using the fromelf Image Converter*:

- *--elf* on page 4-29
- *--output=destination* on page 4-57
- *--strip=option[,option,...]* on page 4-70.

*Linker Reference*:

- *--debug, --no_debug* on page 2-41.

*Compiler Reference*:

- *--bss_threshold=num* on page 3-30.

## 4.15 Undefined symbol errors for pthread symbols

You get undefined symbol errors for pthread symbols despite using `--arm_linux_paths` and `--arm_linux_config_file` when building ARM Linux applications using normal compiler options:

- If you use GCC translation, the compiler supports the `-pthread` and `-lpthread` options. The compiler adds the correct combination of libraries to the linker command line. For example:

  ```
  armcc --translate_gcc --arm_linux_config_file=myconfigfile -o test test.c -pthread
  ```

  This does a compile and link of `test.c`, linking with the pthread library.

- If you use `--arm_linux_paths` rather than GCC command-line translation, you must manually add the pthread libraries to your command-line as appropriate. The pthread libraries are intended to appear first on the command-line before any other libraries, so that they can override other C library symbols. In particular, if you have a legacy build that specifies the libc libraries explicitly, the pthread variants of those functions are not found first by the linker. To link with the pthread libraries when using `--arm_linux_paths`, specify the libraries explicitly on the linker command line. For example:

  ```
  armcc --arm_linux -o test test.c -Llibpthread.so.0 -Llibpthread_nonshared.a.
  ```

### 4.15.1 See also

**Reference**

*Compiler Reference*:

- *--arm_linux* on page 3-16
- *--arm_linux_config_file=path* on page 3-18
- *--arm_linux_paths* on page 3-21
- *-Lopt* on page 3-127
- *-o filename* on page 3-154
- *--translate_gcc* on page 3-200.

# Appendix A
# Revisions for Building Linux Applications with the ARM Compiler toolchain and GNU Libraries

The following technical changes have been made to *Building Linux Applications with the ARM Compiler toolchain and GNU Libraries*:

**Table A-1 Differences between Issue C and Issue F**

| Change | Topics affected |
|---|---|
| Updated the list of environment variables to the new version numbering scheme, for example `ARMCC5INC`. | Various topics |

**Table A-2 Differences between Issue B and Issue C**

| Change | Topics affected |
|---|---|
| Removed the limitation *You must take care when using alloca()*. | *Limitations when building Linux applications* on page 2-4 |