Binary Interoperability Between Toolchains

Application Note 487

ARM®

DAI 0487A (ID060915)

Binary Interoperability Between Toolchains
Application Note 487

Release Information

Table 1 Change history

Date	Issue	Confidentiality	Change
June 2015	A	Non-Confidential	First release

Proprietary Notice

Confidentiality Status

Product Status

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

Web Address

`http://www.arm.com`

1 Introduction

This application note discusses binary interoperability between toolchains, and specifically focuses on bare-metal embedded targets. Bare-metal targets are platforms without an operating system. Bare-metal targets are commonly found in microcontroller platforms that are based around Cortex-M processors.

Binary interoperability is complex and technical. This application note is not a complete reference for interoperability. However, it covers:

- Important considerations.
- Main restrictions.
- Common obstacles to be careful of.

This application note makes references to the *ABI for the ARM 32-bit Architecture*. These documents are available at:
http://infocenter.arm.com/help/topic/com.arm.doc.subset.swdev.abi/index.html

This application note uses notations to refer to the following ABI documents:

- *CLIBABI* to refer to the *C Library ABI for the ARM Architecture*:
  http://infocenter.arm.com/help/topic/com.arm.doc.ihi0039-/index.html
- *RTABI* to refer to the *Run-time ABI for the ARM Architecture*:
  http://infocenter.arm.com/help/topic/com.arm.doc.ihi0043-/index.html

2 Binary interoperability overview

Many modern toolchains that target the ARM architecture support the ABI for the ARM Architecture as a standard for interoperability of binary object files between toolchains.

The ABI provides a standard for many aspects of interoperability. However, there are a number of other considerations that make interoperation between output from different toolchains difficult. The interoperability depends on the exact use case and choices made within each part of the different toolchains in question. This makes full binary interoperation extremely difficult to achieve. Toolchains always have significant freedom to make alternative choices, usually in the interest of performance or code size over portability. Also, there are often instances where toolchains do not strictly conform to the ABI, or omit certain aspects of the ABI support. In general, assessing how to maximize interoperability between the output from multiple toolchains requires reviewing the documentation for the toolchains to understand:

- What portability support is available.
- Which options must be selected for maximum ABI conformance and best portability.

Where possible, providing portable source code that can be rebuilt for a given target with a different toolchain is always the easiest approach for guaranteed portability between toolchains, and is preferred over attempting to guarantee binary interoperability.

This application note assumes certain interoperability restrictions:

- *CLIBABI section 3.2, A C library is all or nothing*, specifies that the static linker and the C library must be from the same toolchain. There necessarily must be the potential for collusion between the C library and linker for a given platform. As such, the linker and library must be from the same toolchain. Any given linker or C library might attempt to provide interoperability with alternative linkers and libraries, however the level of compatibility will be entirely dependent on their implementation.
- The focus is on interoperability for bare-metal embedded targets, which are embedded systems without any OS. If an OS is present, then generally the preferred toolchain is the one recommended for the OS. Any interoperability of other toolchains targeting that OS is entirely at the discretion of the toolchain supplier and OS supplier.
- For simplicity, this application note assumes a single, statically-linked binary. Any use of position-independent or otherwise relocatable code is dependent on the exact mechanisms supported by the toolchains in use and loading mechanism used on the target.

This section contains:

2.1 ABI considerations

The ABI for the ARM Architecture provides a common base standard for many aspects of binary interoperability. However, it provides a number of variants, especially with respect to the *Procedure Call Standard* (PCS), that must be considered when deciding how to build code for interoperation between the output from different toolchains.

All of these variants are intended as options for the target platform as a whole. In an embedded environment this implies that the selections are applicable to the entire static link of the final image, including all object files that can come from different toolchains and must interoperate in the final executable. If multiple variants of the PCS must be supported at final link time, appropriate variants of all linked components must be available.

Note

There might be ways of combining strictly incompatible object files by, for example, ensuring that the functions at the public interfaces are carefully controlled and compatible. However, the disadvantages of this method are:

- The interfaces might be difficult to maintain cleanly.
- The level of support is entirely at the discretion of the toolchains in question.

This method is therefore outside the scope of this application note.

Floating-point argument passing

The *Procedure Call Standard* (PCS) provides variants for passing floating-point arguments between functions in either integer registers or in floating-point registers. Code that passes arguments in floating-point registers will only be compatible with other code that expects arguments to be passed in floating-point registers. Also, such code can only run on targets with the appropriate floating-point hardware support.

Where code is intended to interoperate with other code that might be linked for a target without hardware floating-point support or that might be compiled for the base PCS, the code must either be compiled for the base PCS or alternative builds of the code must be provided for the two PCS variants.

Size of C standard types: `wchar_t` and `enum`

The PCS also provides options for the size of the C standard types:

- `wchar_t`, which can be either 2 bytes or 4 bytes.
- `enum`, which can be either the width of `int`, or the smallest fundamental type able to contain all members of the enum.

These options apply to the target platform as a whole. All objects in the final link must agree on the variant in use. Therefore, if an output from a toolchain is intended for linking with code compiled for different variants, then the interoperable components might need separate builds of different variants, corresponding to each of the PCS options.

2.2 Compiler assumptions about libraries

Many toolchains have a default configuration that includes some collusion of the compiler with the libraries from the same toolchain. Often this is to:

- Improve the speed of the generated code.
- Reduce the generated code size.
- Avoid any cost of portability where it is not needed.

For example, compilers might recognize calls to certain standard C library functions and replace them with calls to more optimal, non-standard variants in specific cases. Generally a compiler will have an option to disable such library collusion at compile time.

Some compilers will also have a requirement for some functionality to be provided by a runtime support library, sometimes called a *helper library*, which is provided as part of the toolchain. These libraries contain private helper functions that are used where it is not feasible or optimal

to generate code for certain source constructs directly in the generated code. It might be possible to include this with any interoperable code. However, this might have technical limitations, for example:

- The final linker might not be able to correctly consume the helper library from another toolchain.
- Appropriate variants of the helper library might not be available.

There might be licensing restrictions on redistribution of the helper library. ARM recommends that you consult the supplier of the toolchains in question for details of their requirements in this area.

2.3 Library standardization across header files

For a typical C interoperability situation, it is expected to want to include the C library headers of one toolchain, and then link with the binary C libraries of the toolchain used for the final link. This implies that everything provided by the C headers of the first toolchain must be made binary-portable and compatible with the end libraries of the second toolchain.

The ABI provides mechanisms for such standardization. As a minimum, when including a standard C header file, application code might be able to request and confirm portability using a guarding mechanism such as the following:

```
#define _AEABI_PORTABILITY_LEVEL 1
#undef _AEABI_PORTABLE
#include <header.h>
#ifndef _AEABI_PORTABLE
#    error "AEABI not supported by header.h"
#endif
```

See *CLIBABI section 5.1.1*, *Detecting whether a header file honors an AEABI portability request*, for more information about this example.

However, this does not completely guarantee the suitability of the resulting code for interoperability. For example, a header from a given toolchain might be interoperable even if it has chosen not to advertise itself in the manner compatible with the ABI mechanism described above. In such cases, the above example would unnecessarily report errors. Conversely, a library's header set might make additional decisions, for example on the basis of the compiler's optimization settings and associated predefined macros, with implications for portability. Some of these effects and difficulties might be subtle. For example, there might be instances where compiler-specific headers, such as `stdint.h` and `stddef.h`, do not strictly conform to the ABI. You must take care to ensure that this does not introduce incompatibilities when linking objects compiled with different toolchains. Ultimately, if the above portability mechanism is not sufficient, it might be necessary to check with the toolchain supplier for the best approach.

2.4 Interoperability of C++

C++ adds significantly more complexity to an interoperability scenario than C. Toolchains might differ substantially in the level of conformance they provide, and might make different implementation decisions about certain low-level details that make interoperability difficult or impossible. Even in the absence of different toolchains, it is not uncommon to find difficulties with interoperability between binaries from different versions of a single toolchain. The reasons for this include:

- The language evolves.
- The compilers and libraries adopt different implementations decisions.

Overall this makes full interoperability for C++ exceptionally difficult in the general case. If you must provide interoperability for C++ code, ARM recommends that the code provides an `extern "C"` public interface to the interoperable components. Use of the C++ *Standard Template Library* (STL) is difficult and would require inclusion and final linking with the same STL as provided with the toolchain. This makes it unsuitable in practice for many typical interoperation scenarios. It is possible that simple C++ code can interoperate between toolchains successfully. However, there is no guarantee of this in general and the difficulty increases as more language features are used.

3 Recommended build options for ARM toolchains

This section contains build option recommendations for generating interoperable code, when using ARM toolchains. It contains:

- *General recommendations*
- *ARM Compiler 5 build options*
- *ARM Compiler 6 build options* on page 9
- *GNU toolchain (GCC, Binutils and Newlib) build options* on page 9
- *Summary of PCS variant options per toolchain* on page 10.

3.1 General recommendations

The linkers in both ARM Compiler and GNU toolchains produce errors or warnings when they encounter objects built for different PCS variants, for example with different sizes of `enum` types or `wchar_t` types. In the case of `armlink`, you can use `--diag_suppress=6242` to suppress the error. However, in both cases you must be careful to avoid the use of conflicting types across the interface between the relevant objects. In general, ARM recommends building appropriate variants so that a single PCS variant can be used at link time.

In general, in cases where object code from ARM Compiler and the GNU tools are being combined, it might be easier to use `armlink` as the final linker. This can provide more control over which incompatibilities are ignored. However, for best practice, ARM still recommends building appropriate object variants for your use case to avoid introducing incorrect behavior.

3.2 ARM Compiler 5 build options

By default, `armcc` enables certain optimizations that assume the presence of the ARM Compiler libraries. These can be disabled using `--library_interface=aeabi_clib`. When this is disabled, the library headers are portable in accordance to the ABI definition, provided that the guarding mechanism, described in *Library standardization across header files* on page 6, is used.

You can select the PCS variants using:

- `--apcs=/softfp` or `--apcs=/hardfp` to indicate how floating-point arguments are passed.
- `--wchar16` or `--wchar32` to indicate the size of the wchar_t type.
- `--[no_]enum_is_int` to indicate the size of enum types.

Note that there are certain other options that automatically imply different defaults. In particular, the `--cpu` and `--fpu` setting can imply different defaults for the floating-point PCS variant, which can be `softfp` or `hardfp`. ARM recommends that you specify the above options explicitly to select the PCS variant appropriate for your needs.

ARM also recommends that all source files are compiled or assembled with the `--apcs=/interwork` option. The ABI relies on a base architecture version of ARMv5TE or above, with full support for ARM/Thumb interworking. As a safety precaution, unless this option is provided, the build attributes for the generated object files are marked as not fully conformant to the ABI.

Similarly, ARM recommends avoiding the use of the `ENTRY` directive in `armasm` assembly source files as this requires a private ELF flag that other linkers might not support. ARM recommends generating debug information as DWARF version 3. The ABI mandates this level of support, and the use of DWARF version 2 might be incompatible with other linkers. DWARF version 3 is the default for `armcc` and `armasm`.

In ARM Compiler 5.05 and its update releases, armcc generates all runtime helper functions in the object files that it generates, and does not require helper libraries to be linked with. Certain older versions of ARM Compiler did not do this, and helper libraries are provided in ARM Compiler 5 purely to support linking with these older toolchains.

3.3 ARM Compiler 6 build options

In ARM Compiler 6, armclang does not currently apply the library collusion optimizations that ARM Compiler 5 uses. The library headers themselves are similar to ARM Compiler 5, and therefore are also portable in accordance to the ABI definition, providing that the guarding mechanism, described in *Library standardization across header files* on page 6, is used.

You can select the PCS variants using the following options to armclang:

- -mfloat-abi=soft, -mfloat-abi=softfp or -mfloat-abi=hard to indicate how floating-point arguments are passed, and whether hardware floating-point instructions are used.
- -fshort-wchar or -fno-short-wchar to indicate the size of the wchar_t type.
- -fshort-enums or -fno-short-enums to indicate the size of enum types.

ARM recommends that you explicitly specify the PCS variant options that are appropriate for your needs.

When generating code for the hardware floating-point PCS variant, using -mfloat-abi=hard, armclang uses alternative symbol naming to select hardware floating-point functions from relevant portions of the C library. These functions are declared in math.h and in stdlib.h. For example, such code will call __hardfp_sin as opposed to sin. This follows the convention documented in *RTABI section 3.10, __hardfp_ name mangling*. Where such code is being linked with another C library and these functions are not provided with the alternative symbol names, it might be necessary to instruct the linker to rename these calls to the standard symbols or provide stub implementations of the functions decorated with __hardfp, which call the normally-named functions.

ARM recommends avoiding the use of the ENTRY directive in armasm assembly source files for the reasons described in *ARM Compiler 5 build options* on page 8.

The helper libraries needed for compatibility with older versions of ARM Compiler 5 are also provided in ARM Compiler 6 to support linking with these older toolchains. They are not needed to link code generated by armclang.

3.4 GNU toolchain (GCC, Binutils and Newlib) build options

You can select the PCS variants using the following options to gcc:

- -mfloat-abi=soft, -mfloat-abi=softfp or -mfloat-abi=hard to indicate how floating-point arguments are passed, and whether hardware floating-point instructions are used.
- -fshort-wchar or -fno-short-wchar to indicate the size of the wchar_t type.
- -fshort-enums or -fno-short-enums to indicate the size of enum types.

ARM recommends that you explicitly specify the PCS variant options that are appropriate for your needs.

Newlib does not apply any compiler or linker collusion that ARM is aware of. However, there might be some helper functions defined in the *RTABI* document that are not provided by newlib. When linking code generated by an alternative toolchain that expects these functions to be present, you might need to provide your own implementations of any missing functions. Definitions of the functions and their semantics are listed in the *RTABI* document.

3.5 Summary of PCS variant options per toolchain

The following table summarizes the options needed for each toolchain to select particular aspects of the PCS variant.

Table 2

	ARM Compiler 5 (armcc)	**ARM Compiler 6 (armclang)**	**GNU toolchain (gcc)**

Software floating-point PCS	`--apcs=/softfp`	`-mfloat-abi=soft`
		`-mfloat-abi=softfp`
		(see note below)
			`-mfloat-abi=soft`
			`-mfloat-abi=softfp`
			(see note below)

Hardware floating-point PCS	`--apcs=/hardfp`	`-mfloat-abi=hard`	`-mfloat-abi=hard`
Short (smallest-fit) `enum` types	`--no_enum_is_int`	`-fshort-enums`	`-fshort-enums`
`enum` types at least `int` width	`--enum_is_int`	`-fno-short-enums`	`-fno-short-enums`
2-byte `wchar_t`	`--wchar16`	`-fshort-wchar`	`-fshort-wchar`
4-byte `wchar_t`	`--wchar32`	`-fno-short-wchar`	`-fno-short-wchar`
Defaults	• Floating-point PCS depends on CPU or FPU selection. • `--no_enum_is_int` • `--wchar16`	• `-mfloat-abi=soft` • `-fno-short-enums` • `-fno-short-wchar`	• `-mfloat-abi=soft` • `-fno-short-enums` • `-fno-short-wchar`

Note

For ARM Compiler 6 and the GNU toolchain:

- `-mfloat-abi=soft` selects the floating-point PCS variant with no use of hardware floating-point instructions in the generated code.
- `-mfloat-abi=softfp` selects the floating-point PCS variant but uses floating-point hardware instructions within the functions in the generated code. The resulting code can only run on targets that support hardware floating-point instructions.