ARM® Compiler

Version 5.06

Migration and Compatibility Guide

ARM

ARM DUI0530M

ARM® Compiler

Migration and Compatibility Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
A	28 May 2010	Non-Confidential	ARM Compiler v4.1 Release
B	30 September 2010	Non-Confidential	Update 1 for ARM Compiler v4.1
C	28 January 2011	Non-Confidential	Update 2 for ARM Compiler v4.1 Patch 3
D	30 April 2011	Non-Confidential	ARM Compiler v5.0 Release
E	29 July 2011	Non-Confidential	Update 1 for ARM Compiler v5.0
F	30 September 2011	Non-Confidential	ARM Compiler v5.01 Release
G	29 February 2012	Non-Confidential	Document update 1 for ARM Compiler v5.01 Release
H	27 July 2012	Non-Confidential	ARM Compiler v5.02 Release
I	31 January 2013	Non-Confidential	ARM Compiler v5.03 Release
J	27 November 2013	Non-Confidential	ARM Compiler v5.04 Release
K	10 September 2014	Non-Confidential	ARM Compiler v5.05 Release
L	29 July 2015	Non-Confidential	ARM Compiler v5.06 Release
M	11 November 2016	Non-Confidential	Update 3 for ARM Compiler v5.06 Release

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

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 Migration and Compatibility Guide

List of Tables
ARM® Compiler Migration and Compatibility Guide

Preface

This preface introduces the *ARM® Compiler Migration and Compatibility Guide*.

It contains the following:

- *About this book* on page 9.

About this book

ARM® Compiler Migration and Compatibility Guide provides migration and compatibility information between the latest released version and previous versions.

Using this book

This book is organized into the following chapters:

Appendix A Migration and Compatibility document revisions
Describes the technical changes that have been made to the Migration and Compatibility Guide.

Glossary

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

See the *ARM Glossary* for more information.

Typographic conventions

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

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

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

`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 *ARM® Compiler Migration and Compatibility Guide*.
- The number ARM DUI0530M.
- If applicable, 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.

Other information

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

Chapter 1
Overview of Compatibility

Describes the compatibility between different versions of ARM Compiler.

It contains the following sections:

1.1 Compatibility between ARM Compiler versions

Although compatibility between different versions of ARM Compiler cannot be guaranteed, there are ways that you can aid compatibility.

ARM Compiler generated code conforms to the ARM *Application Binary Interface* (ABI). Also:

- For C code, ARM expects full backwards compatibility with earlier versions, except as described in the specific sections relating to a given version.
- For C++ code, to guarantee binary compatibility, including backwards compatibility, ARM recommends that you define your interfaces as `extern "C"`. Although many objects and libraries are binary compatible between versions and toolchains, ARM cannot guarantee that there are no differences in some cases. For example, mangled names might be different for symbols.

Related information

ARM Application Binary Interface.

Chapter 2
Configuration Information for Different Versions of the ARM Compilation Tools

Describes the FlexNet, GCC, and Cygwin versions supported by the different versions of the ARM compilation tools.

It contains the following sections:

2.1 FlexNet versions supported

Different versions of ARM® Compiler use different versions of FlexNet.

ARM recommends that you use the latest available version of the FlexNet license server software. You can obtain the latest license server from the ARM Self-Service downloads area on *silver.arm.com*. The following table shows the minimum license server software version required to use the tools.

Table 2-1 FlexNet versions

Compilation tools version	Windows	Linux
ARM Compiler 5.06	11.12.1.0	11.12.1.0
ARM Compiler 5.05	11.12.1.0	11.12.1.0
ARM Compiler 5.04	11.10.1.0	11.10.1.0
ARM Compiler 5.03	11.10.1.0	11.10.1.0
ARM Compiler 5.02	10.8.10.0	10.8.10.0
ARM Compiler 5.01	10.8.10.0	10.8.10.0
ARM Compiler 5.0	10.8.7.0	10.8.7.0
ARM Compiler 4.1	10.8.7.0	10.8.7.0
RVCT 4.0 build 471	10.8.7.0	10.8.7.0
RVCT 4.0	10.8.5.0	9.2
RVCT 3.1 build 836	10.8.7.0	10.8.7.0
RVCT 3.1 build 739	10.8.5.0	10.8.5.0
RVCT 3.1	10.8.5.0	9.2
RVCT 3.0	10.8.5.0	10.8.0
RVCT 2.2	9.0.0	9.0.0
RVCT 2.1	9.0.0	9.0.0
RVCT 2.0	8.1b	8.1b
ADS 1.2	7.2i	7.2i

Related information

ARM DS-5 License Management Guide.

2.2 GCC versions emulated

Different versions of ARM Compiler emulate different versions of GCC in GNU mode.

The GCC versions emulated by default in the compilation tools are:

Table 2-2 GCC versions

Compilation tools version	GCC version
ARM Compiler 5.06	4.7.0
ARM Compiler 5.05	4.2.0
ARM Compiler 5.04	4.2.0
ARM Compiler 5.03	4.2.0
ARM Compiler 5.02	4.2.0
ARM Compiler 5.01	4.2.0
ARM Compiler 5.0	4.2.0
ARM Compiler 4.1	4.2.0
RVCT 4.0	4.2.0
RVCT 3.1	3.3.0

Related information

--gnu (armcc).

--gnu_version=version (armcc).

2.3 Cygwin versions supported

ARM Compiler releases, including update releases, are validated against the Cygwin distribution available at the time of release.

See the *ARM® Compiler 5 Release Notes* for your specific release of the compiler for additional information.

Note

You can use ARM Compiler with Cygwin on the supported Windows platforms. However, Cygwin path translation enabled by `CYGPATH` is only supported on 32-bit Windows platforms.

Related information

About specifying Cygwin paths in compilation tools on Windows.

Chapter 3
Migrating from ARM Compiler v5.05 to v5.06

Describes the changes that affect migration and compatibility between ARM Compiler v5.05 and v5.06.

It contains the following sections:

3.1 Compatibility of ARM Compiler v5.06 with legacy objects and libraries

Backwards compatibility of objects and libraries built with RVCT 3.0 and earlier is supported provided they have not been built with `--apcs /adsabi`.

Given these restrictions, ARM strongly recommends that you rebuild your entire project, including any user, or third-party supplied libraries, with ARM Compiler v5.06. This is to avoid any potential incompatibilities, and to take full advantage of the improved optimization, enhancements, and new features provided by ARM Compiler v5.06.

Related concepts

1.1 Compatibility between ARM Compiler versions on page 1-13.

Related information

--apcs (armasm).

--apcs (armcc).

3.2 Compiler changes between ARM Compiler v5.05 and v5.06

Various changes have been made to `armcc` in ARM Compiler toolchain v5.06.

The following changes have been made to the compiler:

- Some older processors are deprecated. Specify `--cpu=list` to see the supported processors. The tools give a warning if you specify a deprecated processor.
- The behavior of `#pragma arm section` has changed with respect to inline functions.

This section contains the following subsections:

3.2.1 #pragma arm section and inline functions

ARM Compiler v5.05 ignored `#pragma arm section` for inline functions.

By default, ARM Compiler v5.06 respects this pragma for inline functions.

The `--no_ool_section_name` command-line option lets you revert to the ARM Compiler v5.05 behavior.

3.2.2 GCC default version changed

The default value of `--gnu_version` is now `40700`, that is GCC version 4.7.0.

There are some differences in behavior to be aware of:

- The accepted dialect of C/C++ has been changed in GCC. Therefore, existing code that compiles with the former default, `--gnu_version=40200` (GCC version 4.2.0), might not compile with `--gnu_version=40700`.
- In C++, a friend class declaration no longer considers typedefs. For example, the friend declaration refers to `A<T>` and not `A<int>`:

```
template< class T > class A { };
class B : public A<B> {
    typedef A<int> A;
    friend class A;     // finds injected class name of A
};
```

- In C++, it is no longer permissible to convert a pointer, or pointer to member conversion, to a protected base class:

```
struct B {};
struct D1: protected B { };
void f(B*);
struct D2: B {
    D1 d;
    D2() {
        f(&d);    // Not accepted with gnu_version >= 40400
    }
};
```

- In C++11, the type of an integer literal larger than can be represented as `long int`, has the type `long long int`, providing it can be represented by the type `long long int`, otherwise it has the type `unsigned long long int`. Previously in `--gnu` mode, integer literals that could be represented as an `unsigned long int` had the type `unsigned long int`.
- The macro `__cplusplus` is defined as 199711L for C++03, or 201103L for C++11. Previously in `--gnu` mode, if C++ or C++11 was used, it was defined to 1.
- In C++11, the compiler generates default Move and Move assignment operators, when required by the standard.
- In C++ and C++11, functions with identical parameter names are no longer accepted:

```
void f(int i, int i); // No longer accepted
void g(){
    f(0, 0);
}
```

3.2.3 __ldrex and __strex intrinsics deprecated

The `__ldrex` and `__strex` family of intrinsics have been deprecated, and use of the `LDREX` and `STREX` family of instructions in inline assembly code has also been deprecated.

Use one of the following alternatives:

- Write code that uses these instructions in an embedded assembler function.
- Use an appropriate member of the `__sync_*` family of GNU built-in functions, for example `__sync_lock_test_and_set()`.

For example, the following code uses the deprecated `__ldrex` and `__strex` intrinsics to implement a mutex lock, preventing concurrent processes from writing to output at the same time:

```
#define LOCKED 1
#define UNLOCKED 0

static int output_mutex = UNLOCKED;

void lock_mutex(void) {
  int status;
  do {
    status = __ldrex(&output_mutex);
  } while(status == LOCKED || __strex(LOCKED, &output_mutex) );
}

void unlock_mutex() {
  output_mutex = UNLOCKED;
}

void output_string(char * str)
{
  int i;
  /* Wait until the output mutex lock is acquired */
  lock_mutex();

  /* Output each character in turn */
  for(i=0 ; str[i] != '\0' ; i++) {
    putc(str[i], stdout);
  }

  /* Release output mutex lock */
  unlock_mutex();
}
```

The locking mechanism `lock_mutex()` can be refactored to use the `__sync_*` family of GNU built-in functions as follows:

```
void lock_mutex(void) {
  while (!__sync_bool_compare_and_swap(&output_mutex, UNLOCKED, LOCKED)) {};
}
```

Related information

GNU built-in functions.

Chapter 4
Migrating from ARM Compiler v5.04 to v5.05

Describes the changes that affect migration and compatibility between ARM Compiler v5.04 and v5.05.

It contains the following sections:

4.1 Compatibility of ARM Compiler v5.05 with legacy objects and libraries

Backwards compatibility of objects and libraries built with RVCT 3.0 and earlier is supported provided they have not been built with `--apcs /adsabi`.

Given these restrictions, ARM strongly recommends that you rebuild your entire project, including any user, or third-party supplied libraries, with ARM Compiler v5.05. This is to avoid any potential incompatibilities, and to take full advantage of the improved optimization, enhancements, and new features provided by ARM Compiler v5.05.

Related concepts

1.1 Compatibility between ARM Compiler versions on page 1-13.

Related information

--apcs (armasm).

--apcs (armcc).

4.2 Compatibility of code compiled with C++11 with code compiled with C++03

Appendix C of the C++11 standard describes incompatibilities between C++11 and C++03.

This section contains the following subsections:

4.2.1 Use of C++11 with the ARM C++ Standard Libraries

ARM Compiler provides the different types of C++ library.

The libraries provided are:

- The runtime library containing support for language features, such as exceptions.
- The Rogue Wave C++ standard library.

Changes in the standard libraries from ARM Compiler 5.05

The Rogue Wave C++ standard library implements the C++98 standard. There is no support for the C++11 library standard.

Using the runtime libraries from previous versions of ARM Compiler

Older releases of the ARM Compiler runtime library do not contain support for the new language runtime changes. You must compile your C++11 code with the `--cpp_compat` language option to prevent the compiler from using features that require updated runtime library support.

C++11 compatibility mode

A command line option `--cpp_compat` has been added to `armcc` and `armlink`.

Related information

Rogue Wave Standard C++ Library Documentation.

--cpp_compat (armcc).

--cpp_compat (armlink).

4.3 Compiler changes between ARM Compiler v5.04 and v5.05

Various changes have been made to `armcc` in ARM Compiler toolchain v5.05.

The following changes have been made to the compiler:

- Support for many C++11 language features.
- Support for a subset of C++11 that allows a runtime library from a previous version of the tools to be used.
- The compiler front-end has been updated. There are a small number of C++ name mangling differences.

This section contains the following subsections:

- *4.3.1 C++ name mangling differences in ARM Compiler v5.05* on page 4-25.

4.3.1 C++ name mangling differences in ARM Compiler v5.05

Various name mangling changes have been made in ARM Compiler v5.05.

Non-type non-dependent template arguments

When one template uses another template, the way that the non-type arguments to the used template that do not depend on the type parameters of the using template are mangled has changed.

The cases affected are:

sizeof

In ARM Compiler v5.04 expressions involving `sizeof(x)` had a mangling that included the `sizeof`. ARM Compiler v5.05 mangles the integer constant that the `sizeof()` expression evaluates to.

These changes in behavior might result in problems when all of the following conditions are true:

- A translation unit is compiled with ARM Compiler v5.05.
- A second translation unit is compiled with ARM Compiler v5.04 or earlier.
- The translation units instantiate or refer to a template that includes `sizeof(x)`.

__alignof

In ARM Compiler v5.04 in `--gnu` mode, expressions involving `__alignof__(x)` had a mangling that included the `__alignof__`. ARM Compiler v5.04 in `--gnu` mode mangles the integer constant that the `__alignof__(x)` expression evaluates to.

These changes in behavior might result in problems when all of the following conditions are true:

- A translation unit is compiled with ARM Compiler v5.05 using the `--gnu` option.
- A second translation unit is compiled with ARM Compiler v5.04 or earlier using the `--gnu` option.
- The translation units instantiate or refer to a template that includes `__alignof__(x)` expressions.

Binary operations

In ARM Compiler v5.04, binary operations were included in the mangling. ARM Compiler v5.05 in C++11 mode has the same behavior. ARM Compiler v5.05 in non-C++11 mode evaluates the integer constant and then mangles the result.

This change might result in problems when all of the following conditions are true:

- A translation unit is compiled with ARM Compiler v5.05 without the `--cpp11` option.
- A second translation unit is compiled with ARM Compiler v5.04 or earlier, or ARM Compiler v5.05 using the `--cpp11` option.
- The translation units instantiate or refer to a template that includes binary operations.

Example

The following example contains templates that include a `sizeof` and a binary operation:

`file1.cpp`

```
template <class T, T N> struct S {};
// f1 undefined if file1.cpp is compiled with 5.04 and file2.cpp is
// compiled with 5.05
template <class T> int f1(S<T, sizeof(char)>);
// f2 undefined if file1.cpp is compiled with 5.05 with --cpp11 or
// with 5.04 and file2.cpp is compiled with 5.05 without --cpp11.
template <class T> int f2(S<T, 1+1>);
int function()
{
    S<int,sizeof(char)> s1;
    S<int,1+1> s2;
    return f1(s1) + f2(s2);
}
```

`file2.cpp`

```
template <class T, T N> struct S {};
template <class T> int f1(S<T, sizeof(char)>)
{
    return 1;
}
template <class T> int f2(S<T, 1+1>)
{
    return 2;
}

template int f1<int>(S<int,sizeof(char)>);
template int f2<int>(S<int,1+1>);

extern int function();

int main()
{
    function();
}
```

Substitution of _Complex types

`_Complex` types are only available in C++ when using `--gnu`.

In ARM Compiler v5.04, `_Complex` types failed to have the substitution mechanism applied when generating name manglings. This has been fixed in ARM Compiler v5.05. However, this means that problems might occur when all of the following conditions are true:

- A translation unit is compiled with ARM Compiler v5.05 using the `--gnu` option.
- A second translation unit is compiled with ARM Compiler v5.04 or earlier using the `--gnu` option.
- The translation units define or refer to a function that takes more than one `_Complex` argument.

Example

For example:

`file1.cpp`

```
// fn is undefined if file1.cpp is compiled with 5.04 and file2 is
// compiled with 5.05.
extern void fn(float _Complex a, float _Complex b);
int main()
{
    float _Complex cf;
    fn(cf, cf);
}
```

`file2.cpp`

```
void fn(float _Complex a, float _Complex b)
{

}
```

Const-volatile-qualified non-member function type as template argument

In ARM Compiler v5.04, `armcc` incorrectly mangled a *const-volatile-qualified* (cv-qualified) non-member function type used as a template argument as if it was not cv-qualified. This has been fixed in ARM Compiler v5.05.

This means that problems might occur when both of the following conditions are true:

- A translation unit is compiled with ARM Compiler v5.05.
- A second translation unit is compiled with ARM Compiler v5.04 or earlier.
- One of these translation units defines or refers to a function that takes as an argument a template with an argument of cv-qualified non-member function type.

Example

`file1.cpp`

```
template <class T> struct A { };
typedef void cfn(int) const;
// f is undefined if file1.cpp is compiled with 5.04 and file2 is
// compiled with 5.05 or vice versa.
extern int f(A<cfn>);
int main()
{
    A<cfn> a;
    f(a);
}
```

`file2.cpp`

```
template <class T> struct A { };
typedef void cfn(int) const;
int f(A<cfn>)
{
    return 1;
}
```

lvalue of function type used as non-type template argument

In v5.04, an lvalue of function type used as a non-type template argument was incorrectly mangled as `&(function_name())` instead of `function_name()`. This has been fixed in v5.05.

This means that a link failure is possible when both of the following situations occur:

- A translation unit is compiled with ARM Compiler v5.05.
- A second translation unit is compiled with ARM Compiler v5.04 or earlier.
- The translation units instantiate or refer to a template with a non-type argument of lvalue of function type.

Example

`file1.cpp`

```
template <class T> struct A
{
    template <void (*PF)()> struct B {};
};
void ff();
// f is undefined if file1.cpp is compiled with 5.04 and file2 is
// compiled with 5.05, or vice versa.
template<class T> typename A<T>::template B<ff> f(T);
int main()
{
    f(1);
}
```

`file2.cpp`

```
template <class T> struct A
{
    template <void (*PF)()> struct B {};
```

```
};
void ff();
template<class T> typename A<T>::template B<ff> f(T)
{
    typename A<T>::template B<ff> ret;
    return ret;
}
template A<int>::B<ff> f<int>(int);
```

Related information

__alignof__.
--cpp11.
--gnu.

Chapter 5
Migrating from ARM Compiler v5.03 to v5.04

Describes the changes that affect migration and compatibility between ARM Compiler v5.03 and v5.04.

It contains the following sections:

5.1 Compatibility of ARM Compiler v5.04 with legacy objects and libraries

Backwards compatibility of objects and libraries built with RVCT 3.0 and earlier is supported provided they have not been built with `--apcs /adsabi`.

Given these restrictions, ARM strongly recommends that you rebuild your entire project, including any user, or third-party supplied libraries, with ARM Compiler v5.04. This is to avoid any potential incompatibilities, and to take full advantage of the improved optimization, enhancements, and new features provided by ARM Compiler v5.04.

Related concepts

1.1 Compatibility between ARM Compiler versions on page 1-13.

Related information

--apcs (armasm).

--apcs (armcc).

5.2 Documentation changes between ARM Compiler v5.03 and v5.04

Various changes have been made to the documentation in ARM Compiler v5.04.

Changes to document structure and titles

The user and reference documents for each of the compiler, assembler, linker, and ARM C and C++ libraries have been merged, and the titles of all documents have changed. The changes are summarized in the following table:

5.03 document	5.04 document
Introducing the ARM Compiler toolchain	*Getting Started Guide*
Developing Software for ARM Processors	*Software Development Guide*
Using the Compiler	*armcc User Guide*
Compiler Reference	Merged into the *armcc User Guide*
Using the Assembler	*armasm User Guide*
Assembler Reference	Merged into the *armasm User Guide*
Using the Linker	*armlink User Guide*
Linker Reference	Merged into the *armlink User Guide*
Using ARM C and C++ Libraries and Floating-Point Support	*ARM C and C++ Libraries and Floating-Point Support User Guide*
ARM C and C++ Libraries and Floating-Point Support Reference	Merged into the *ARM C and C++ Libraries and Floating-Point Support User Guide*
Creating Static Software Libraries with armar	*armar User Guide*
Using the fromelf Image Converter	*fromelf User Guide*
Errors and Warnings Reference	*Errors and Warnings Reference Guide*

Technical changes in the documentation

For technical changes that have been made to the ARM Compiler documentation, see the following revision summaries:

- *Revisions for Migration and Compatibility Guide* on page Appx-A-89 (this document).
- *Revisions for Getting Started Guide*.
- *Revisions for Software Development Guide*.
- *Revisions for armcc User Guide*.
- *Revisions for armasm User Guide*.
- *Revisions for armlink User Guide*.
- *Revisions for ARM C and C++ Libraries and Floating-Point Support User Guide*.
- *Revisions for armar User Guide*.
- *Revisions for fromelf User Guide*.
- *Revisions for Errors and Warnings Reference Guide*.

Chapter 6
Migrating from ARM Compiler v5.02 to v5.03

Describes the changes that affect migration and compatibility between ARM Compiler v5.02 and v5.03.

It contains the following sections:

6.1 Compatibility of ARM Compiler v5.03 with legacy objects and libraries

Backwards compatibility of objects and libraries built with RVCT 3.0 and earlier is supported provided they have not been built with `--apcs /adsabi`.

Given these restrictions, ARM strongly recommends that you rebuild your entire project, including any user, or third-party supplied libraries, with ARM Compiler v5.03. This is to avoid any potential incompatibilities, and to take full advantage of the improved optimization, enhancements, and new features provided by ARM Compiler v5.03.

Related concepts

1.1 Compatibility between ARM Compiler versions on page 1-13.

Related information

--apcs (armasm).

--apcs (armcc).

6.2 Compiler changes between ARM Compiler v5.02 and v5.03

Various changes have been made to `armcc` in ARM Compiler toolchain v5.03.

The following changes have been made in ARM Compiler toolchain v5.03:

- The `armcc` message numbers 3001 to 4001 have been modified. Therefore, if you suppress diagnostic messages, you might have to modify the message numbers.

Related information

Toolchain environment variables.
--version_number (armasm).
--version_number (armcc).
--version_number (armlink).
--version_number (fromelf).
--version_number (armar).

6.3 Documentation changes between ARM Compiler v5.02 and v5.03

Various changes have been made to the documentation in ARM Compiler toolchain v5.03.

For technical changes that have been made to the ARM Compiler toolchain documentation, see the following revision summaries:

- *Revisions for Migration and Compatibility* (this document).
- *Revisions for Introducing the ARM Compiler toolchain*.
- *Revisions for Developing Software for ARM Processors*.
- *Revisions for Using the Compiler*.
- *Revisions for Using the Assembler*.
- *Using the Linker*.
- *Revisions for Using ARM C and C++ Libraries and Floating-Point Support*.
- *Revisions for Creating Static Software Libraries with armar*.
- *Revisions for Using the fromelf Image Converter*.
- *Revisions for Errors and Warnings Reference*.
- *Revisions for Assembler Reference*.
- *Revisions for Compiler Reference*.
- *Revisions for Linker Reference*.
- *Revisions for ARM C and C++ Libraries and Floating-Point Support*.

Note

Building Linux Applications with ARM Compiler toolchain and GNU Libraries is no longer being provided as part of the ARM Compiler documentation set.

Chapter 7
Migrating from ARM Compiler v5.0 to v5.01 or later

Describes the changes that affect migration and compatibility between ARM Compiler v5.0 and v5.01 or later.

It contains the following sections:

7.1 Compatibility of ARM Compiler v5.01 with legacy objects and libraries

Backwards compatibility of objects and libraries built with RVCT 3.0 and earlier is supported provided they have not been built with `--apcs /adsabi`.

Given these restrictions, ARM strongly recommends that you rebuild your entire project, including any user, or third-party supplied libraries, with ARM Compiler v5.01. This is to avoid any potential incompatibilities, and to take full advantage of the improved optimization, enhancements, and new features provided by ARM Compiler v5.01.

Related concepts

1.1 Compatibility between ARM Compiler versions on page 1-13.

Related information

--apcs (armasm).

--apcs (armcc).

7.2 General changes between ARM Compiler v5.0 and v5.01 or later

Various general changes have been made in ARM Compiler toolchain v5.01.

The following changes have been made in ARM Compiler toolchain v5.01:

- A separate NEON compiler license is no longer required in ARM Compiler 5.01 and later.
- The version-specific environment variables have changed to use a single digit version, for example, `ARMCC5INC`.
- The version number reported by the tools using `--version_number` has changed:
  - In version 5.0 and earlier, the format is `VVbbbb`.
  - In version 5.01 and later, the format is `VVVbbbb`.

  For example, version 5.01 build 2345 is reported as `5012345`.

Related information

Toolchain environment variables.
--version_number (armasm).
--version_number (armcc).
--version_number (armlink).
--version_number (fromelf).
--version_number (armar).

7.3 Documentation changes between ARM Compiler v5.0 and v5.01 or later

Various changes have been made to the documentation in ARM Compiler toolchain v5.01.

For technical changes that have been made to the ARM Compiler toolchain documentation, see the following revision summaries:

- *Revisions for Migration and Compatibility* (this document).
- *Revisions for Introducing the ARM Compiler toolchain*.
- *Revisions for Developing Software for ARM Processors*.
- *Revisions for Using the Compiler*.
- *Revisions for Using the Assembler*.
- *Using the Linker*.
- *Revisions for Using ARM C and C++ Libraries and Floating-Point Support*.
- *Revisions for Creating Static Software Libraries with armar*.
- *Revisions for Using the fromelf Image Converter*.
- *Revisions for Errors and Warnings Reference*.
- *Revisions for Assembler Reference*.
- *Revisions for Compiler Reference*.
- *Revisions for Linker Reference*.
- *Revisions for ARM C and C++ Libraries and Floating-Point Support*.
- *Revisions for Building Linux Applications with the ARM Compiler toolchain and GNU Libraries*.

Chapter 8
Migrating from ARM Compiler v4.1 Patch 3 or later to v5.0

Describes the changes that affect migration and compatibility between ARM Compiler v4.1 Patch 3 and v5.0.

It contains the following sections:

8.1 Compatibility of ARM Compiler v5.0 with legacy objects and libraries

Backwards compatibility of objects and libraries built with RVCT 3.0 and earlier is supported provided they have not been built with `--apcs /adsabi`.

Given these restrictions, ARM strongly recommends that you rebuild your entire project, including any user, or third-party supplied libraries, with ARM Compiler v5.0. This is to avoid any potential incompatibilities, and to take full advantage of the improved optimization, enhancements, and new features provided by ARM Compiler v5.0.

Related concepts

1.1 Compatibility between ARM Compiler versions on page 1-13.

Related information

--apcs (armasm).

--apcs (armcc).

8.2 General changes between ARM Compiler v4.1 Patch 3 or later and v5.0

Various general changes have been made in ARM Compiler toolchain v5.0.

The following changes have been made in ARM Compiler toolchain v5.0:

- The tools no longer require any environment variables to be set.
- An additional convention for finding the default header and library directories has been added to the v5.0 tools. When no environment variables or related command-line options are present:
  - The compiler looks in `../include`.
  - The linker looks in `../lib`.

  These locations match the relative paths to the include and library directories from the DS-5 `bin` directory.

Related information

Toolchain environment variables.

8.3 Compiler changes between ARM Compiler v4.1 Patch 3 or later and v5.0

Various changes have been made to armcc in ARM Compiler toolchain v5.0.

The following changes have been made to the compiler:

- The *Edison Design Group* (EDG) front-end used by the compiler has been updated to version 4.1. However, this does not create any compatibility issues.
- In version 4.1, `armcc` searched for paths relative to the current working directory. In version 5.0 and later, it searches relative to the current place.
- If `ARMCC50INC` is not set and `-J` is not present on the command line, the compiler searches for the default includes in `../include`, relative to the location of `armcc.exe`.
- Improved GCC compatibility, and supports a GCC fallback mode.
- The *Link-time code generation* (LTCG) feature is deprecated. As an alternative ARM recommends you use the `--multifile` compiler option.
- Profiler-guided optimization with `--profile` is deprecated, and is not currently compatible with ARM Streamline.

Related information

Using GCC fallback when building applications.
Compiler search rules and the current place.
-Jdir[,dir,...] (armcc).
--multifile, --no_multifile (armcc).
-Warmcc,--gcc_fallback (armcc).
Predefined macros.
Toolchain environment variables.

8.4 Linker changes between ARM Compiler v4.1 Patch 3 or later and v5.0

Various changes have been made to armlink in ARM Compiler toolchain v5.0.

The following changes have been made to the linker:

- If `ARMCC50LIB` is not set and `--libpath` is not present on the command line, the linker searches for the default libraries in `../lib`, relative to the location of `armlink.exe`.
- The *Link-time code generation* (LTCG) feature is deprecated. As an alternative ARM recommends you use the `--multifile` compiler option.
- Profiler-guided optimization with `--profile` is deprecated, and is not currently compatible with ARM Streamline.

Related information

--libpath=pathlist (armlink).

--multifile, --no_multifile (armcc).

Toolchain environment variables.

8.5 Documentation changes between ARM Compiler v4.1 Patch 3 or later and v5.0

Various changes have been made to the documentation in ARM Compiler v5.0.

For technical changes that have been made to the ARM Compiler toolchain documentation, see the following revision summaries:

- *Revisions for Migration and Compatibility* (this document).
- *Revisions for Introducing the ARM Compiler toolchain*.
- *Revisions for Developing Software for ARM Processors*.
- *Revisions for Using the Compiler*.
- *Revisions for Using the Assembler*.
- *Using the Linker*.
- *Revisions for Using ARM C and C++ Libraries and Floating-Point Support*
- *Revisions for Creating Static Software Libraries with armar*.
- *Revisions for Using the fromelf Image Converter*.
- *Revisions for Errors and Warnings Reference*.
- *Revisions for Assembler Reference*.
- *Revisions for Compiler Reference*
- *Revisions for Linker Reference*.
- *Revisions for ARM C and C++ Libraries and Floating-Point Support*.
- *Revisions for Building Linux Applications with the ARM Compiler toolchain and GNU Libraries*.

Chapter 9
Migrating from ARM Compiler v4.1 build 561 to v4.1 Patch 3 or later

Describes the changes that affect migration and compatibility between ARM Compiler v4.1 build 561 and v4.1 Patch 3 or later.

It contains the following sections:

9.1 Compatibility of ARM Compiler v4.1 Patch 3 with legacy objects and libraries

Backwards compatibility of objects and libraries built with RVCT 3.0 and earlier is supported provided they have not been built with `--apcs /adsabi`.

Given these restrictions, ARM strongly recommends that you rebuild your entire project, including any user, or third-party supplied libraries, with ARM Compiler v4.1 Patch 3. This is to avoid any potential incompatibilities, and to take full advantage of the improved optimization, enhancements, and new features provided by ARM Compiler v4.1 Patch 3.

Related concepts

1.1 Compatibility between ARM Compiler versions on page 1-13.

Related information

--apcs (armasm).

--apcs (armcc).

9.2 C and C++ library changes between ARM Compiler v4.1 build 561 and v4.1 Patch 3 or later

Various changes have been made to the ARMC C and C++ library in ARM Compiler toolchain v4.1 Patch 3.

The new implementation of `alloca()` allocates memory on the stack and not on the heap. This does not cause compatibility problems with software that assumes the old heap-based implementation of `alloca()`. However, such software might contain operations that are no longer required, such as implementing `__user_perthread_libspace` for the `alloca` state that is no longer used.

Related information

Library heap usage requirements of the ARM C and C++ libraries.
Use of the __user_libspace static data area by the C libraries (none).
Building an application without the C library.

Chapter 10
Migrating from ARM Compiler v4.1 to v4.1 build 561

Describes the changes that affect migration and compatibility between ARM Compiler v4.1 and v4.1 build 561.

It contains the following sections:

10.1 Compatibility of ARM Compiler v4.1 build 561 with legacy objects and libraries

Backwards compatibility of objects and libraries built with RVCT 3.0 and earlier is supported provided they have not been built with `--apcs /adsabi`.

Given these restrictions, ARM strongly recommends that you rebuild your entire project, including any user, or third-party supplied libraries, with ARM Compiler v4.1 build 561. This is to avoid any potential incompatibilities, and to take full advantage of the improved optimization, enhancements, and new features provided by ARM Compiler v4.1 build 561.

Related concepts

1.1 Compatibility between ARM Compiler versions on page 1-13.

Related information

--apcs (armasm).

--apcs (armcc).

10.2 Compiler changes between ARM Compiler v4.1 and v4.1 build 561

Various changes have been made to armcc in ARM Compiler toolchain v4.1 build 561.

The compiler faults use of the `at` attribute when it is used on declarations with incomplete non-array types. For example, if `foo` is not declared, the following causes an error:

```
struct foo a __attribute__((at(0x16000)));
```

Related information

__attribute__((at(address))) variable attribute.

10.3 Linker changes between ARM Compiler v4.1 and v4.1 build 561

There are no technical changes to `armlink` that affect migration between ARM Compiler v4.1 and v4.1 build 561.

10.4 Assembler changes between ARM Compiler v4.1 and v4.1 build 561

There are no technical changes to `armasm` that affect migration between ARM Compiler v4.1 and v4.1 build 561.

10.5 C and C++ library changes between ARM Compiler v4.1 and v4.1 build 561

Various changes have been made to the ARM C and C++ library in ARM Compiler toolchain v4.1 build 561.

The symbol `__use_accurate_range_reduction` is retained for backward compatibility, but no longer has any effect.

The C99 complex number functions in the previous hardware floating point version of the library only had the *hardfp* linkage functions and not the *softfp* linkage functions. The new library has both the *hardfp* linkage and *softfp* linkage functions. This means that existing object code that was built to use hardware floating point might not function correctly when calling complex functions from the library. The linker issues a warning in this case. You must recompile all the code that might use the affected functions and that was built to use hardware floating point. You must relink them with the new library.

Related concepts

1.1 Compatibility between ARM Compiler versions on page 1-13.

Related information

--apcs=qualifier...qualifier (armcc).

--fpu=name (armcc).

10.6 fromelf changes between ARM Compiler v4.1 and v4.1 build 561

`fromelf` can now process all files, or a subset of files, in an archive.

Related information

input_file (fromelf).

10.7 Documentation changes between ARM Compiler v4.1 and v4.1 build 561

Various changes have been made to the documentation in ARM Compiler toolchain v4.1 build 561.

For technical changes that have been made to the ARM Compiler toolchain documentation, see the following revision summaries:

- *Revisions for Migration and Compatibility* (this document).
- *Revisions for Introducing the ARM Compiler toolchain*.
- *Revisions for Developing Software for ARM Processors*.
- *Revisions for Using the Compiler*.
- *Revisions for Using the Assembler*.
- *Using the Linker*.
- *Revisions for Using ARM C and C++ Libraries and Floating-Point Support*.
- *Revisions for Using the fromelf Image Converter*.
- *Revisions for Errors and Warnings Reference*.
- *Revisions for Assembler Reference*
- *Revisions for Compiler Reference*.
- *Revisions for Linker Reference*.
- *Revisions for ARM C and C++ Libraries and Floating-Point Support*.
- *Revisions for Building Linux Applications with the ARM Compiler toolchain and GNU Libraries*.

Chapter 11
Migrating from RVCT v4.0 to ARM Compiler v4.1

Describes the changes that affect migration and compatibility between RVCT v4.0 and ARM Compiler v4.1.

It contains the following sections:

11.1 General changes between RVCT v4.0 and ARM Compiler v4.1

Various general changes have been made in ARM Compiler toolchain v4.1.

The convention for naming environment variables, such as those for setting default header and library directories, has changed. These are now prefixed with `ARMCC` rather than `RVCT`. For example, `ARMCC41INC` rather than `RVCT40INC`.

Compatibility of ARM Compiler v4.1 with legacy objects and libraries

Backwards compatibility of objects and libraries built with RVCT 3.0 and earlier is supported provided they have not been built with `--apcs /adsabi`.

Given these restrictions, ARM strongly recommends that you rebuild your entire project, including any user, or third-party supplied libraries, with ARM Compiler v4.1. This is to avoid any potential incompatibilities, and to take full advantage of the improved optimization, enhancements, and new features provided by ARM Compiler v4.1.

Related concepts

1.1 Compatibility between ARM Compiler versions on page 1-13.

Related information

Toolchain environment variables.

11.2 Compiler changes between RVCT v4.0 and ARM Compiler v4.1

Various changes have been made to armcc in ARM Compiler toolchain v4.1.

Sign rules on enumerators has changed in line with convention. Enumerator container is now unsigned unless a negative constant is defined. The RVCT v4.0 10Q1 patch made this change in GCC mode only.

`-O3` no longer implies `--multifile`. The `--multifile` option has always been available as a separate option and it is recommended you put this into your builds.

Related references

11.1 General changes between RVCT v4.0 and ARM Compiler v4.1 on page 11-58.

Related information

--multifile, --no_multifile (armcc).

-Onum (armcc).

11.3 Linker changes between RVCT v4.0 and ARM Compiler v4.1

Various changes have been made to armlink in ARM Compiler toolchain v4.1.

The following changes to the linker have been made:

GNU ld script support

`armlink` v4.1 supports a subset of GNU linker control scripts. To more closely match the behavior of GNU ld, `armlink` uses an internal linker control script when you specify the `--sysv` command-line option. In previous versions of `armlink` an internal scatter file was used.

The use of a control script produces a logically equivalent, but physically different layout to RVCT v4.0. To revert back to the default scatter file layout use the command-line option `--no_use_sysv_default_script`.

You can replace the internal control script with a user-defined control script using the `-T` option.

ARM/Thumb synonyms removed

Support for the deprecated feature ARM/Thumb Synonyms has been removed in 4.1. The ARM/Thumb synonyms feature permits an ARM Global Symbol Definition of symbol `S` and a Thumb Global Symbol Definition of `S` to coexist. All branches from ARM state are directed at the ARM Definition, all branches from Thumb state are directed at the Thumb Definition.

In 4.0 `armlink` gives a deprecated feature warning `L6455E` when it detects ARM/Thumb Synonyms.

In 4.1 `armlink` gives an error message `L6822E` when it detects ARM/Thumb Synonyms.

To recreate the behavior of synonyms rename both the ARM and Thumb definitions and relink. For each undefined symbol you have to point it at the ARM or the Thumb synonym.

Related references

11.1 General changes between RVCT v4.0 and ARM Compiler v4.1 on page 11-58.

Related information

--use_sysv_default_script, --no_use_sysv_default_script linker option.
--sysv (armlink).

11.4 Assembler changes between RVCT v4.0 and ARM Compiler v4.1

Various changes have been made to armasm in ARM Compiler toolchain v4.1.

The following changes to the assembler have been made:

Change to the way the assembler reads and processes files

Older assemblers sometimes permitted the source file being assembled to introduce new lines of code between the two passes of the assembler. In the following example, the symbol `num` is defined in the second pass because the symbol `foo` is not defined in the first pass when :DEF: foo is evaluated.

```
        AREA x,CODE
        [ :DEF: foo
num     EQU 42
        ]
foo     DCD num
        END
```

The way the assembler reads and processes the file has now changed, and is stricter. You must rewrite code such as this to ensure that the path through the file is the same in both passes. The introduction of new lines in the second pass of the assembler is considered an error, and results in the following error:

```
Error: A1903E : Line not seen in first pass; cannot be assembled.
```

The following code shows another example where the assembler faults:

```
       AREA FOO, CODE
       IF :DEF: VAR
       ELSE
VAR    EQU 0
       ENDIF
       END
```

To avoid this error, you must rewrite this code as:

```
       AREA FOO, CODE
       IF :LNOT: :DEF: VAR
VAR    EQU 0
       ENDIF
       END
```

This works correctly because new lines are not seen in the second pass of the assembler. Instead the lines that were in the first pass are ignored in the second pass.

Change to messages output by the assembler

Generally, any messages referring to a position on the source line now has a caret character pointing to the offending part of the source line, for example:

```
"foo.s", line 3 (column 19): Warning: A1865W: '#' not seen before constant expression
    3 00000000         ADD r0,r1,1
                               ^
```

Changes to diagnostic messages

In the ARM instruction set, various instructions using SP (r13) were deprecated when Thumb-2 technology was introduced. These instructions are no longer diagnosed as deprecated unless assembling for a CPU that supports Thumb-2 technology. To enable the warnings on earlier CPUs, you can use the option `--diag_warning=1745,1786,1788,1789,1892`. This change was introduced in the RVCT v4.0 patch 5 (build 697).

Obsolete command-line option

The `-O` command-line option is obsolete. Use `-o` instead.

Related references

11.1 General changes between RVCT v4.0 and ARM Compiler v4.1 on page 11-58.

Related information

--diag_warning=tag{, tag} (armasm).
-o filename (armasm).
How the assembler works.
2 pass assembler diagnostics.

11.5 C and C++ library changes between RVCT v4.0 and ARM Compiler v4.1

Various changes have been made to the ARM C and C++ library in ARM Compiler toolchain v4.1.

The libraries now use more 32-bit encoded Thumb code on targets that support Thumb-2 technology. This is expected to result in reduced code size without affecting performance. The linker option `--no_thumb2_library` falls back to the old-style libraries if necessary.

Math function returns in some corner cases now conform to POSIX/C99 requirements. You can enable older behavior with:

```
#pragma import __use_rvct_matherr
```

From RVCT v4.0 09Q4 patch onwards, you can enable the newer behavior with:

`#pragma import __use_c99_matherr`.

Related references

11.1 General changes between RVCT v4.0 and ARM Compiler v4.1 on page 11-58.

Related information

How the ARM C library fulfills ISO C specification requirements.

--thumb2_library, --no_thumb2_library (armlink).

Chapter 12
Migrating from RVCT v3.1 to RVCT v4.0

Describes the changes that affect migration and compatibility between RVCT v3.1 and RVCT v4.0.

It contains the following sections:

12.1 Default --gnu_version changed from 303000 (GCC 3.3) to 402000 (GCC 4.2)

The default GNU version has changed in RVCT v4.0.

This affects which GNU extensions are accepted, such as `__attribute__((visibility(…)))` and lvalue casts, even in non-GNU modes.

Related information

--gnu_version=version (armcc).

12.2 General changes between RVCT v3.1 and RVCT v4.0

Various general changes have been made in RVCT v4.0.

The following changes affect multiple tools:

Restrictions on --fpu

`--fpu=VFPv2` or `--fpu=VFPv3` are only accepted if CPU architecture is greater than or equal to ARMv5TE. This affects all tools that accept `--fpu`.

Note

The assembler assembles VFP instructions when you use the `--unsafe` option, so do not use `--fpu` when using `--unsafe`. If you use `--fpu` with `--unsafe`, the assembler downgrades the reported architecture error to a warning.

Remove support for v5TExP and derivatives, and all ARMv5 architectures without T

The following `--cpu` choices are obsolete and have been removed:

- 5.
- 5E.
- 5ExP.
- 5EJ.
- 5EWMMX2.
- 5EWMMX.
- 5TEx.
- ARM9E-S-rev0.
- ARM946E-S-rev0.
- ARM966E-S-rev0.

Compatibility of RVCT v4.0 with legacy objects and libraries

Backwards compatibility of RVCT v2.x, v3.x and v4.0 object and library code is supported provided you have not built them with `--apcs /adsabi` and use the RVCT v4.0 linker and C/C++ libraries. Forward compatibility is not guaranteed.

Given these restrictions, ARM strongly recommends that you rebuild your entire project, including any user, or third-party supplied libraries, with RVCT v4.0 and later. This is to avoid any potential incompatibilities, and to take full advantage of the improved optimization, enhancements, and new features provided by RVCT v4.0 and later.

Related information

--apcs=qualifier...qualifier (armcc).
--cpu=name (armcc).
--fpu=name (armcc).
--cpu=name (armlink).
--fpu=name (armlink).
--apcs=qualifier…qualifier (armasm).
--cpu=name (armasm).
--fpu=name (armasm).
--unsafe (armasm).

12.3 Changes to symbol visibility between RVCT v3.1 and RVCT v4.0

Changes to symbol visibility have been made in RVCT v4.0.

The following changes to symbol visibility have been made:

Change the ELF visibility that represents __declspec(dllexport)

When using the `--hide_all` compiler command-line option, which is the default, the ELF visibility that represents `__declspec(dllexport)` in RVCT v3.1 and earlier was STV_DEFAULT. In RVCT v4.0 it is STV_PROTECTED. Symbols that are STV_PROTECTED can be referred to by other DLLs but cannot be preempted at load-time.

When using the `--no_hide_all` command-line option, the visibility of imported and exported symbols is still STV_DEFAULT as it was in RVCT v3.1.

__attribute(visibility(…))

The GNU-style `__attribute(visibility(…))` has been added and is available even without specifying the `--gnu` compiler command-line option. Using it overrides any implicit visibility. For example, the following results in STV_DEFAULT visibility instead of STV_HIDDEN:

```
__declspec(visibility("default")) int x = 42;
```

RVCT v3.1 symbol visibility summary

The following tables summarize the visibility rules in RVCT v3.1:

Table 12-1 RVCT v3.1 symbol visibility summary

Code	`--hide_all` (default)	`--no_hide_all`	`--dllexport_all`
`extern int x;` `extern int g(void);`	STV_HIDDEN	STV_DEFAULT	STV_HIDDEN
`extern int y = 42;` `extern int f() { return g() + x; }`	STV_HIDDEN	STV_DEFAULT	STV_DEFAULT
`__declspec(dllimport) extern int imx;` `__declspec(dllimport) extern int img(void);`	STV_DEFAULT	STV_DEFAULT	STV_DEFAULT
`__declspec(dllexport) extern int exy = 42;` `__declspec(dllexport) extern int exf() { return img() + imx; }`	STV_DEFAULT	STV_DEFAULT	STV_DEFAULT

`/* exporting undefs (unusual?) */`
`__declspec(dllexport) extern int exz;`
`__declspec(dllexport) extern int exh(void);`
	STV_HIDDEN	STV_HIDDEN	STV_HIDDEN

Table 12-2 RVCT v3.1 symbol visibility summary for references to run-time functions

Code	`--no_dllimport_runtime` `--hide_all` (default)	`--no_hide_all`	`--dllexport_all`
`/* references to runtime functions, for example __aeabi_fmul */ float fn(float a, float b) { return a*b; }`	STV_HIDDEN	STV_DEFAULT	STV_DEFAULT

RVCT v4.0 symbol visibility summary

The following tables summarize the visibility rules in RVCT v4.0:

Table 12-3 RVCT v4.0 symbol visibility summary

Code	`--hide_all` (default)	`--no_hide_all`	`--dllexport_all`
`extern int x; extern int g(void);`	STV_HIDDEN	STV_DEFAULT	STV_HIDDEN
`extern int y = 42; extern int f() { return g() + x; }`	STV_HIDDEN	STV_DEFAULT	STV_PROTECTED
`__declspec(dllimport) extern int imx; __declspec(dllimport) extern int img(void);`	STV_DEFAULT	STV_DEFAULT	STV_DEFAULT
`__declspec(dllexport) extern int exy = 42; __declspec(dllexport) extern int exf() { return img() + imx; }`	STV_PROTECTED	STV_PROTECTED	STV_PROTECTED
`/* exporting undefs (unusual?) */ __declspec(dllexport) extern int exz; __declspec(dllexport) extern int exh(void);`	STV_PROTECTED	STV_PROTECTED	STV_PROTECTED

Table 12-4 RVCT v4.0 symbol visibility summary for references to run-time functions

Code	`--no_dllimport_runtime` `--hide_all` (default)	`--no_hide_all`	`--dllexport_all`
`/* references to runtime functions, for example __aeabi_fmul */ float fn(float a, float b) { return a*b; }`	STV_HIDDEN	STV_DEFAULT	STV_DEFAULT

Related references

12.2 General changes between RVCT v3.1 and RVCT v4.0 on page 12-66.

Related information

--default_definition_visibility=visibility compiler option.
--dllexport_all, --no_dllexport_all (armcc).
--dllimport_runtime, --no_dllimport_runtime compiler option.

--gnu (armcc).
--hide_all, --no_hide_all (armcc).

12.4 Compiler changes between RVCT v3.1 and RVCT v4.0

Various changes have been made to `armcc` in RVCT v4.0.

The following compiler changes have been made:

Single compiler executable

The executables `tcc`, `armcpp` and `tcpp` are no longer delivered.

To compile for Thumb, use the `--thumb` command-line option.

To compile for C++, use the `--cpp` command-line option.

Note

The compiler automatically selects C++ for files with the `.cpp` extension, as before.

Vectorizing compiler

The NEON vectorizing compiler is provided as standard functionality, and is no longer provided as a separate add-on. A license to use the NEON vectorizing compiler is provided with the Professional Edition of the ARM development tools.

VAST Changes

VAST has been upgraded through two versions (VAST 11 for 4.0 Alpha and 4.0 Alpha2 and later). Apart from the following issue, you do not have to make any changes to your v3.1 builds to use the new VAST.

RVCT v3.1 reassociated saturating ALU operations. This meant programs like the following could produce different results with `--vectorize` and `--no_vectorize`:

```
int g_448464(short *a, short *b, int n)
{
    int i; short s = 0;
    for (i = 0; i < n; i++)  s = L_mac(s, a[i], b[i]);
    return s;
}
```

In RVCT v4.0, you might see a performance degradation because of this issue.

The `--reassociate_saturation` and `--no_reassociate_saturation` command-line options have been added to permit reassociation to occur.

Related references

12.2 General changes between RVCT v3.1 and RVCT v4.0 on page 12-66.

Related information

--cpp (armcc).

--reassociate_saturation, --no_reassociate_saturation compiler option.

--thumb (armcc).

--vectorize, --no_vectorize (armcc).

12.5 Linker changes between RVCT v3.1 and RVCT v4.0

Various changes have been made to armlink in RVCT v4.0.

The following linker changes have been made:

Helper functions

Before RVCT v4.0, helper functions were implemented in helper library files such as `h_tf.l`, provided with the ARM compiler. All helper library filenames start with `h_`. RVCT v4.0 no longer requires helper libraries. Instead, all helper functions are generated by the compiler and become part of an object file.

Placing ARM library helper functions with scatter files

In RVCT v3.1 and earlier, the helper functions reside in libraries provided with the ARM compiler. Therefore, it was possible to use `armlib` and `cpplib` in a scatter file to inform the linker where to place these helper functions in memory.

In RVCT v4.0 and later, the helper functions are generated by the compiler in the resulting object files. They are no longer in the standard C libraries, so it is no longer possible to use `armlib` and `cpplib` in a scatter file. Instead, place the helper functions using `*.* (i.__ARM_*)` in your scatter file.

Warning L6932W when linking with helper libraries

When using RVCT v4.0 and later, you might see the following linker warning:

```
Warning: L6932W: Library reports warning: use of helper library h_xx.l is deprecated
```

Before RVCT v4.0, one reason for linking with a helper library is if an object file references the helper function `__ARM_switch8`. You can find this out by looking at verbose output from the linker using the `--verbose` option, for example:

```
Loading member object1.o from lib1.a.
reference :  strncmp
reference :  __ARM_switch8
```

In RVCT v4.0 and later, because helper libraries are no longer required, the verbose output from the linker might show, for example:

```
Loading member object2.o from lib2.a.
…
definition:  __ARM_common_switch8_thumb
```

In this case, the helper function `__ARM_common_switch8_thumb` is in the object file `object2.o`, rather than in a helper library.

So, if you are using RVCT v4.0 and the linker is generating warning message L6932W, it is likely that you are linking with objects or libraries that were built with RVCT v3.1, not RVCT v4.0.

Linker steering files and symbol visibility

In RVCT v3.1 the visibility of a symbol was overridden by the steering file or `.directive` commands `IMPORT` and `EXPORT`. When this occurred the linker issued a warning message, for example:

```
Warning: L6780W: STV_HIDDEN visibility removed from symbol hidden_symbol through
EXPORT.
```

In RVCT v4.0 the steering file mechanism respects the visibility of the symbol, so an `IMPORT` or `EXPORT` of a STV_HIDDEN symbol is ignored. You can restore the v3.1 behavior with the `--override_visibility` command-line option.

Linker-defined symbols

In the majority of cases region related symbols behave identically to v3.1.

Section-relative symbols

The execution region Base and Limit symbols are now section-relative. There is no sensible section for a `$$Length` symbol so this remains absolute.

This means that the linker-defined symbol is assigned to the most appropriate section in the execution region. The following example shows this:

```
ExecRegion ER
RO Section 1 ; Image$$ER$$Base and Image$$ER$$RO$$Base, val 0
RO Section 2 ; Image$$ER$$RO$$Limit, val Limit(RO Section 2)
RW Section 1 ; Image$$ER$$RW$$Base, val 0
RW Section 2 ; Image$$ER$$Limit and Image$$ER$$RW$$Limit,
             ; val Limit(RW Section 2)
ZI Section 1 ; Image$$ER$$ZI$$Base, val 0
ZI Section 2 ; Image$$ER$$ZI$$Limit, val Limit(ZI Section 2)
```

In each case the value of the …`$$Length` symbol is the value of the …`$$Limit symbol` minus the …`$$Base` symbol.

If there is no appropriate section that exists in the execution region then the linker defines a zero-sized section of the appropriate type to hold the symbols.

Impact of the change

The change to section-relative symbols removes several special cases from the linker implementation, that might improve reliability. It also means that dynamic relocations work naturally on SysV and BPABI links.

Alignment

The …`$$Limit` symbols are no longer guaranteed to be four-byte aligned because the limit of the section it is defined in might not be aligned to a four-byte boundary.

This might affect you if you have code that accidentally relies on the symbol values being aligned. If you require an aligned `$$Limit` or `$$Length` then you must align the symbol value yourself.

For example, the following legacy initialization code might fail if `Image$$<Region_Name>$$Length` is not word aligned:

```
    LDR R1,|Load$$region_name$$Base|
    LDR R0,|Image$$region_name$$Base|
    LDR R4,|Image$$region_name$$Length|
    ADD R4, R4, R0
copy_rw_data
    LDRNE  R3,[R1],#4
    STRNE  R3,[R0],#4
    CMP  R0,R4
    BNE  copy_rw_data
```

Writing your own initialization code is not recommended, because system initialization is more complex than in earlier toolchain releases. ARM recommends that you use the `__main` code provided with the ARM Compiler toolchain.

Delayed Relocations

The linker has introduced an extra address assignment and relocation pass after RW compression. This permits more information about load addresses to be used in linker-defined symbols.

Be aware that:

- `Load$$`*`region_name`*`$$Base` is the address of *`region_name`* prior to C-library initialization
- `Load$$`*`region_name`*`$$Limit` is the limit of *`region_name`* prior to C-library initialization
- `Image$$`*`region_name`*`$$Base` is the address of *`region_name`* after C-library initialization
- `Image$$`*`region_name`*`$$Limit` is the limit of *`region_name`* after C-library initialization.

Load Region Symbols have the following properties:

- They are ABSOLUTE because section-relative symbols can only have Execution addresses.
- They take into account RW compression
- They do not include ZI because it does not exist prior to C-library initialization.

In addition to `Load$$`*`region_name`*`$$Base`, the linker now supports the following linker-defined symbols:

```
Load$$region_name$$Limit
Load$$region_name$$Length
Load$$region_name$$RO$$Base
Load$$region_name$$RO$$Limit
Load$$region_name$$RO$$Length
Load$$region_name$$RW$$Base
Load$$region_name$$RW$$Limit
Load$$region_name$$RW$$Length
```

Limits of Delayed Relocation

All relocations from RW compressed execution regions must be performed prior to compression because the linker cannot resolve a delayed relocation on compressed data.

If the linker detects a relocation from a RW-compressed region `REGION` to a linker-defined symbol that depends on RW compression then the linker disables compression for `REGION`.

Load Region Symbols

RVCT v4.0 now permits linker-defined symbols for load regions. They follow the same principle as the `Load$$` symbols for execution regions. Because a load region might contain many execution regions it is not always possible to define the `$$RO` and `$$RW` components. Therefore, load region symbols only describe the region as a whole.

```
; Base address of <Load Region Name>
Load$$LR$$Load_Region_Name$$Base

; Load Address of last byte of content in Load region.
Load$$LR$$Load_Region_Name$$Limit

; Limit - Base
Load$$LR$$Load_Region_Name$$Length
```

Image-related symbols

The RVCT v4.0 linker implements these in the same way as the execution region-related symbols.

They are defined only when scatter files are not used. This means that they are available for the `--sysv` and `--bpabi` linking models.

```
Image$$RO$$Base  ; Equivalent to Image$$ER_RO$$Base
Image$$RO$$Limit ; Equivalent to Image$$ER_RO$$Limit
Image$$RW$$Base  ; Equivalent to Image$$ER_RW$$Base
Image$$RW$$Limit ; Equivalent to Image$$ER_RW$$Limit
Image$$ZI$$Base  ; Equivalent to Image$$ER_ZI$$Base
Image$$ZI$$Limit ; Equivalent to Image$$ER_ZI$$Limit
```

Interaction with `ZEROPAD`

An execution region with the `ZEROPAD` keyword writes all ZI data into the file:

- `Image$$` symbols define execution addresses post initialization.

  In this case, it does not matter that the zero bytes are in the file or generated. So for `Image$$` symbols, `ZEROPAD` does not affect the values of the linker-defined symbols.
- `Load$$` symbols define load addresses pre initialization.

  In this case, any zero bytes written to the file are visible, Therefore, the `Limit` and `Length` take into account the zero bytes written into the file.

Build attributes

The RVCT v4.0 linker fully supports reading and writing of the ABI Build Attributes section. The linker can now check more properties such as `wchar_t` and `enum` size. This might result in the linker diagnosing errors in old objects that might have inconsistencies in the Build Attributes. Most of the Build Attributes messages can be downgraded to permit `armlink` to continue.

The `--cpu` option now checks the FPU attributes if the CPU chosen has a built-in FPU. For example, `--cpu=cortex-a8` implies `--fpu=vfpv3`. In RVCT v3.1 the `--cpu` option only checked the build attributes of the chosen CPU.

The error message `L6463U`: `Input Objects contain <archtype> instructions but could not find valid target for <archtype> architecture based on object attributes. Suggest using --cpu option to select a specific cpu.` is given in one of two situations:

- The ELF file contains instructions from architecture *`archtype`* yet the Build Attributes claim that *`archtype`* is not supported.
- The Build Attributes are inconsistent enough that the linker cannot map them to an existing CPU.

If setting the `--cpu` option still fails, the option `--force_explicit_attr` causes the linker to retry the CPU mapping using Build Attributes constructed from `--cpu=`*`archtype`*. This might help if the Error is being given solely because of inconsistent Build Attributes.

C library initialization

A change to the linker when dealing with C library initialization code causes specially named sections in the linker map file created with the `--map` command-line option. You can ignore these specially named sections.

ARM Linux

When building a shared object the linker automatically imports any reference with STV_DEFAULT visibility that is undefined. This matches the behavior of GCC. This might result in some failed links now being successful.

Prelink support reserves some extra space, this results in slightly larger images and shared objects. The prelink support can be turned off with `--no_prelink_support`.

There have been numerous small changes with regards to symbol visibility, these are described in the Symbol visibility changes.

RW compression

Some error handling code is run later so that information from RW compression can be used. In almost all cases, this means more customer programs are able to link. There is one case where RVCT v4.0 has removed a special case so that it could diagnose more RW compression errors.

Multiple in-place execution regions with RW compression are no longer a special case. It was possible to write:

```
LR1 0x0
{
    ER1 +0 { file1.o(+RW) }
    ER2 +0 { file2.o(+RW) }
}
```

This is no longer possible under v4.0 and the linker gives an error message that ER1 decompresses over ER2. This change has been made to permit the linker to diagnose:

```
LR1 0x0
{
    ER1 +0 { file1.o(+RW) }
    ER2 +0 { file2.o(+RO) }   ; NOTE RO not RW
}
```

This fails at runtime on RVCT v3.1.

Related references

12.2 General changes between RVCT v3.1 and RVCT v4.0 on page 12-66.

Related information

Optimization with RW data compression.
Accessing linker-defined symbols.
Region-related symbols.
Image$$ execution region symbols.
Load$$ execution region symbols.
Importing linker-defined symbols in C and C++.
Section-related symbols.
Example of placing ARM library helper functions.
Initialization of the execution environment and execution of the application.
--bpabi (armlink).
--cpu=name (armlink).
--force_explicit_attr (armlink).
--fpu=name (armlink).
--map, --no_map (armlink).
--override_visibility (armlink).
--prelink_support, --no_prelink_support (armlink).
--sysv (armlink).
--verbose (armlink).
EXPORT.
IMPORT.
Execution region attributes.

12.6 Assembler changes between RVCT v3.1 and RVCT v4.0

Various changes have been made to armasm in RVCT v4.0.

The following changes to the assembler have been made:

- The `-O` command-line option is deprecated. `-O` is a synonym for `-o` to output to a named file. This has been deprecated to avoid user confusion with the `armcc` option with the same name.
- The `-D` command-line option is obsolete. Use `--depend` instead.
- `LDM r0!,{r0-r4}` no longer ignores writeback. Previously in Thumb, `LDM r0!,{r0-r4}` assembled with a warning to the 16-bit encoded `LDM` instruction and no writeback was performed. Because the syntax requests writeback, and this encoding is only available when 32-bit encoded Thumb instructions are supported, it produces an error. To get the 16-bit encoded Thumb instruction you must remove the writeback.
- The logical operator `|` is deprecated because it can cause problems with substitution of variables in source. The assembler warns on the first use of `|` as a logical operator. Use the `:OR:` operator instead.

Related references

12.2 General changes between RVCT v3.1 and RVCT v4.0 on page 12-66.

Related information

Addition, subtraction, and logical operators (armasm).

--depend=dependfile (armasm).

-o filename (armasm).

12.7 fromelf changes between RVCT v3.1 and RVCT v4.0

Various changes have been made to fromelf in RVCT v4.0.

Use of single letters as parameters to the `--text` option, either with a `/` or `=` as a separator is obsolete. The syntax `--text/cd` or `--text=cd` are no longer accepted. You have to specify `-cd`.

Related references

12.2 General changes between RVCT v3.1 and RVCT v4.0 on page 12-66.

Related information

--text (fromelf).

12.8 C and C++ library changes between RVCT v3.1 and RVCT v4.0

Various changes have been made to the ARM C and C++ library in RVCT v4.0.

The following changes to the libraries have been made:

Support for non-standard C library math functions

Non-standard C library math functions are no longer supplied in `math.h`. They are still provided in the library itself. You can still request the header file from ARM if needed. Contact your supplier.

Remove __ENABLE_LEGACY_MATHLIB

In RVCT v2.2 changes were made to the behavior of some mathlib functions to bring them in-line with C99. If you relied on the old non-C99 behavior, you could revert the behavior by defining the following at compile time:

```
#define __ENABLE_LEGACY_MATHLIB
```

This has been removed in RVCT v4.0.

Related references

12.2 General changes between RVCT v3.1 and RVCT v4.0 on page 12-66.

Chapter 13
Migrating from RVCT v3.0 to RVCT v3.1

Describes the changes that affect migration and compatibility between RVCT v3.0 and RVCT v3.1.

It contains the following sections:

13.1 General changes between RVCT v3.0 and RVCT v3.1

Various general changes have been made in RVCT v3.1.

The following changes affect multiple tools:

- Support for the old ABI (`--apcs=/adsabi`) is no longer available.
- `-O3` no longer implies `--fpmode=fast`.

Compatibility of RVCT v3.1 with legacy objects and libraries

RVCT v3.1 removes the option `--apcs /adsabi`. Compilation of ADS-compatible objects, and linking of legacy ADS objects and libraries is no longer possible.

Backwards compatibility of RVCT 2.x object and library code is supported provided you use the RVCT v3.1 linker and C/C++ libraries. Forward compatibility is not guaranteed.

Given these restrictions, ARM strongly recommends that you rebuild your entire project, including any user, or third-party supplied libraries, with RVCT v3.1. This is to avoid any potential incompatibilities, and to take full advantage of the improved optimization, enhancements, and new features provided by RVCT v3.1.

Related information

--apcs=qualifier...qualifier (armcc).
--dwarf2 (armcc).
--dwarf3 (armcc).
--fpmode=model (armcc).
-g (armcc).
-Onum (armcc).

13.2 Assembler changes between RVCT v3.0 and RVCT v3.1

Various changes have been made to armasm in RVCT v3.1.

Disassembly output now conforms to the new *Unified Assembly Language* (UAL) format. VFP mnemonics have changed, so that `FMULS` is now `VMUL.F32`.

Related references

13.1 General changes between RVCT v3.0 and RVCT v3.1 on page 13-80.

Related information

Unified Assembler Language.

Assembly language changes after RVCT v2.1.

VFP directives and vector notation.

13.3 Linker changes between RVCT v3.0 and RVCT v3.1

Various changes have been made to armlink in RVCT v3.1.

In a scatter file, special case handling of the load address of root ZI has been removed. Consider:

```
LR1 0x8000
{
    ER_RO +0
    {
        *(+RO)
    }
    ER_RW +0
    {
        *(+RW)
    }
    ER_ZI +0
    {
        *(+ZI)
    }
}
LR2 +0
{
    …
}
```

In versions of RVCT up to v3.0 the linker includes the size of `ER_ZI` when calculating the base address of `LR2`. The justification being that at image initialization `ER_ZI` is written over the top of `LR2`.

In version 3.1 and later this special case has been removed because you can write an equivalent scatter file using `ImageLimit()` built-in function, for example:

```
LR1 0x8000
{
    ER_RO +0
    {
         *(+RO)
    }
    ER_RW +0
    {
        *(+RW)
    }
    ER_ZI +0
    {
        *(+ZI)
    }
}
LR2 ImageLimit(LR1)
{
    …
}
```

Related references

13.1 General changes between RVCT v3.0 and RVCT v3.1 on page 13-80.

Related information

Execution address built-in functions for use in scatter files.

Chapter 14
Migrating from RVCT v2.2 to RVCT v3.0

Describes the changes that affect migration and compatibility between RVCT v2.2 and RVCT v3.0.

It contains the following sections:

14.1 General changes between RVCT v2.2 and RVCT v3.0

Various general changes have been made in RVCT v3.0.

The following changes affect multiple tools:

- DWARF3 is the default.
- Since RVCT v2.1, `-g` no longer implies `-O0`. If you specify `-g` without an optimization level, the following warning is produced:

```
Warning: C2083W: -g defaults to -O2 if no optimisation level is specified
```

Compatibility with legacy RVCT v2.x objects and libraries

Backwards compatibility of RVCT v2.x object and library code is supported provided you use the RVCT v3.0 linker and C/C++ libraries. Forward compatibility is not guaranteed.

You must link using the RVCT v3.0 linker, not the linker of older ARM tools. This is because older linkers cannot process objects produced by the RVCT v3.0 compiler.

Given these restrictions, ARM strongly recommends that you rebuild your entire project, including any user-supplied libraries, with RVCT v3.0 to avoid any potential incompatibilities, and to take full advantage of the improved optimization, enhancements, and new features provided by RVCT v3.0.

Related information

--apcs=qualifier...qualifier (armcc).
--dwarf2 (armcc).
--dwarf3 (armcc).
--fpmode=model (armcc).
-g (armcc).
-Onum (armcc).

14.2 Compiler changes between RVCT v2.2 and RVCT v3.0

Various changes have been made to armcc in RVCT v3.0.

With the resolution of C++ core issue #446, temporaries are now created in some cases of conditional class rvalue expressions where they were not before.

Starting with RVCT v4.0 10Q1 (build 771) you can use the `--diag_warning=2817` command-line option to get a warning if this situation exists.

Related references

14.1 General changes between RVCT v2.2 and RVCT v3.0 on page 14-84.

Related information

--diag_warning=optimizations (armcc).

14.3 Linker changes between RVCT v2.2 and RVCT v3.0

Various changes have been made to armlink in RVCT v3.0.

The following changes have been made to the linker:

- If you specify the compiler option `--fpu=softvfp` together with a CPU with implicit VFP hardware, the linker no longer chooses a library that implements the software floating-point calls using VFP instructions. If you require this legacy behavior, specify the compiler option `--fpu=softvfp+vfp`.
- `armlink` writes the contents of load regions into the output ELF file in the order that load regions are written in the scatter file. Each load region is represented by one ELF program segment. In RVCT v2.2 the Program Header Table entries describing the program segments are given the same order as the program segments in the ELF file. To be more compliant with the ELF specification, in RVCT v3.0 and later the Program Header Table entries are sorted in ascending virtual address order.
- The `--no_strict_ph` command-line option has been added to switch off the sorting of the Program Header Table entries.

Related references

14.1 General changes between RVCT v2.2 and RVCT v3.0 on page 14-84.

Related information

--strict_ph, --no_strict_ph (armlink).

--cpu=name (armcc).

--fpu=name (armcc).

14.4 C and C++ library changes between RVCT v2.2 and RVCT v3.0

Various changes have been made to the ARM C and C++ library in RVCT v3.0.

The function `__user_initial_stackheap()` sets up and returns the locations of the initial stack and heap. In RVCT v3.x and later, ARM recommends you modify your source code to use `__user_setup_stackheap()` instead. `__user_initial_stackheap()` is still supported for backwards compatibility with earlier versions of the ARM C and C++ libraries.

If you continue to use `__user_initial_stackheap()`, be aware of the following changes in RVCT v3.0.

- In RVCT v2.x and earlier, the default implementation of `__user_initial_stackheap()` uses the value of the symbol `Image$$ZI$$Limit`. This symbol is not defined if you specify a scatter file using the `--scatter` linker command line option. Therefore, you must re-implement `__user_initial_stackheap()` to set the heap and stack boundaries if you are using a scatter file, otherwise your link step fails.

  In RVCT v3.x and later, multiple implementations of `__user_initial_stackheap()` are provided by the ARM C library. RVCT automatically selects the correct implementation using information given in the scatter file. This means that you do not have to re-implement this function if you are using scatter files.
- When migrating your application from RVCT v2.2 to RVCT v3.0, you might see the following linker error:

  ```
  Error L6218E: Undefined symbol main (referred from kernel.o).
  ```

  The linker is reporting that your application does not include a `main()` function. The error is generated because of the way RVCT v3.0 selects from the multiple implementations of `__user_initial_stackheap()`. These implementations refer to the `__rt_exit()` function. This is contained in `kernel.o`, that also contains the `__rt_lib_init()` function. Because the `__rt_lib_init()` function calls `main()`, this results in an undefined symbol error when `main()` is not present.

  If you do not provide a `main()` function as the entry point of your C code, the simplest solution is to implement either:
  — An empty, dummy `main()` function.
  — A dummy implementation of `__rt_exit()`.

  If one of these stubs is contained in a separate source file, it is usually removed by the unused section elimination feature of the linker so there is no overhead in the final linked image.

Related references

14.1 General changes between RVCT v2.2 and RVCT v3.0 on page 14-84.

Related information

Elimination of unused sections.
Image symbols.
__user_setup_stackheap() (none).
Legacy function __user_initial_stackheap() (none).
--scatter=filename (armlink).

Appendix A
Migration and Compatibility document revisions

Describes the technical changes that have been made to the Migration and Compatibility Guide.

It contains the following sections:

A.1 Revisions for Migration and Compatibility Guide

The following technical changes have been made to the Migration and Compatibility Guide.

Table A-1 Differences between Issue L and Issue M

Change	Topics affected
Enhanced the topics describing the supported versions of: • FlexNet. • Cygwin.	• *2.1 FlexNet versions supported* on page 2-15 • *2.3 Cygwin versions supported* on page 2-17

Table A-2 Differences between Issue K and Issue L

Change	Topics affected
Added new topic for new default GCC version.	*3.2.2 GCC default version changed* on page 3-20

Table A-3 Differences between Issue J and Issue K

Change	Topics affected
Updated the supported versions of: • FlexNet. • GCC. • Cygwin.	• *2.1 FlexNet versions supported* on page 2-15 • *2.2 GCC versions emulated* on page 2-16 • *2.3 Cygwin versions supported* on page 2-17
Added new topic for compiler changes.	*4.3 Compiler changes between ARM Compiler v5.04 and v5.05* on page 4-25
Added new topics that describe the compatibility with legacy objects and libraries.	• *1.1 Compatibility between ARM Compiler versions* on page 1-13 • *4.1 Compatibility of ARM Compiler v5.05 with legacy objects and libraries* on page 4-23 • *5.1 Compatibility of ARM Compiler v5.04 with legacy objects and libraries* on page 5-30 • *6.1 Compatibility of ARM Compiler v5.03 with legacy objects and libraries* on page 6-33 • *7.1 Compatibility of ARM Compiler v5.01 with legacy objects and libraries* on page 7-37 • *10.1 Compatibility of ARM Compiler v4.1 build 561 with legacy objects and libraries* on page 10-50 • *9.1 Compatibility of ARM Compiler v4.1 Patch 3 with legacy objects and libraries* on page 9-47 • *8.1 Compatibility of ARM Compiler v5.0 with legacy objects and libraries* on page 8-41 • *7.1 Compatibility of ARM Compiler v5.01 with legacy objects and libraries* on page 7-37 • *6.1 Compatibility of ARM Compiler v5.03 with legacy objects and libraries* on page 6-33 • *5.1 Compatibility of ARM Compiler v5.04 with legacy objects and libraries* on page 5-30

Table A-4 Differences between Issue I and Issue J

Change	Topics affected
Added new chapter for changes between v5.03 and v5.04.	*Chapter 5 Migrating from ARM Compiler v5.03 to v5.04* on page 5-29
Enhanced the topics describing the supported versions of: • FlexNet. • GCC. • Cygwin.	• *2.1 FlexNet versions supported* on page 2-15 • *2.2 GCC versions emulated* on page 2-16 • *2.3 Cygwin versions supported* on page 2-17
Added information about the NEON compiler licensing	*7.2 General changes between ARM Compiler v5.0 and v5.01 or later* on page 7-38

Table A-5 Differences between Issue H and Issue I

Change	Topics affected
Added new chapter for changes between v5.02 and v5.03.	*Chapter 6 Migrating from ARM Compiler v5.02 to v5.03* on page 6-32
Where appropriate, changed the terminology that implied that 16-bit Thumb and 32-bit Thumb are separate instruction sets. ——— **Note** ——— There is only one Thumb instruction set. Thumb-2 is the technology, not a separate instruction set. ———	Various topics
Enhanced the topic describing the supported Cygwin versions	*2.3 Cygwin versions supported* on page 2-17

Table A-6 Differences between Issue G and Issue H

Change	Topics affected
Updated the FlexNet version used by ARM Compiler toolchain.	*2.1 FlexNet versions supported* on page 2-15
Corrected the GCC version emulated for ARM Compiler v5.01.	*2.2 GCC versions emulated* on page 2-16
Removed the changes that do not have an impact on migration or compatibility.	*10.3 Linker changes between ARM Compiler v4.1 and v4.1 build 561* on page 10-52

Table A-7 Differences between Issue F and Issue G

Change	Topics affected
Updated the table of FLEX`net` versions.	*2.1 FlexNet versions supported* on page 2-15

Table A-8 Differences between Issue D and Issue F

Change	Topics affected
Added new chapter for changes between v5.0.1 and v5.01.	*Chapter 7 Migrating from ARM Compiler v5.0 to v5.01 or later* on page 7-36
Updated the version of FLEX`net` supported.	*2.1 FlexNet versions supported* on page 2-15
Updated the version of GCC supported.	*2.2 GCC versions emulated* on page 2-16

Table A-8 Differences between Issue D and Issue F (continued)

Change	Topics affected
Added details about backwards compatibility with legacy objects and library code.	• *11.1 General changes between RVCT v4.0 and ARM Compiler v4.1* on page 11-58 • *12.2 General changes between RVCT v3.1 and RVCT v4.0* on page 12-66 • *13.1 General changes between RVCT v3.0 and RVCT v3.1* on page 13-80 • *14.1 General changes between RVCT v2.2 and RVCT v3.0* on page 14-84.
Where appropriate: • changed Thumb-2 to 32-bit Thumb.	• *11.5 C and C++ library changes between RVCT v4.0 and ARM Compiler v4.1* on page 11-63 • *12.6 Assembler changes between RVCT v3.1 and RVCT v4.0* on page 12-76
Modified the description of the C and C++ library changes between RVCT v2.2 and RVCT v3.0.	*14.4 C and C++ library changes between RVCT v2.2 and RVCT v3.0* on page 14-87
Added a description of linker warning L6932W when linking with helper libraries.	*12.5 Linker changes between RVCT v3.1 and RVCT v4.0* on page 12-71

Table A-9 Differences between Issue C and Issue D

Change	Topics affected
Added topic for Cygwin supported versions.	*2.3 Cygwin versions supported* on page 2-17
Added migration chapter for v4.1 Patch 3 to v5.0.	*Chapter 8 Migrating from ARM Compiler v4.1 Patch 3 or later to v5.0* on page 8-40
Added details about how `armlink` chooses libraries when specifying the compiler option `--fpu=softvfp` together with a CPU with implicit VFP hardware.	*14.3 Linker changes between RVCT v2.2 and RVCT v3.0* on page 14-86

Table A-10 Differences between Issue B and Issue C

Change	Topics affected
Added migration topic for v4.1 build 561 to v4.1 Patch 3.	*9.2 C and C++ library changes between ARM Compiler v4.1 build 561 and v4.1 Patch 3 or later* on page 9-48
Added details about placing ARM library helper functions with scatter files.	*12.5 Linker changes between RVCT v3.1 and RVCT v4.0* on page 12-71
Added more examples to demonstrate the change to the 2 pass assembler.	*11.4 Assembler changes between RVCT v4.0 and ARM Compiler v4.1* on page 11-61

Table A-11 Differences between Issue A and Issue B

Change	Topics affected
Added details for migrating from ARM Compiler v4.1 to v4.1 build 561.	*Chapter 10 Migrating from ARM Compiler v4.1 to v4.1 build 561* on page 10-49
Added details for `__user_initial_stackheap()` and `__user_setup_stackheap()` for migrating from v2.2 to v3.0, and later.	*14.4 C and C++ library changes between RVCT v2.2 and RVCT v3.0* on page 14-87

Table A-11 Differences between Issue A and Issue B (continued)

Change	Topics affected
Added details for the addition of *softfp* linkage functions in the hardware floating point version of the library.	*10.5 C and C++ library changes between ARM Compiler v4.1 and v4.1 build 561* on page 10-54
Added details for the deprecation of the | logical operator.	*12.6 Assembler changes between RVCT v3.1 and RVCT v4.0* on page 12-76