ARM® Compiler v5.06 for µVision®

Version 5

Migration and Compatibility Guide

ARM DUI0593E

ARM® Compiler v5.06 for µVision®

Migration and Compatibility Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
A	June 2011	Non-Confidential	Release for ARM Compiler v4.1 for µVision
B	July 2012	Non-Confidential	Release for ARM Compiler v5.02 for µVision
C	30 May 2014	Non-Confidential	Release for ARM Compiler v5.04 for µVision
D	12 December 2014	Non-Confidential	Release for ARM Compiler v5.05 for µVision
E	15 August 2015	Non-Confidential	Release for ARM Compiler v5.06 for µVision

Non-Confidential Proprietary Notice

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 v5.06 for µVision® Migration and Compatibility Guide

List of Tables
ARM® Compiler v5.06 for µVision® Migration and Compatibility Guide

Preface

This preface introduces the *ARM® Compiler v5.06 for µVision® Migration and Compatibility Guide*.

It contains the following:

About this book

ARM® Compiler for µVision® Migration and Compatibility Guide. This manual provides migration and compatibility information between the latest released version and previous versions. It is also available as a PDF.

Using this book

This book is organized into the following chapters:

Chapter 1 Overview of Compatibility
Describes the compatibility between different versions of ARM Compiler.

Chapter 2 Configuration Information for Different Versions of the ARM Compilation Tools
Describes the FlexNet and Cygwin versions supported by the different versions of the ARM compilation tools.

Chapter 3 Migrating from ARM Compiler v5.05 for µVision to ARM Compiler v5.06 for µVision
Describes the changes that affect migration and compatibility between ARM Compiler v5.05 for µVision and ARM Compiler v5.06 for µVision.

Chapter 4 Migrating from ARM Compiler v5.04 for µVision to ARM Compiler v5.05 for µVision
Describes the changes that affect migration and compatibility between ARM Compiler v5.04 for µVision and ARM Compiler v5.05 for µVision.

Chapter 5 Migrating from ARM Compiler v5.02 for µVision to ARM Compiler v5.04 for µVision
Describes the changes that affect migration and compatibility between ARM Compiler v5.02 for µVision and ARM Compiler v5.04 for µVision.

Chapter 6 Migrating from ARM Compiler v4.1 for µVision to ARM Compiler v5.02 for µVision
Describes the changes that affect migration and compatibility between ARM Compiler v4.1 for µVision and ARM Compiler v5.02 for µVision.

Chapter 7 Migrating from RVCT v4.0 for µVision to ARM Compiler v4.1 for µVision
Describes the changes that affect migration and compatibility between RVCT v4.0 for µVision and ARM Compiler v4.1 for µVision.

Chapter 8 Migrating from RVCT v3.1 for µVision to RVCT v4.0 for µVision
Describes the changes that affect migration and compatibility between RVCT v3.1 for µVision and RVCT v4.0 for µVision.

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 v5.06 for µVision® Migration and Compatibility Guide*.
- The number ARM DUI0593E.
- 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 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 support different versions of FlexNet.

The FlexNet versions in the compilation tools are:

Table 2-1 FlexNet versions

Compilation tools version	FlexNet version
ARM Compiler 5.06	11.12.1.0
ARM Compiler 5.05	11.12.1.0
ARM Compiler 5.04	11.10.1.0
ARM Compiler 5.02	10.8.10.0
ARM Compiler 4.1	10.8.7.0
RVCT 4.0	10.8.5.0
RVCT 3.1	10.8.5.0

Related information

ARM DS-5 License Management Guide.

2.2 Cygwin versions supported

Different versions of ARM Compiler support different versions of Cygwin.

The Cygwin versions supported in the compilation tools are:

Table 2-2 Cygwin version supported

Compilation tools version	Cygwin version
ARM Compiler 5.06	2.0.4
ARM Compiler 5.05	1.7.32
ARM Compiler 5.04	1.7.25
ARM Compiler 5.02	1.7.15

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 for µVision to ARM Compiler v5.06 for µVision

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

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 1-11.

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* on page 3-17.

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.

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

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

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 1-11.

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 4-21.

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)`.

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();
}
```

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

--cpp11.

Chapter 5
Migrating from ARM Compiler v5.02 for µVision to ARM Compiler v5.04 for µVision

Describes the changes that affect migration and compatibility between ARM Compiler v5.02 for µVision and ARM Compiler v5.04 for µVision.

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.

5.2 Compiler changes between ARM Compiler v5.02 for µVision and ARM Compiler v5.04 for µVision

Various changes have been made to `armcc` in between ARM Compiler v5.02 for µVision and ARM Compiler v5.04 for µVision.

The following changes to armcc have been made:

- 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

C and C++ Compiler Errors and Warnings.

--diag_suppress (armcc).

5.3 Documentation changes between ARM Compiler v5.02 for µVision and ARM Compiler v5.04 for µVision

Various changes have been made to the documentation between ARM Compiler v5.02 for µVision and ARM Compiler v5.04 for µVision.

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:

5.02 document	5.04 document
Introducing the ARM Compiler toolchain	*Getting Started 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*
Migration and Compatibility	*Migration and Compatibility Guide (This document)*

Chapter 6
Migrating from ARM Compiler v4.1 for µVision to ARM Compiler v5.02 for µVision

Describes the changes that affect migration and compatibility between ARM Compiler v4.1 for µVision and ARM Compiler v5.02 for µVision.

It contains the following sections:

6.1 General changes between ARM Compiler v4.1 for µVision and ARM Compiler v5.02 for µVision

Various general changes have been made between ARM Compiler v4.1 for µVision and ARM Compiler v5.02 for µVision.

The following changes have been made:

- 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`.
- 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 4.1 and earlier, the format is `VVbbbb`.
  - In version 5.02 and later, the format is `VVVbbbb`.

  For example, version 5.02 build 2345 is reported as `5022345`.

Related information

Toolchain environment variables.

--version_number (armasm).

--version_number (armcc).

--version_number (armlink).

--version_number (fromelf).

--version_number (armar).

6.2 Compiler changes between ARM Compiler v4.1 for µVision and ARM Compiler v5.02 for µVision

Various changes have been made to `armcc` between ARM Compiler v4.1 for µVision and ARM Compiler v5.02 for µVision.

The following changes to `armcc` have been made:

- 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.
- If `ARMCC5INC` 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`.
- The *Link-time code generation* (LTCG) feature is deprecated. As an alternative ARM recommends you use the `--multifile` compiler option.

Related information

-Jdir[,dir,...].

--multifile, --no_multifile (armcc).

Toolchain environment variables.

6.3 Linker changes between ARM Compiler v4.1 for µVision and ARM Compiler v5.02 for µVision

Various changes have been made to `armlink` between ARM Compiler v4.1 for µVision and ARM Compiler v5.02 for µVision.

The following changes to `armlink` have been made:

Library searching

If `ARMCC5LIB` 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`.

Link-time code generation

The *Link-time code generation* (LTCG) feature is deprecated. As an alternative ARM recommends you use the `--multifile` compiler option.

Related information

About the ARM Compiler toolchain.

--libpath=pathlist (armlink).

--multifile, --no_multifile (armcc).

Toolchain environment variables.

6.4 Assembler changes between ARM Compiler v4.1 for µVision and ARM Compiler v5.02 for µVision

There are no technical changes to `armasm` that affect migration between ARM Compiler v4.1 for µVision and v5.02 for µVision.

Chapter 7
Migrating from RVCT v4.0 for µVision to ARM Compiler v4.1 for µVision

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

It contains the following sections:

7.1 General changes between RVCT v4.0 for µVision and ARM Compiler v4.1 for µVision

Various general changes have been made between RVCT v4.0 for µVision and ARM Compiler v4.1 for µVision.

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 information

Toolchain environment variables.

7.2 Compiler changes between RVCT v4.0 for µVision and ARM Compiler v4.1 for µVision

Various changes have been made to `armcc` between RVCT v4.0 for µVision and ARM Compiler v4.1 for µVision.

The following changes to `armcc` have been made:

Sign rules on enumerators have changed in line with convention. The enumerator container is now unsigned unless a negative constant is defined.

`-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.

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 references

7.1 General changes between RVCT v4.0 for µVision and ARM Compiler v4.1 for µVision on page 7-34.

Related information

--multifile, --no_multifile (armcc).

-Onum.

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

7.3 Linker changes between RVCT v4.0 for µVision and ARM Compiler v4.1 for µVision

Various changes have been made to `armlink` between RVCT v4.0 for µVision and ARM Compiler v4.1 for µVision.

The following changes to `armlink` have been made:

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

7.1 General changes between RVCT v4.0 for µVision and ARM Compiler v4.1 for µVision on page 7-34.

Related information

About the ARM Compiler toolchain.

7.4 Assembler changes between RVCT v4.0 for µVision and ARM Compiler v4.1 for µVision

Various changes have been made to `armasm` between RVCT v4.0 for µVision and ARM Compiler v4.1 for µVision.

The following changes to `armasm` have been made:

Change to the way the assembler reads and processes files

Older assemblers sometimes permitted the source file being assembled to vary 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.

```
        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 following code shows another example where the assembler faults:

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

The resulting error is:

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

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

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

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

Various instructions in ARM (using SP) were deprecated when 32-bit Thumb instructions were introduced. These instructions are no longer diagnosed as deprecated unless assembling for a CPU that has 32-bit Thumb instructions. To enable the warnings on earlier CPUs, you can use the option `--diag_warning=1745,1786,1788,1789,1892`.

Obsolete command-line option

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

Related references

7.1 General changes between RVCT v4.0 for µVision and ARM Compiler v4.1 for µVision on page 7-34.

Related information

--diag_warning=tag{, tag} (armasm).

-o filename.

How the assembler works.
2 pass assembler diagnostics.

7.5 C and C++ library changes between RVCT v4.0 for µVision and ARM Compiler v4.1 for µVision

Various changes have been made to the ARM C and C++ libraries between RVCT v4.0 for µVision and ARM Compiler v4.1 for µVision.

The libraries now use more Thumb2 code on targets that support Thumb2. 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
```

You can enable the newer behavior with:

```
#pragma import __use_c99_matherr
```

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.

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 references

7.1 General changes between RVCT v4.0 for µVision and ARM Compiler v4.1 for µVision on page 7-34.

Related information

How the ARM C library fulfills ISO C specification requirements.
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.
--apcs=qualifer...qualifier (armcc).
--fpu=name (armcc).

7.6 fromelf changes between RVCT v4.0 for µVision and ARM Compiler v4.1 for µVision

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

Related information

input_file.

Chapter 8
Migrating from RVCT v3.1 for µVision to RVCT v4.0 for µVision

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

It contains the following sections:

8.1 General changes between RVCT v3.1 for µVision and RVCT v4.0 for µVision

Various general changes have been made between RVCT v3.1 for µVision and RVCT v4.0 for µVision.

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. 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.

Related information

--cpu=name (armcc).

--fpu=name (armcc).

--cpu=name (armlink).

--fpu=name (armlink).

--cpu=name (armasm).

--fpu=name (armasm).

--unsafe (armasm).

8.2 Changes to symbol visibility between RVCT v3.1 for µVision and RVCT v4.0 for µVision

Changes to symbol visibility have been made between RVCT v3.1 for µVision and RVCT v4.0 for µVision

The following changes to symbol visibility have been made:

Change to ELF visibility used to represent __declspec(dllexport)

When using the `--hide_all` compiler command-line option, which is the default, the ELF visibility used to represent `__declspec(dllexport)` in RVCT v3.1 for µVision and earlier was STV_DEFAULT. In RVCT v4.0 for µVision 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 v3.1for µVision.

__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 for µVision symbol visibility summary

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

Table 8-1 RVCT v3.1 for µVision 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 8-2 RVCT v3.1 for µVision 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 for µVision symbol visibility summary

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

Table 8-3 RVCT v4.0 for µVision 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 8-4 RVCT v4.0 for µVision 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

8.1 General changes between RVCT v3.1 for µVision and RVCT v4.0 for µVision on page 8-42.

8.3 Compiler changes between RVCT v3.1 for µVision and RVCT v4.0 for µVision

Various changes have been made to `armcc` between RVCT v3.1 for µVision and RVCT v4.0 for µVision.

The following changes to `armcc` 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.

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 for µVision 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 for µVision, 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

8.1 General changes between RVCT v3.1 for µVision and RVCT v4.0 for µVision on page 8-42.

Related information

--cpp (armcc).

--reassociate_saturation, --no_reassociate_saturation (armcc).

--thumb (armcc).

8.4 Linker changes between RVCT v3.1 for µVision and RVCT v4.0 for µVision

Various changes have been made to `armlink` between RVCT v3.1 for µVision and RVCT v4.0 for µVision.

The following changes to `armlink` have been made:

Placing ARM library helper functions with scatter files

In RVCT v3.1 for µVision 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 for µVision 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.

Linker steering files and symbol visibility

In RVCT v3.1 for µVision 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 for µVision 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.

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$$$$Base`, the linker now supports the following linker-defined symbols:

```
Load$$region_name$$Base
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 for µVision 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.

```
Load$$LR$$Load_Region_Name$$Base   ; Base address of <Load Region Name>
Load$$LR$$Load_Region_Name$$Limit  ; Load address of last byte of content in Load
region.
Load$$LR$$Load_Region_Name$$Length ; Limit - Base
```

Image-related symbols

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

They are defined only when scatter files are not used.

```
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 for µVision 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-R4F` implies `--fpu=vfpv3_d16`. In RVCT v3.1 for µVision the `--cpu` option only checked the build attributes of the chosen CPU.

The error message `L6463E: 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.

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 for µVision 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 used to be 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 for µVision.

Related references

8.1 General changes between RVCT v3.1 for µVision and RVCT v4.0 for µVision on page 8-42.

Related information

Optimization with RW data compression.
Linker-defined symbols.
Region-related symbols.
Image$$ execution region symbols.
Load$$ execution region symbols.
Methods of 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.
--cpu=name (armlink).
--force_explicit_attr (armlink).
--fpu=name (armlink).
--map, --no_map (armlink).
--override_visibility (armlink).
EXPORT (armlink).
IMPORT (armlink).
Execution region attributes.

8.5 Assembler changes between RVCT v3.1 for µVision and RVCT v4.0 for µVision

Various changes have been made to `armasm` between RVCT v3.1 for µVision and RVCT v4.0 for µVision.

The following changes to `armasm` 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 `LDM` instruction and no writeback was performed. Because the syntax requests writeback, and this encoding is only available in Thumb-2, it produces an error. To get the 16-bit 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

8.1 General changes between RVCT v3.1 for µVision and RVCT v4.0 for µVision on page 8-42.

Related information

Addition, subtraction, and logical operators.

--depend=dependfile (armasm).

-o filename.

8.6 fromelf changes between RVCT v3.1 for µVision and RVCT v4.0 for µVision

Various changes have been made to `fromelf` between RVCT v3.1 for µVision and RVCT v4.0 for µVision.

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

8.1 General changes between RVCT v3.1 for µVision and RVCT v4.0 for µVision on page 8-42.

Related information

--text (fromelf).

8.7 C and C++ library changes between RVCT v3.1 for µVision and RVCT v4.0 for µVision

Various changes have been made to the ARM C and C++ libraries between RVCT v3.1 for µVision and RVCT v4.0 for µVision.

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 for µVision.

Related references

8.1 General changes between RVCT v3.1 for µVision and RVCT v4.0 for µVision on page 8-42.