Migrate ARM Compiler 5 to ARM Compiler 6

MDK Tutorial

ARMKEIL Microcontroller Tools

AN298, March 2017, V 1.0

feedback@keil.com

Abstract

ARM® Compiler 6 is the compilation toolchain for the ARM architecture, available in MDK Version 5.

ARM Compiler 6 brings together the modern LLVM compiler infrastructure and the highly optimized ARM C libraries to produce performance and power optimized embedded software for the ARM architecture. Efficient code generation, better diagnostics, and faster feature development.

Since ARM Compiler 6 is based on a new technology it is only partially compatible with previous ARM Compiler Versions (5 and before). This application note guides to port your application source code, compiler settings and make best use of new compiler optimizations and diagnosis facilities.

Contents

Introduction

ARM® Compiler 6.6 has many advantages. Some highlights are:

- Best-In-Class code size for ARMv7-M (Cortex-M3 to Cortex-M7)
- Complete support for ARMv8-M (Cortex-M23 and Cortex-M33)
- Latest C++ standards supported (C++14)
- GCC Compatibility

Next ARM Compiler 6 version (introduced in May 2017) additionally offers:

- Improved Code-Size for ARMv6M (Cortex-M0 / M0+)
- Functional Safety Certification Package

Prerequisites

To use ARM Compiler 6 it is recommended to use the following MDK version at least:

- MDK version 5.23 or higher with ARM Compiler 6.6

MDK 5.23 is shipped with ARM Compiler 5.06 and ARM Compiler 6.6. Both are installed alongside on your hard disk.

The following software packs are the minimum versions supporting ARM Compiler 6:

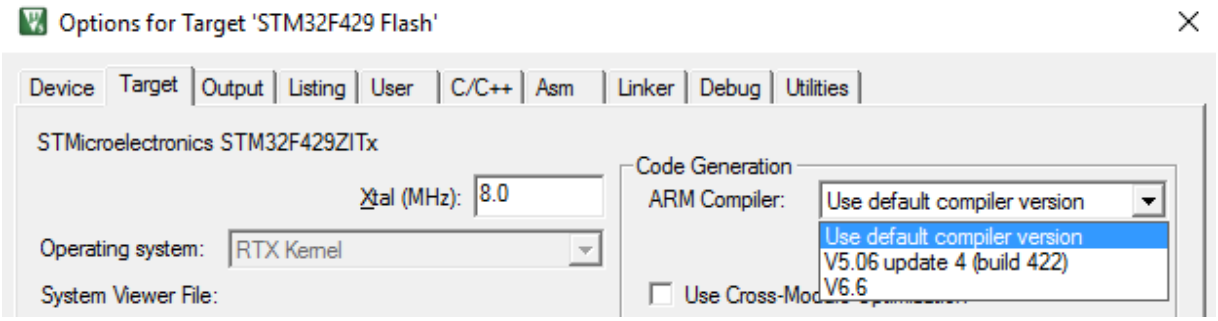
- Keil MDK-Middleware Pack Version **7.4.0** or higher
- Keil ARM Compiler Support Pack Version **1.3.0** or higher
- ARM CMSIS Pack Version **5.0.1** or higher

The above versions will already be installed with MDK version 5.23 by default.

For Device Family Packs (DFP) please contact the vendor of the pack for information which versions are compatible with ARM Compiler 6.

Switch the Compiler in µVision IDE

µVision provides a convenient interface to control all versions of ARM C/C++ Compilers.



Select Compiler Version in Options for Target

Before you begin it is recommended that you create a backup copy of your project.

1. Open your project in uVision.
2. Choose Project – Options for Target from the menu.

   On the Target tab you will find the setting for the used compiler version.

3. Set the ARM Compiler to Version 6.6 (or any newer version that your MDK release includes)
4. Confirm the dialog by pressing OK.

   Before this step you cannot make changes to compiler options.

Because there is no equivalent for most ARM Compiler 5 settings all compiler settings will be set to default. This document will discuss all relevant settings to help to identify the correct alternatives.

Diagnostic Settings: Warnings and Errors

ARM Compiler 6 offers a wide range of warning messages. These messages are typically very useful to point out potential issues with your code, which allows you to improve source quality.

Since not all warnings are required at all stages of the development the uVision IDE offers 4 levels of warnings.

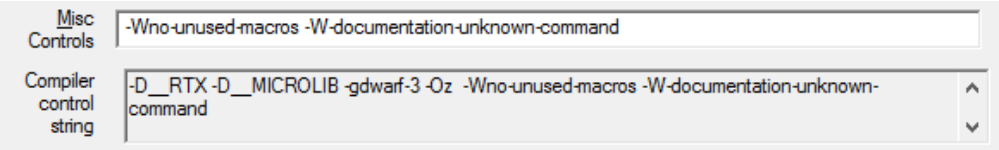
Level	Description
No Warnings	This is useful when focus is on error messages. No warning messages are generated.
All Warnings	All possible warnings are generated. Choose this at a later stage of the development to further improve code quality.
Moderate Warnings	This is comparable to the normal warning level of ARM Compiler 5. Shows mainly critical warnings and suppresses many less important warnings.
<unspecified>	No warning settings are passed to the compiler. This allows to set customized warning options using the Misc Control field.

For a complete reference of all available warning options you can refer to the LLVM clang documentation:
http://clang.llvm.org/docs/DiagnosticsReference.html

When a warning is issued the compiler always reports which diagnostic parameter is related.

```
Build Output
void Thread_LED (void const *argument) {
                           ^
Thread_LED.c(21): warning: function 'Thread_LED' could be declared with attribute 'noreturn' [-Wmissing-noreturn]
```

You can disable warnings of a certain diagnostic group by prefixing the parameter with `-Wno-` .
For example the `-Wmissing-noreturn` is disabled by the option `-Wno-missing-noreturn`.



Use Misc Controls to add or disable custom warnings to the Compiler Controls String

In a first step of migration it is recommended to switch the level to “No Warnings”. This will allow you to focus on error messages. Error messages are likely to be caused when ARM Compiler 5 language extensions have been used.

Incompatible Language Extensions

Language extension of ARM Compiler 5 and 6 are mainly incompatible.

It is recommended were possible to switch to the CMSIS abstraction of keywords. This guarantees code portability to all CMSIS supported compilers.

ARM Compiler 5	ARM Compiler 6	CMSIS (Recommended)
__align(x)	**__attribute__((aligned(x)))**	**__ALIGNED(x)**
__alignof__	**__alignof__**	
__ALIGNOF__	**__alignof__**	
__asm	See Assembler Migration	**__ASM**
__const	**__attribute__((const))**	
__forceinline	**__attribute__((always_inline))**	
__global_reg	Not supported	
__inline	**__inline__**. The use of this depends on the language mode.	**__INLINE** **__STATIC_INLINE**
__int64	No equivalent. Use **long long**. When you use long long in C90 mode, the compiler gives a warning.	**int64_t**
__irq	**__attribute__((interrupt))**	*Not required for Cortex-M ISRs*
__packed **__packed x struct**	**__attribute__((packed))** **struct x attribute((packed))**	**__PACKED** **__PACKED_STRUCT x**
__pure	__attribute__((const))	
__smc	Use inline assembler instructions or equivalent routine.	
__softfp	__attribute__((pcs("aapcs")))	
__svc	Use inline assembler instructions or equivalent routine.	
__svc_indirect	Use inline assembler instructions or equivalent routine.	
__thread	__thread	
__value_in_regs	__attribute__((value_in_regs))	
__weak	__attribute__((weak))	**__WEAK**
__writeonly	Not supported	

Hint: You might want to use the find feature of the uVision IDE to spot all occurences of above keywords:

Ctrl+F opens the Find dialog

See also the full documentation for CMSIS on www.keil.com/cmsis - Core – Reference

And check the ARM Infocenter - infocenter.arm.com - for Chapter 3 / Compiler Source Code Compatibility of the ARM® Compiler Migration and Compatibility Guide

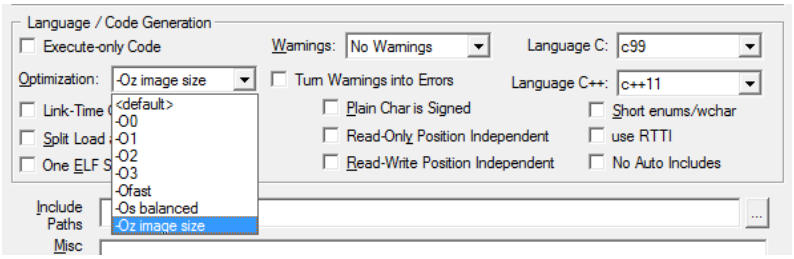
Select a Compiler Optimization Level

ARM Compiler 5 offered four different optimization levels 0 to 3. These behave differently than the levels from ARM Compiler 6. There is no direct conversion from the old to the new levels.

Use the below table to choose the optimization that best fits your requirements:

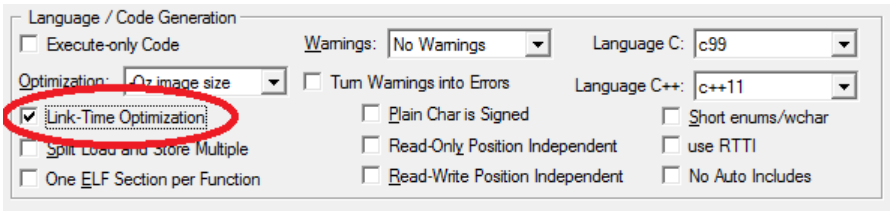
Level	Description
-O0	No Optimization. Not recommended for use in ARM Compiler 6.6
-O1	Limited Optimization. This is currently the recommended level for source level debugging.
-O2	Optimization for speed. Code size will increase due to many loop-unrolls and function inlining.
-O3	Optimization for speed. Faster, but larger code than –O2 produces
-Os	Balanced Optimization. Optimizes for speed were code size increase is reasonable.
-Oz	Purely optimize for code size.

A typical approach to start with is to compile on all levels and identify the result that makes best use of your memory resources. This is almost always the best choice for performance as well.



Setting the Compiler Optimization in the Options for Target Dialog

The most advanced optimization will be performed with the additional option “Link-Time Optimization”.



Activate the Link-Time Optimization for Level Oz for smallest image size.

Floating Point Optimization

Specifying a *fast-math* parameter in the “Misc Controls” for the Compiler controls the floating point behavior.

Default	IEEE 754 compliant with fixed rounding. Most common IEEE exceptions.
-ffast-math	Aggressive floating-point optimizations. Not IEEE compliant, but fastest math.
-fno-fast-math	IEEE compliant with configurable rounding, all IEEE exceptions.

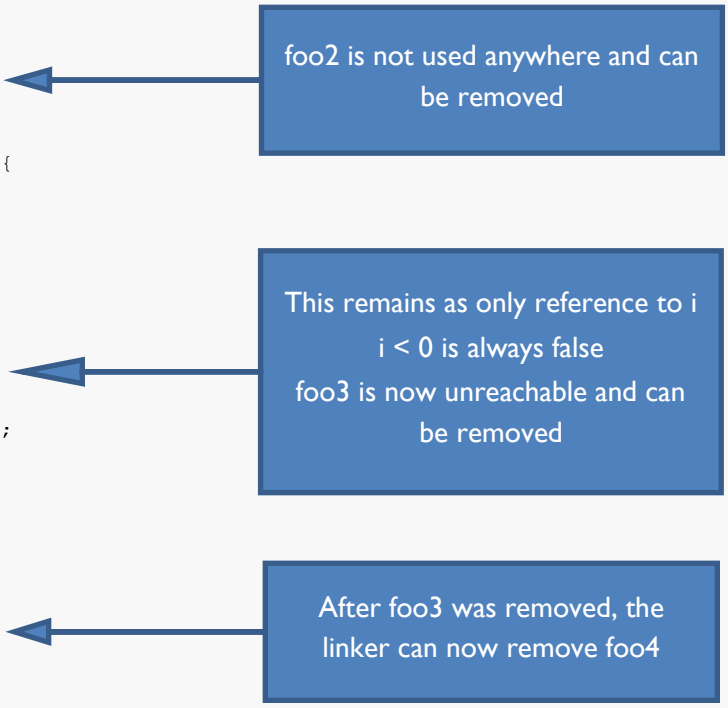
See also: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0774g/sam1433407361626.html

Link Time Optimization

One of the best optimizations to gain performance and maintain small code size is the so-called inter-modular optimization, also referred to as Link time optimization. In contrast to compiler optimizations it analyzes the whole program. Other optimizations are limited to single functions or single source modules.

Example

```
--- a.h ---
extern int foo1(void);
extern void foo2(void);
extern void foo4(void);

--- a.c ---
#include "a.h"

static int i = 0;

void foo2(void) {
  i = -1;
}

static int foo3() {
  foo4();
  return 10;
}

int foo1(void) {
  int data = 0;

  if (i < 0)
    data = foo3();

  data = data + 42;
  return data;
}

--- main.c ---
#include <stdio.h>
#include "a.h"

void foo4(void) {
  printf("Hi\n");
}

int main() {
  return foo1();
}
```



Avoid potential issues with LTO

Although the linker has almost full visibility there are some remaining blind spots. Especially when functions are called by interrupt vector or function pointers they do not always show up in the call tree analysis.

This can lead to functions or variables to be removed accidently.

Variables that are accessed from interrupt level should always be tagged with the keyword `volatile`.

Functions can be marked with the `used` attribute to prevent elimination. For example:

```
void foo2 (void) __attribute__((used)) {
  i = -1;
}
```

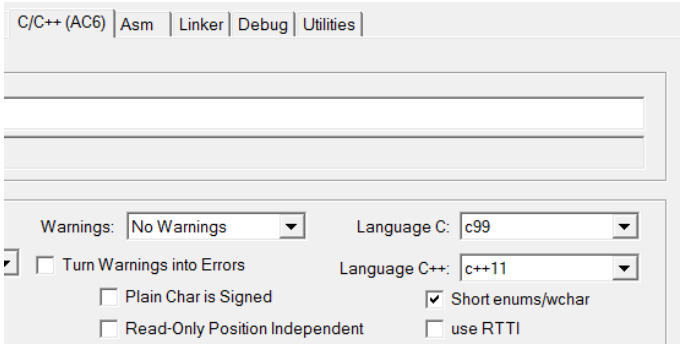
Also it is not advised to build libraries with the LTO optimization active. In LTO mode the compiler generates object files with an intermediate format called bitcode.

The bitcode file format is not guaranteed to be compatible across ARM Compiler 6 versions.

Object and Library Compatibility

When your project contains binary objects or libraries compiled with ARM Compiler 5 you can continue to use them. There are some minor compatibility issues with storage types.

By default ARM Compiler 5 stores enums and wide-characters in 16-bit values while ARM Compiler 6 stores them in 32-bit values. ARM Compiler 6 can be set to 16-bit enums/wchars. To achieve this the project needs to be build with two additional options: `-fshort-enums` & `-fshort-wchar`



Short enums/wchar enables the –fshort- options

If possible you should recompile your libraries with ARM Compiler 6. Functions from older libraries will not benefit of new compiler optimizations and also prevent some link-time optimizations across the application.

Data Packing

Packing data structures is to place members of the structure into memory without padding space. This typically leads to a situation where some structure members are no longer on natural alignment boundaries of the ARM architecture.

Access to these unaligned variables often comes with a strong penalty in form of extended code size and reduced execution performance. While migrating packed structures in your code to the new syntax it is advised to reevaluate the necessity for packing.

It has to be noted that the ARM Compiler 6 replacement __attribute__((packed)) provides a limited functionality compared to __packed.

The __attribute__((packed)) variable attribute applies to members of a structure or union, but it does not apply to variables that are not members of a struct or union.

The placement of the attribute is different from the placement of __packed. If your legacy code contains

`typedef __packed struct`, then replace it with:

`__PACKED_STRUCT` which is defined by the CMSIS Core.

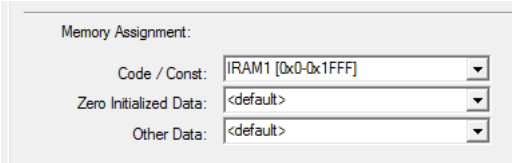
Linker: Scatter-Loading Descriptions

There are several issues that might occur when custom scatter files are used in your project to control the memory layout.

Specifying module names in segment descriptions with LTO

When LTO optimization is activated the names of modules will often not be detected by the linker. This is caused by the fact that the linker combines all modules into a single anonymous compilation unit first.

Also changing the memory assignment in the options of a source file in the IDE can result in this issue.



The code of a module is loaded to RAM at runtime.

The best solution to locate functions and variables in special memory segments is to name them in the source code and locate this named section.

Example: The following function prototype from a flash algorithm has to be executed in RAM:

```
int ProgramPage (uint32_t adr, uint32_t sz, uint8_t *buf)
{
  …
}
```

Use the section attribute to add the function to a section with a custom name:

```
__attribute__((section("myflash"))) int ProgramPage (uint32_t adr, uint32_t sz, uint8_t *buf)
{
  …
}
```

In the scatter description this section is located to a RAM segment:

```
  RW_IRAM1 0x20000000 0x00050000  {  ; RW data
   .ANY (+RW +ZI)
   *(name)      ; replaces the formerly used module.o(+RO)
  }
```

C Preprocessing used in Scatter file

If the scatter file contains preprocessing directives like *#define* and *#include* one of the first lines of the file will set up the interpreter. With ARM Compiler 5 this will typically look like:

```
#! armcc -E
```

Transform the line to use the ARM Compiler 6 executable „armclang“ instead:

```
#! armclang -E --target=arm-arm-none-eabi -mcpu=cortex-m7 -xc
```

The –mcpu parameter is mandatory for –E. You can adapt it to the target core used in your project.

Assembler Migration

ARM Compiler 6 changes the strategy of handling assembly code completely.

Assembly syntax is now compatible with the GNU style rather than the ARM style. Assembling is done by the compiler executable. There is no separate assembler executable.

Migrate ARM Syntax to GNU Syntax

The following statements are different between ARM and GNU syntax:

- Directives.
- Format of labels, comments, and some types of literals.
- Some symbol names.
- The operators.

The following examples show simple, equivalent, assembly code in both ARM and GNU syntax:

ARM syntax

```
; Simple ARM syntax example
; Iterate round a loop 10 times, adding 1 to a register each time.

        AREA ||.text||, CODE, READONLY, ALIGN=2

main PROC
        MOV      w5,#0x64       ; W5 = 100
        MOV      w4,#0          ; W4 = 0
        B        test_loop      ; branch to test_loop
loop
        ADD      w5,w5,#1       ; Add 1 to W5
        ADD      w4,w4,#1       ; Add 1 to W4
test_loop
        CMP      w4,#0xa        ; if W4 < 10, branch back to loop
        BLT      loop
        ENDP
        END
```

GNU syntax

```
// Simple GNU syntax example
// Iterate round a loop 10 times, adding 1 to a register each time.

        .section .text,"x"
        .balign 4

main:                           //
        MOV      w5,#0x64       // W5 = 100
        MOV      w4,#0          // W4 = 0
        B        test_loop      // branch to test_loop
loop:
        ADD      w5,w5,#1       // Add 1 to W5
        ADD      w4,w4,#1       // Add 1 to W4
test_loop:
        CMP      w4,#0xa        // if W4 < 10, branch back to loop
        BLT      loop
        .end                    //
```

Refer to the ARM Compiler 6 Reference Manual for a detailed description of differences.

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0742g/chr1398241769674.html

Assembler Modules

Current releases of ARM Compiler 6 are deployed with the legacy ARM Assembler (armasm.exe). This allows you to use existing assembler files untouched. The object files of the armasm can directly be used by the new linker. When you create new assembler modules it is recommended to create these with the GNU syntax.

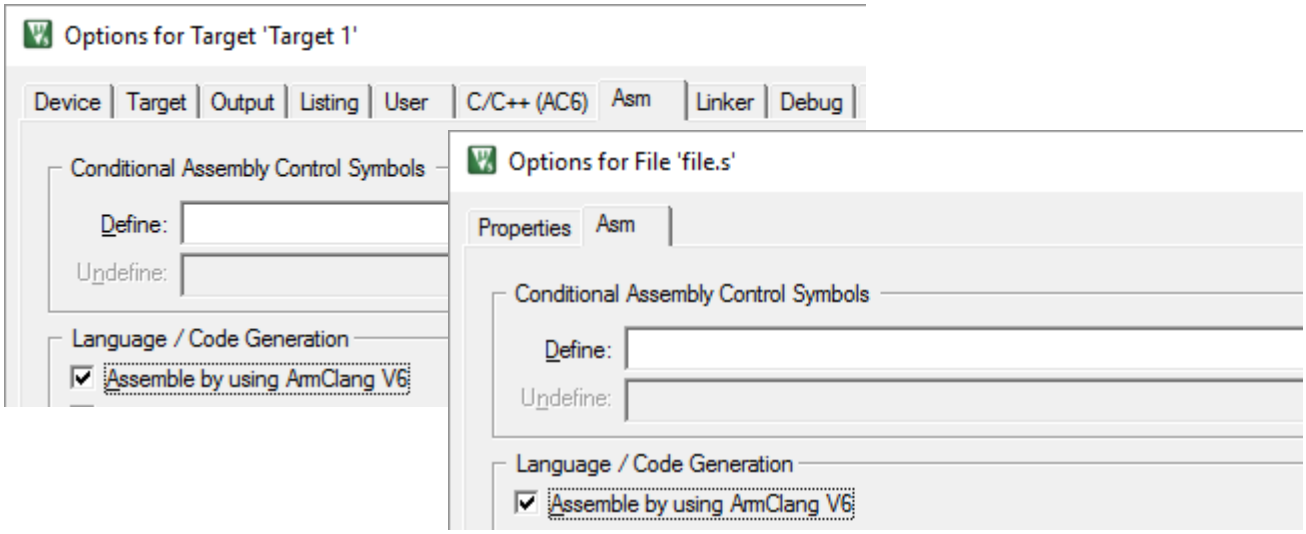
An example of invoking the legacy ARM Assembler:

```
armasm --cpu=Cortex-M4.fp -o file.o file.s
```

Assembling using the ARM Compiler 6 (accepts GNU syntax source only):

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m4.fp -c -o file.o file.s
```

The µVision IDE will handle the command line of the assembler of choice.
Select **"Assemble by using ArmClang"** V6 to use the new GNU syntax assembly globally or for each file:



Inline Assembler

The *__asm* keyword can incorporate inline GCC syntax assembly code in line with C source. ARM Compiler 6 only supports GNU style inline assembler statements.

The general form of an `asm` inline assembly statement is:

```
__asm(code [: output_operand_list [: input_operand_list [: clobber_list]]]);
```

code is the assembly code. For example: **`"ADD %[result], %[input_i]"`**

output_operand_list is an optional list of output operands, separated by commas. Each operand consists of a symbolic name in square brackets, a constraint string, and a C expression in parentheses. Example for a single output operand: **`[result] "=r" (res).`**

input_operand_list is an optional list of input operands, separated by commas. Input operands use the same syntax as output operands. E.g.: **`[input_i] "r"`**

clobber_list is an optional list of clobbered registers, or other values

See also for more details:

https://gcc.gnu.org/onlinedocs/gcc/Using-Assembly-Language-with-C.html#Using-Assembly-Language-with-C

Embedded Assembler

The embedded assembler that allowed to declare assembler language functions in a C source context is deprecated in its previous form.

Example of an Embedded Assembler strcpy function:

```
__asm int f(int i)
{
    ADD r0, r0, #1
}
```

Instead implement the function as a normal C function with Inline Assembly.

```
int f(int i)
{
    __asm("ADD r0, r0, #1");
}
```

Using the so called Extented Asm syntax defined by the GNU GCC you can still access symbols and labels declared in C language. See: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

Example of GNU Extended Asm syntax accessing a variable from the C context:

```
unsigned long Swap(unsigned long value)
{
__asm  (
        "EOR     r3, %1, %1, ROR #16\n\t"
        "BIC     r3, r3, #0x00FF0000\n\t"
        "MOV     %0, %1, ROR #8\n\t"
        "EOR     %0, %0, r3, LSR #8"
        : "=r" (value)
        : "0"(value)
        : "r3"
);
return value;
}
```

Other Implementation Specific Differences

Some important differences in implementation defined handling are outlined below. Check you source for the occurrence of these.

	ARM Compiler 5	**ARM Compiler 6**
Integer Division	Remainder and numerator sign match (checked)	Remainder and Numerator may not match
Floating Point	IEEE754, rounding to nearest representation by default No exceptions	Facilities, Operations and Representation as of IEEE by default (*ARM Compiler 5: --fpmode=ieee_full*)
Enum Packing	Smallest integer type suitable (8 to 64 bit)	at least int, max long long (32 to 64 bit)
Bit-field Signedness	default unsigned	default signed
sizeof(wchar_t)	2 Bytes	4 Bytes

Conclusion

The migration to ARM Compiler 6 is feasible with the assistance of uVision and the documentation provided. Although there is some effort involved, especially when converting language extensions, the result is mostly smaller code size and better performance.