Getting the Best Optimized Code for your Embedded Application

Abstract

This document examines the ARM Compilation Tools, as used inside the Keil MDK-ARM (Microcontroller Development Kit), and how to use them to optimize your code for best performance or smallest code-size.

Contents

Revision History

- August 2009: Initial Version

ARM Compilation Tools

The ARM Compilation Tools are the only compilation tools co-developed with the ARM processors, and specifically designed to optimally support the ARM architecture. They are a result of 20 years of development, and are recognized as the industry-leading C and C++ compilation tools for the ARM, Thumb, and Thumb-2 instructions sets.

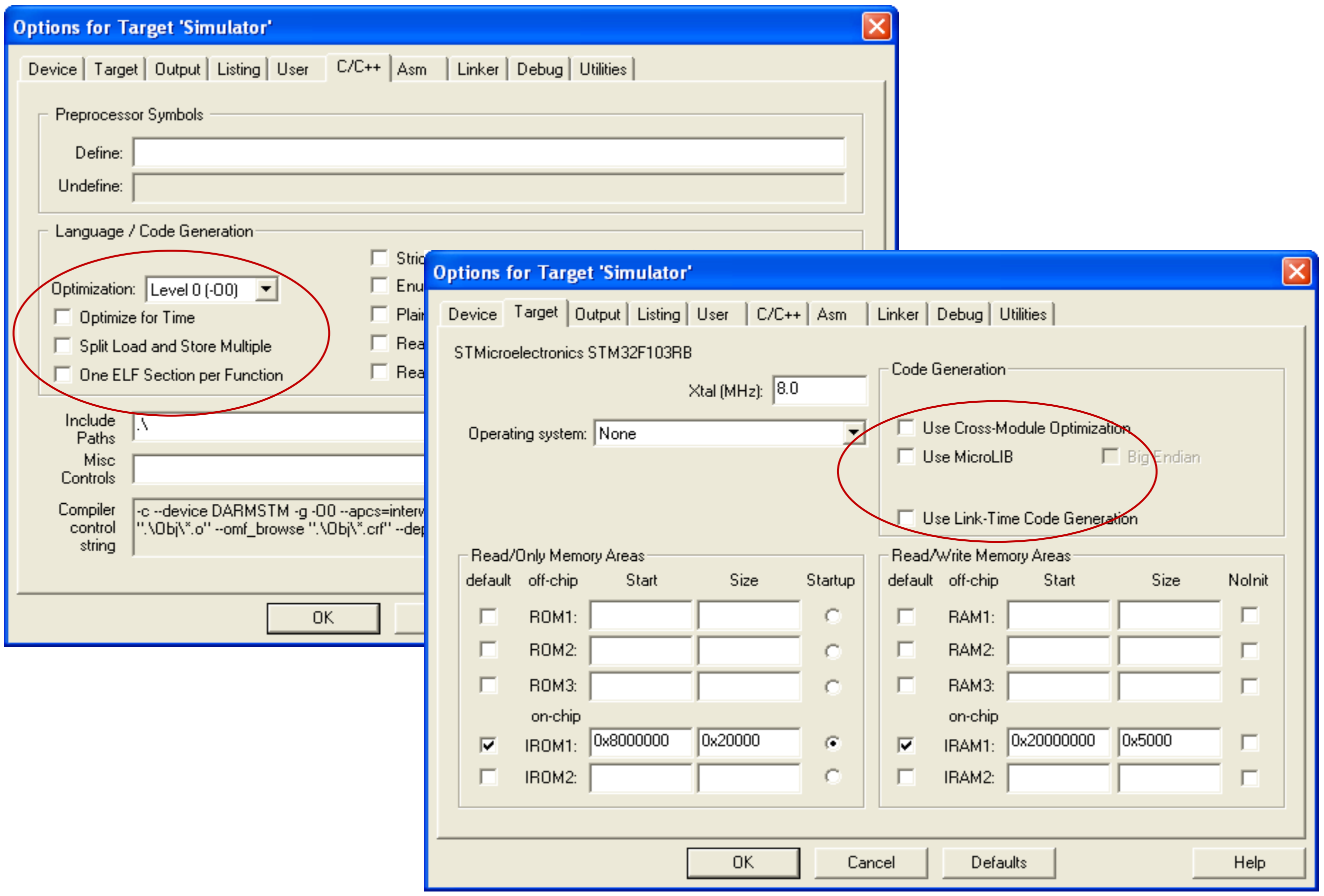
The ARM Compilation tools consist of:

- *The ARM Compiler, which enables you to compile C and C++ code. It is an optimizing compiler, and features command-line options to enable you to control the level of optimization*
- *Linker and Utilities, which assign addresses and lay out sections of code to form a final image*
- *A selection of libraries, including the ISO standard C libraries, and the MicroLIB C library which is optimized for embedded applications*
- *Assembler, which generates machine code instructions from ARM, Thumb or Thumb-2 assembly-level source code*

Compiler Options for Embedded Applications

The ARM Compilation Tools include a number of compiler optimizations to help you best target your code for your chosen microcontroller device and application area.

They can be accessed from within µVision by clicking on Project – Options for Target.
The options described this document can be found on the **Target** and **C/C++** tabs of the **Options for Targets** dialog.



Preprocessor Symbols
Define:
Undefine:

Language / Code Generation
Optimization: Level 0 (-O0)
☐ Optimize for Time
☐ Split Load and Store Multiple
☐ One ELF Section per Function
☐ Stric
☐ Enu
☐ Plai
☐ Rea
☐ Rea

Include Paths: .\
Misc Controls:
Compiler control string: -c --device DARMSTM -g -O0 --apcs=interw ".\Obj*.o" --omf_browse ".\Obj*.crf" --dep

STMicroelectronics STM32F103RB
Xtal (MHz): 8.0
Operating system: None

Code Generation
☐ Use Cross-Module Optimization
☐ Use MicroLIB ☐ Big Endian
☐ Use Link-Time Code Generation

Read/Only Memory Areas

default	off-chip	Start	Size	Startup
☐	ROM1:			○
☐	ROM2:			○
☐	ROM3:			○
	on-chip			
☑	IROM1:	0x8000000	0x20000	●
☐	IROM2:			○

Read/Write Memory Areas

default	off-chip	Start	Size	NoInit
☐	RAM1:			☐
☐	RAM2:			☐
☐	RAM3:			☐
	on-chip			
☑	IRAM1:	0x20000000	0x5000	☐
☐	IRAM2:			☐

- **Cross-Module Optimization** takes information from a prior build and uses it to place UNUSED functions into their own ELF section in the corresponding object file. This option is also known as Linker Feedback, and requires you to build your application twice to take advantage of it for reduced code size.

  Cross-Module Optimization has been shown to reduce code size, by removing unused functions from your application. It can also improve the performance of your application, by allowing modules to share inline code.

- The **MicroLIB** C library has been optimized to reduce the size of embedded applications. It is a subset of the ISO standard C runtime library, and offers a tradeoff between functionality and code size. Some of the standard C library functions such as *memcpy()* are slower, while some features of the default library are not supported. Unsupported features include:

  - Operating system functions e.g. abort*(), exit(), time(), system(), getenv(),*
  - Wide character and multi-byte support e.g. *mbtowc(), wctomb()*
  - The *stdio* file I/O function, with the exception of *stdin, stdout* and *stderr*
  - Position-independent and thread-safe code

  Use the MicroLIB C library for applications where overall performance can be traded off against the need to reduce code size and memory cost.

- **Link-Time Code Generation** instructs the compiler to create objects in an intermediate format so that the linker can perform further code optimizations. This gives the code generator visibility into cross-file dependencies of all objects simultaneously, allowing it to apply a higher level of optimizations. Link-time code generation can reduce code size, and allow your application to run faster.

- **Optimization Levels** can also be adjusted. The different levels of optimization allow you to trade off between the level of debug information available in the compiled code, and the performance of the code. The following optimization levels are available:

  - **-O0** applies minimum optimizations.
    Most optimizations are switched off, and the code generated has the best debug view.
  - **-O1** applies restricted optimization.
    For example, unused inline functions and unused static functions are removed. At this level of optimization, the compiler also applies automatic optimizations such as removing redundant code and re-ordering instructions so as to avoid an interlock situation. The code generated is reasonably optimized, with a good debug view.
  - **-O2** applies high optimization (This is the default setting).
    Optimizations applied at this level take advantage of ARM’s in-depth knowledge of the processor architecture, to exploit processor-specific behavior of the given target. It generates well optimized code, but with limited debug view.
  - **-O3** applies the most aggressive optimization.
    The optimization is in accordance with the user’s –Ospace/-Otime choice. By default, multi-file compilation is enabled, which leads to a longer compile time, but gives the highest levels of optimization.

- The **Optimize for Time** checkbox causes the compiler to optimize with a greater focus on achieving the best performance when checked (-Otime) or the smallest code size when unchecked (-Ospace).

  Unchecking **Optimize for Time** selects the –Ospace option which instructs the compiler to perform optimizations to reduce the image size at the expense of a possible increase in execution time. For example, using out-of-line function calls instead of inline code for large structure copies. This is the default option. When running the compiler from the command line, this option is invoked using '-Ospace'

  Checking **Optimize for Time** selects the –Otime option which instructs the compiler to optimize the code for the fastest execution time, at the risk of an increase in the image size. It is recommended that you compile the time-critical parts of your code with –Otime, and the rest using the –Ospace directive.

- **Split Load and Store Multiples** instructs the compiler to split LDM and STM instructions involving a large number of registers into a series of loads/stores of fewer multiple registers. This means that an LDM of 16 registers can be split into 4 separate LDMs of 4 registers each. This option helps to reduce the interrupt latency on ARM systems which do not have a cache or write buffer, and systems which use zero-wait state 32-bit memory.

  For example, the ARM7 and ARM9 processors take can only take an exception on an instruction boundary. If an exception occurs at the start of an LDM of 16 registers in a cacheless ARM7/ARM9 system, the system will finish making 16 accesses to memory before taking the exception. Depending on the memory arbitration system, this can result in a very high interrupt latency. Breaking the LDM into 4 individual LDMs for 4 registers means that the processor will take the exception after loading a maximum of 4 registers, thereby greatly reducing the interrupt latency.

  Selecting this option improves the overall performance of the system.

- The **One ELF Section per Function** option tells the compiler to put all functions into their own individual ELF sections. This allows the linker to remove unused functions.

  An ELF code section typically contains the code for a number of functions. The linker is normally only able to remove unused ELF sections, not unused functions. An ELF section can only be removed if all its contents are unused. Therefore, splitting each function into its own ELF section allows the compiler to easily identify which ones are unused, and remove them.

  Selecting this option increases the time required to compile your code, but results in improved performance.

The combination of options applied will depend on your optimization goal –whether you are optimizing for smallest code size, or best performance.

The next section illustrates the best optimization options for each of these goals.

Optimizing for Smallest Code Size

To optimize your code for the smallest size, the best options to apply are:

- The MicroLIB C library
- Cross-module optimization
- Optimization level 2 (-O2)

Compile the Measure example without any optimizations

The Measure example uses analog and digital inputs to simulate a data logger.

File -- Open Project
C:\Keil\ARM\Boards\Keil\MCBSTM32\Measure\Measure.uv2

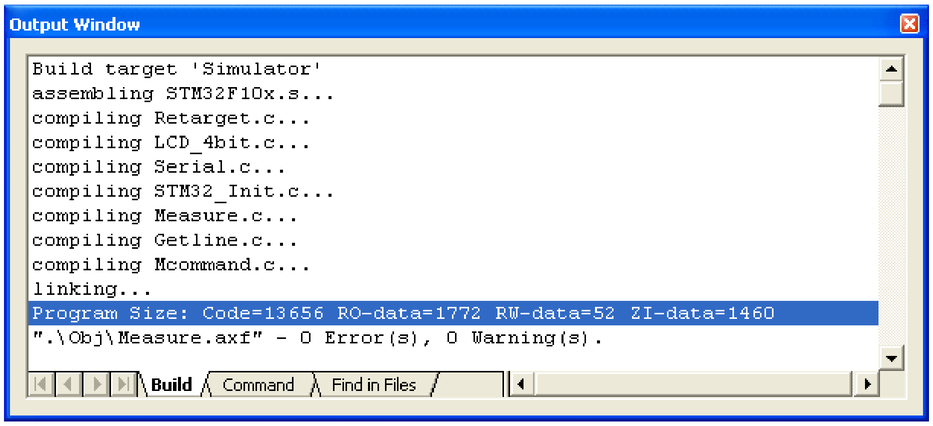
Click the **Options for Target** button

In the Target tab:
- Uncheck **Cross-Module Optimization**
- Uncheck **Use MicroLIB**
- Uncheck **Use Link-Time Code Generation**

In the C/C++ tab:
- Set Optimization Level to Zero

Then click **OK** to save your changes.

Project – Build target



```
Build target 'Simulator'
assembling STM32F10x.s...
compiling Retarget.c...
compiling LCD_4bit.c...
compiling Serial.c...
compiling STM32_Init.c...
compiling Measure.c...
compiling Getline.c...
compiling Mcommand.c...
linking...
Program Size: Code=13656 RO-data=1772 RW-data=52 ZI-data=1460
".\Obj\Measure.axf" - 0 Error(s), 0 Warning(s).
```

Without any compiler optimizations applied, the **initial code size is 13,656 Bytes**.

Optimize the Measure example for Size

Apply the compiler optimizations in turn, and re-compile each time to see their effect in reducing the code size for the example.

- Options for Target – Target tab: Use the MicroLIB C library
- Options for Target – Target tab: Use cross-module optimization - Remember to compile twice
- Options for Target – C/C++ tab: Enable Optimization level 2 (-O2)

Optimization Applied	Compile Size	Size Reduction	Improvement
MicroLIB C library	8,960 Bytes	4,696 Bytes	34% smaller
Cross-Module Compilation	13,500 Bytes	156 Bytes	1.1% smaller
Optimization level –O2	12,936 Bytes	720 Bytes	5.3% smaller
All 3 optimization options	8,116 Bytes	5,540 Bytes	40.6% smaller

Applying all the optimizations will reduce the code size down to 8,116 Bytes.

The fully optimized code is 5,540 Bytes smaller, a total code size reduction of 40.6%

Optimizing for Best Performance

To optimize your code for performance, the best options to apply are:

- Cross-module optimization
- Optimization level 3 (-O3)
- Optimize for time

Run the Dhrystone benchmark without any optimizations

The Dhrystone benchmark is used to measure and compare the performance of different computers, or the efficiency of the code generated for the same computer by different compilers.

File – Open Project
C:\Keil\ARM\Examples\DHRY\DHRY.uv2

Click the **Options for Target** button
Turn off optimization settings in the Target and C/C++ tabs, then click **OK**

Project – Build target

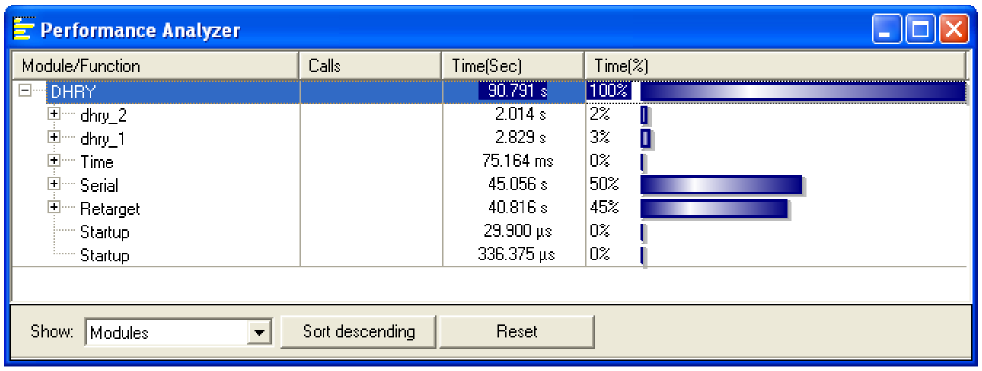
Enter Debug mode

View – Serial Windows – UART #1
Open the **UART #1** window

View – Analysis Windows – Performance Analyzer
Open the **Performance Analyzer**

Debug – Run
Start running the application

When prompted:
Enter **50000** in the **UART#1** window and press Enter



Module/Function	Calls	Time(Sec)	Time(%)
DHRY		90.791 s	100%
dhry_2		2.014 s	2%
dhry_1		2.829 s	3%
Time		75.164 ms	0%
Serial		45.056 s	50%
Retarget		40.816 s	45%
Startup		29.900 µs	0%
Startup		336.375 µs	0%

In the Performance Analyzer window, note that

- The drhy_1 loop took 2.829s
- The dhry_2 took 2.014s

In the UART #1 window, note that

- It took 138.0 ms for 1 run through Dhrystone
- The application is executing 7246.4 Dhrystones per second

Optimize the Dhrystone example for Performance

Re-compile the example with all three of the following optimizations applied:

- Options for Target – Target tab: Cross-module optimization – Remember to compile twice
- Options for Target – C/C++ tab: Optimization level 3 (-O3)
- Options for Target – C/C++ tab: Optimize for Time

Re-run the application, and examine the performance.

Measurement	Without optimizations	With Optimizations	Improvement
dhry_1	2.829s	1.695s	40.1% faster
dhry_2	2.014s	1.011s	49.8% faster
Microseconds for 1 run through Dhrystone	138.0	70	49.3% faster
Dhrystones per second	7246.4	14,285.7	97.1% more

The fully optimized code achieves approximately **2x the performance** of the un-optimized code.

Summary

The ARM Compilation Tools offer a range of options to apply when compiling your code. These options can be combined to optimize your code for best performance, for smallest code size, or for any performance point between these two extremes, to best suit your targeted microcontroller device and market.

When optimizing your code, MDK-ARM makes it easy and convenient to measure the effect of the different optimization settings on your application. The code size is clearly displayed after compilation, and a range of analysis tools such as the Performance Analyzer enable you to measure performance.

The optimization options in the ARM Compilation Tools, together with the easy-to-use analysis tools in MDK-ARM, help you to easily optimize your application to meet your specific requirements.