CHAPTER 2

Developing Software for the Cortex-M Family

Introduction

One of the big advantages of using a Cortex-M processor is that it has wide development tool support. There are toolchains available from almost zero cost up to several thousand dollars depending on the depth of your pockets and the type of application you are developing. Today there are five main toolchains that are used for Cortex-M development.

Table 2.1: Cortex Processor Toolchains

1. GNU GCC
2. Greenhills
3. IAR embedded workbench for ARM
4. Keil microcontroller development kit for ARM (MDK-ARM)
5. Tasking VX toolset for ARM

Strictly speaking, the GNU GCC is a compiler linker toolchain and does not include an integrated development environment (IDE) or a debugger. A number of companies have created a toolchain around the GCC compiler by adding their own IDE and debugger to provide a complete development system. Some of these are listed in the appendix; there are quite a few, so this is not a complete list.

Keil Microcontroller Development Kit

The Keil MDK-ARM provides a complete development environment for all Cortex-M-based microcontrollers.

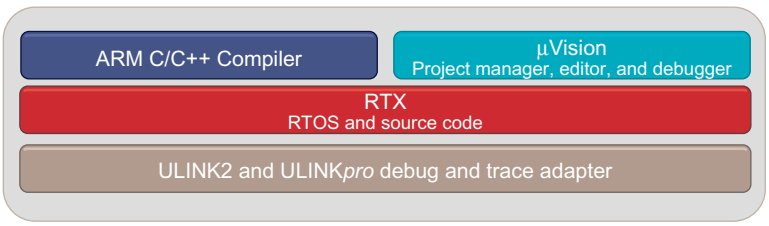


Figure 2.1
The MDK-ARM contains an IDE, compiler, RTOS, and debugger.

The MDK-ARM includes its own development environment called μVision (MicroVision), which acts as an editor, project manager, and debugger. One of the great strengths of the MDK-ARM is that it uses the ARM C compiler. This is a very widely used C\C++ compiler which has been continuously developed by ARM since the first CPUs were created. The MDK-ARM also includes an integrated RTOS called Real Time Executive or RTX. All of the Cortex-M processors are capable of running an OS and we will look at using an RTOS in Chapter 7. As well as including an RTOS, the MDK-ARM also includes a DSP library which can be used on the Cortex-M4 and Cortex-M3 processors. We will look at this library in Chapter 7.

The Tutorial Exercises

There are a couple of key reasons for using the MDK-ARM as the development environment for this book. First, it includes the ARM C compiler that is the industry reference compiler for ARM processors. Second, it includes a software simulator that models the Cortex-M processor and the peripherals of Cortex-M-based microcontrollers. This allows you to get most of the tutorial examples in this book without the need for a hardware debugger or evaluation board. The simulator is a very good way to learn how each Cortex-M processor works as you can get more detailed debug information from the simulator than from a hardware debugger. The MDK-ARM also includes the first RTOS to support the Cortex microcontroller software interface standard (CMSIS) RTOS specification. We will see more of this in Chapter 6, but it is basically a universal application programming interface (API) for Cortex-M RTOS. While we can use the simulator to experiment with the different Cortex-M processors, there comes a point when you will want to run your code on some real hardware. Now there are a number of very low-cost modules that include debugger hardware. The appendix provides URLs to web sites where these boards can be purchased.

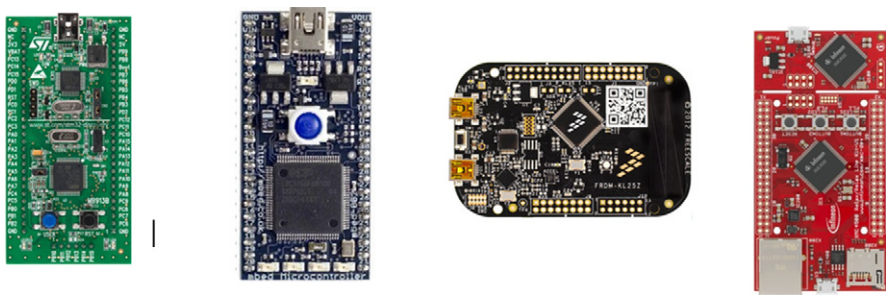


Figure 2.2
Low-cost Cortex-M modules include the STMicroelectronics discovery board, Freescale Freedom board, ARM (NXP) MBED module, and Infineon Technologies Relax board.

The tutorial exercises described in this book are geared toward using the software simulator to demonstrate the different features of the different Cortex-M processors. Matching example sets are also available to run on the different hardware modules where possible. If there are any differences between the instructions for the simulation exercise and the hardware exercise, an updated version of the exercise instructions is included as a PDF in the project directory.

Installation

For the practical exercises in this book, we need to install the MDK-ARM lite toolchain and the example set you plan to use.

First download the Keil MDK-ARM Lite Edition from www.keil.com.

The MDK-ARM Lite Edition is a free version of the full toolchain which allows you to build application code up to 32 K image size. It includes the fully working compiler, RTOS, and simulator and works with compatible hardware debuggers. The direct link for the download is https://www.keil.com/arm/demo/eval/arm.htm.

If the website changes and the link no longer works just the main Keil website at www.keil.com and follow the link to the MDK-ARM tools.

Run the downloaded executable to install the MDK-ARM onto your PC.

Download the example set for the practical exercises.

The different example sets can be found at http://booksite.elsevier.com/9780080982960.

If you do not have any hardware, then download the simulation example set. Otherwise, download the example set for the hardware module you are planning to use.

Once you have downloaded the example set that you are planning to use, run the installer and you are ready to go.

By default, the example set will be installed to C:\ Cortex_M_Tutorial_Exercises.

Exercise Building a First Program

Now that the toolchain and the example sets are installed, we can look at setting up a project for a typical small Cortex-M-based microcontroller. Once the project is set up, we can get familiar with the μVision IDE, build the code, and take our first steps with the debugger.

All of the example sets start by building this project for the simulator, and I recommend that you follow the first exercise described below. If you have a hardware module, a second example in this exercise is used to connect the μVision IDE to the hardware module.

The hardware-based example will have different application code to match the facilities available on the module.

The Blinky Project

In this example, we are going to build a simple project called *blinky*. The code in this project is designed to read a voltage using the microcontroller's ADC. The value obtained from the ADC is then displayed as a bar graph on a small LCD display.

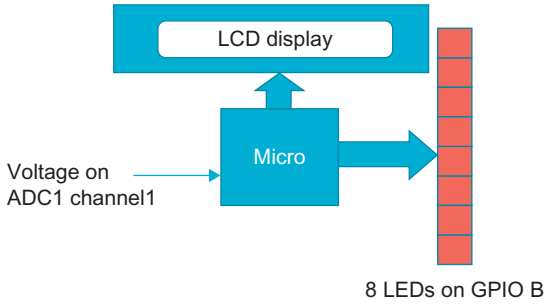
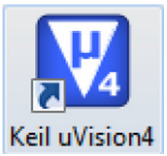


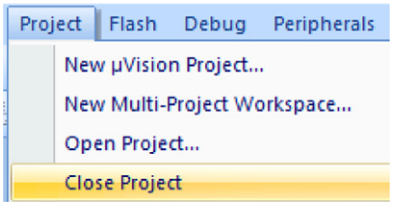
Figure 2.3
The blinky project hardware consists of an analog, voltage source, and external LCD display and a bank of LEDs.

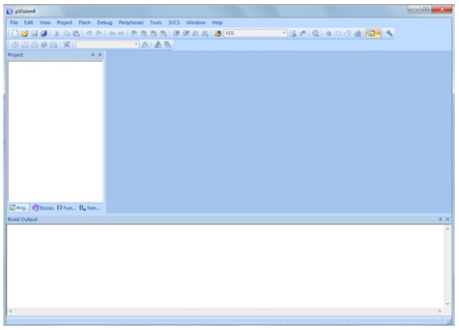
The code also flashes a group of LEDs in sequence. There are eight LEDs attached to port pins on GPIO port B. The speed at which the LEDs flash is set by the ADC value.

Start the µVision IDE by clicking on the UV4 icon.

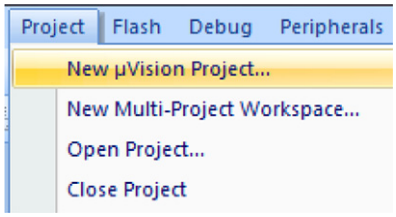


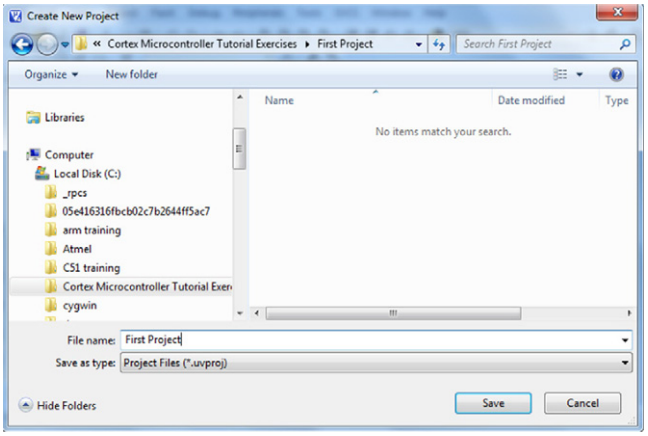
Once the IDE has launched, close any open project by selecting Project\Close Project from the main menu bar.





Start a new project by selecting Project\New μVision Project.





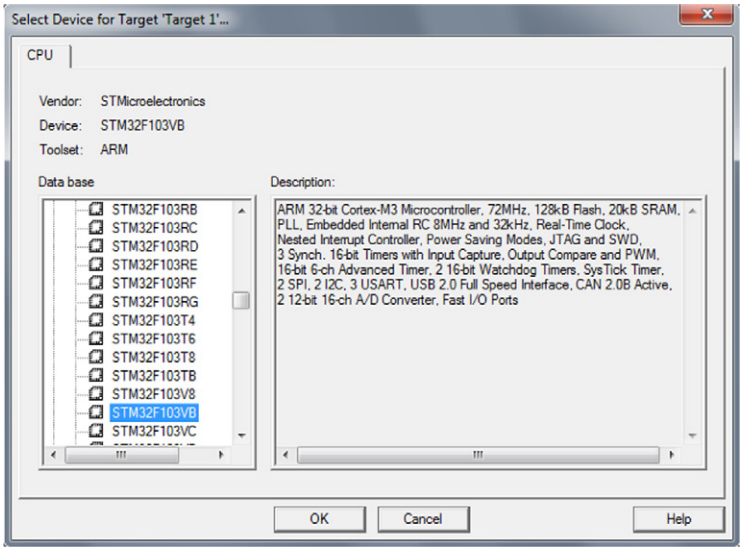
This will open a menu asking for a project name and directory.

You can give the project any name you want but make sure you select the Cortex-M tutorial\first project directory.

This directory contains the C source code files which we will use in our project.

Enter your project name and click save.

Next, select the microcontroller to use in the project.

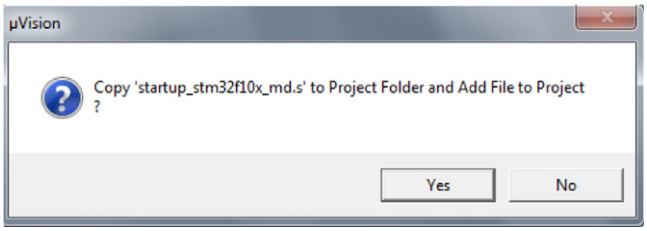


Once you have selected the project directory and saved the project name, a new dialog with a device database will be launched. Here, we must select the microcontroller that we are going to use for this project. Navigate the device database and select STMicroelectronics

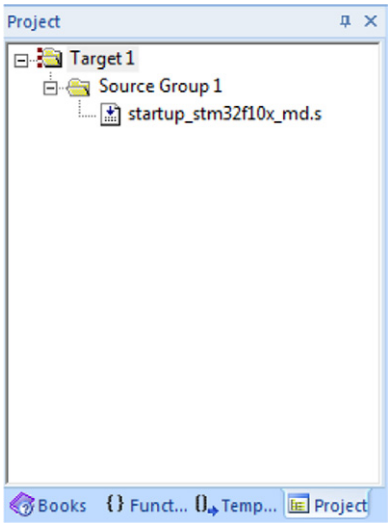
and the STM32F103RB and click OK. This will configure the project to use the STM32, which includes setting up the correct compiler options, linker script file, simulation model, debugger connection, and flash programming algorithms.

When you have selected the STM32F103RB click OK.

Then, click Yes to add the startup code to the project.

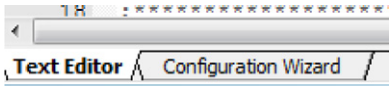


Once you have selected the device, the project wizard will ask if you want to add the STM32 startup code to the project, again say yes to this. The startup file provides the necessary code to initialize the C runtime environment and runs from the processor reset vector to the main() function in your application code. As we will see later, it provides the processor vector table, stack, and variable initialization. It does not provide any hardware initialization for user peripherals or the processor system peripherals. Once you have completed the processor wizard, your project window should have a target folder, and a source group folder. The startup file should be inside the source group folder as shown below.

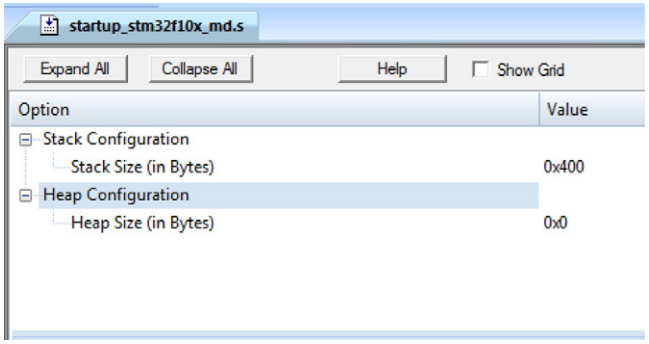


Double click on the startup_stm32F10x.md.s file to open it in the editor.

Click on the Configuration Wizard tab at the bottom of the editor window.

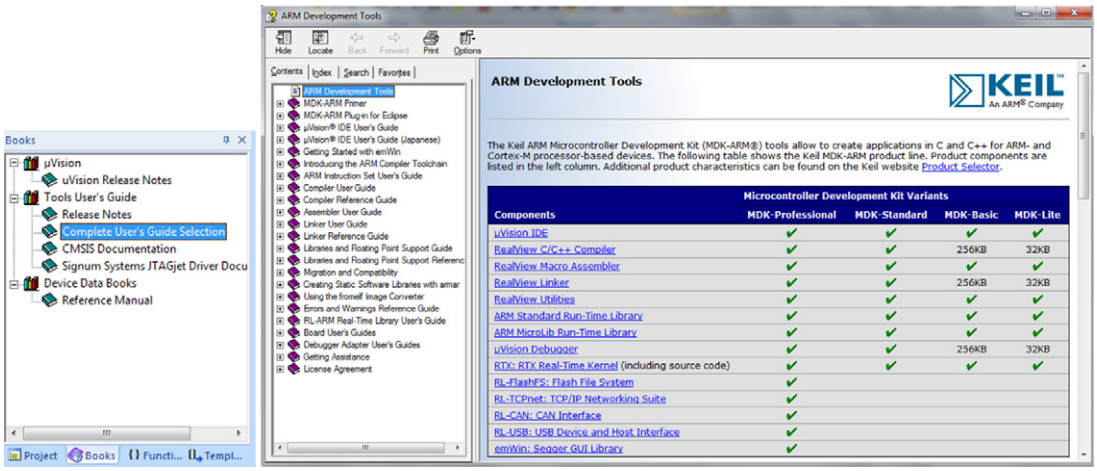


This converts the plain text source file to a view that shows the configuration options within the file.



This view is created by XML tags in the source file comments. Changing the values in the configuration wizard modifies the underlying source code. In this case, we can set the size of the stack space and the heap space.

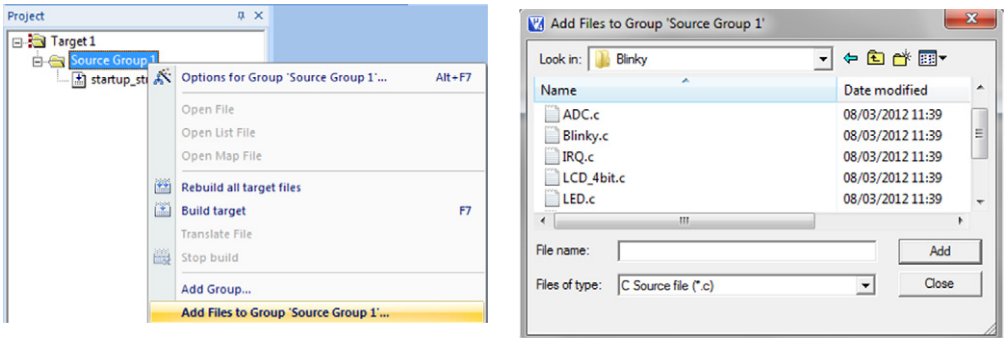
In the project view, click the Books tab at the bottom of the window.



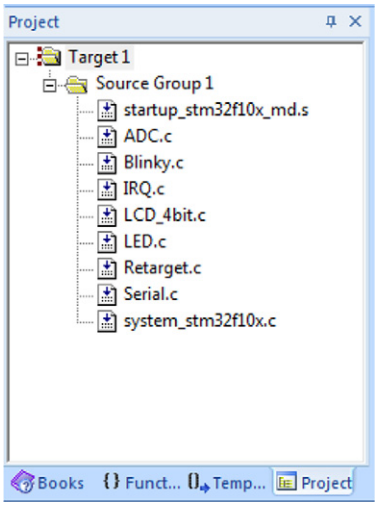
In the books window, the "Complete Users Guide Selection" opens the full help system.

Switch back to the project view and add the project C source files.

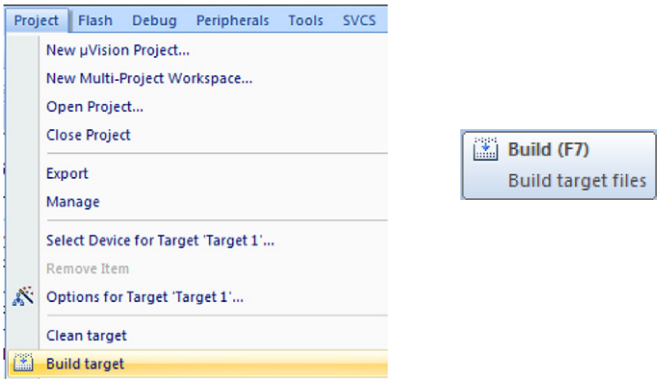
Highlight the source group folder in the project window, then right click and select "Add Files to Group Source Group 1."



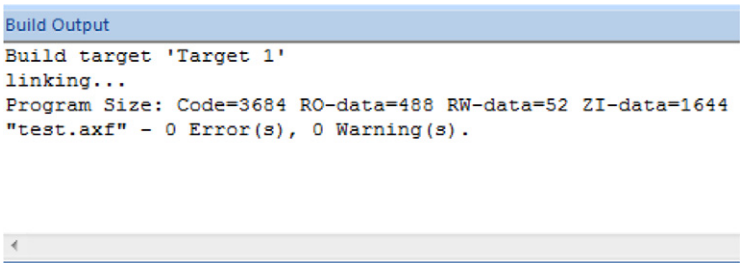
This will open an "Add Files to Group" dialog, which contains the eight ".c" files that are in the first project directory. Select each of these files and add them to the project. It is possible to highlight all the .c files in the add files dialog and add them in one go. Now, the project window should contain all the source files.



Build the project by selecting Project\Build target.



This will compile each of the .c modules in turn and then link them together to make a final application program. The output window shows the result of the build process and reports any errors or warnings.



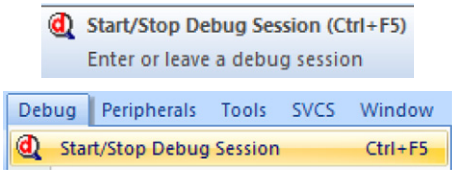
The program size is reported as listed in the following table.

Table 2.2: Linker Data Types

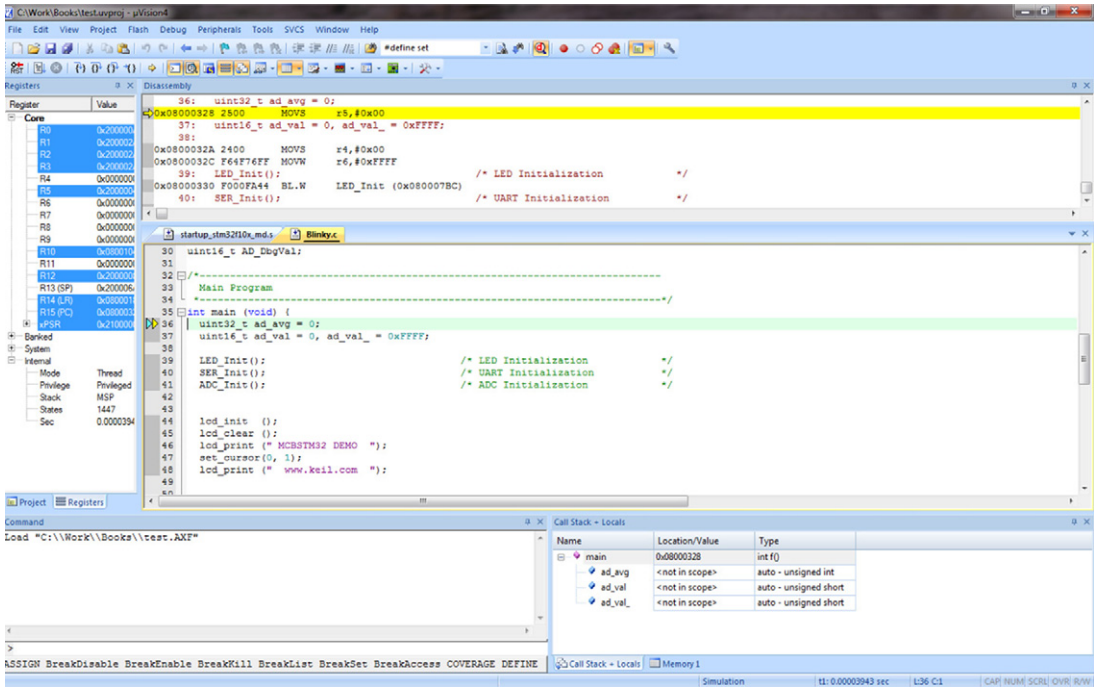
Code	Size of the executable image
RO data	Size of the code constants in the flash memory
RW data	Size of the initialized variables in SRAM
ZI data	Size of the uninitialized variables in SRAM

If errors or warnings are reported in the build window clicking on them will take you to the line of code in the editor window.

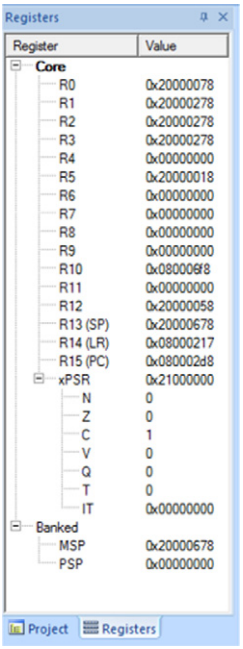
Now start the debugger and run the code.



This will connect μVision to the simulation model and download the project image into the simulated memory of the microcontroller. Once the program image has been loaded, the microcontroller is reset and the code is run until it reaches main() ready to start debugging.

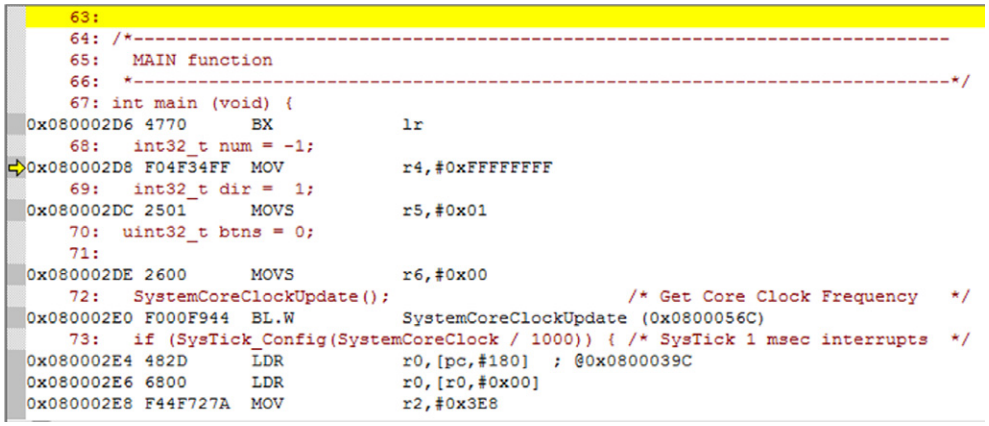


The μVision debugger is divided into a number of windows that allow you to examine and control the execution of your code. The key windows are as follows.



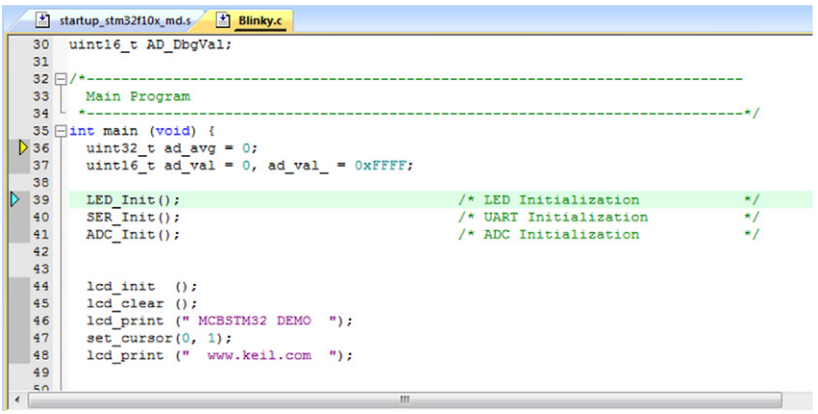
Register Window

The register window displays the current contents of the CPU register file (R0−R15), the program status register (xPSR), and also the main stack pointer (MSP) and the process stack pointer (PSP). We will look at all of these registers in the next chapter.

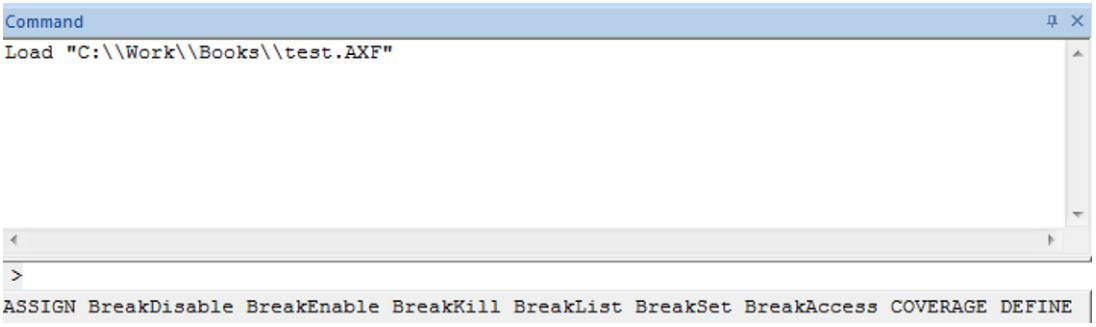


Disassembly Window

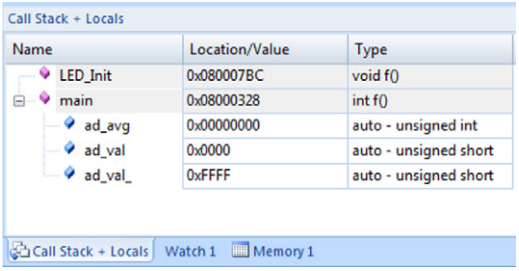
As its name implies the disassembly window will show you the low level assembler listing interleaved with the high level "C" code listing. One of the great attractions of the Cortex-M family is that all of your project code is written in a high level language such as C\C++. You never need to write low level assembly routines. However, it is useful if you are able to "read" the low level assembly code to see what the compiler is doing. The disassembly window shows the absolute address of the current instruction; next the opcode is shown, which is either a 16- or a 32-bit instruction. The raw opcode is then displayed as an assembler mnemonic. The current location of the program counter is shown by the yellow arrow in the left hand margin. The dark gray blocks indicate the location of executable lines of code.



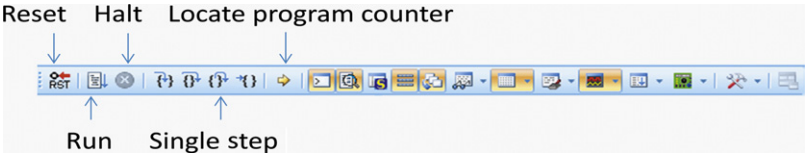
The source code window has a similar layout as the disassembly window. This window just displays the high level C source code. The current location of the program counter is shown by the yellow arrow in the left hand margin. The blue arrow shows the location of the cursor. Like the disassembly window, the dark gray blocks indicate the location of executable lines of code. The source window allows you to have a number of project modules open. Each source module can be reached by clicking the tab at the top of the window.



The command window allows you to enter debugger commands to directly configure and control the debugger features. These commands can also be stored in a text file and executed as a script when the debugger starts.

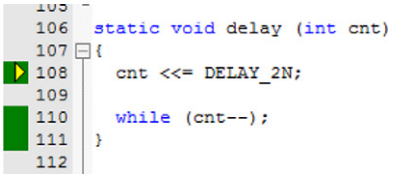


Next to the command window is a group of watch windows. These windows allow you to view local variables, global variables, and the raw memory.



You can control execution of the code through icons on the toolbar. The code can be a single stepped C or assembler line at a time, run at full speed and halted. The same commands are available through the debug menu, which also shows the function key shortcuts that you may prefer.

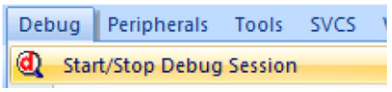
Start the code running for a few seconds and then press the halt icon.



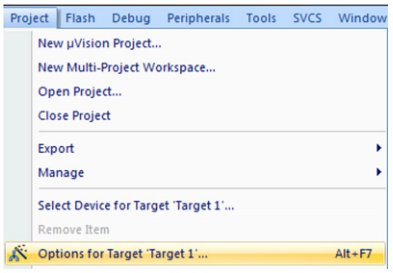
As code is executed in the simulator, the dark gray blocks next to the line number will turn green to indicate that they have been executed. This is a coverage monitor that allows you to verify that your program is executing as expected. In Chapter 8, we will see how to get this information from a real microcontroller.

It is likely that you will halt in a delay function. This is part of the application code that handles the LCD. Currently, the code is waiting for a handshake line to be asserted from the LCD. However, in the simulation only the Cortex-M processor and the microcontroller peripherals are supported. Simulation for external hardware devices like an LCD module has to be created to match the target hardware. This is done through a script file that uses a C-like language to simulate responses from the LCD.

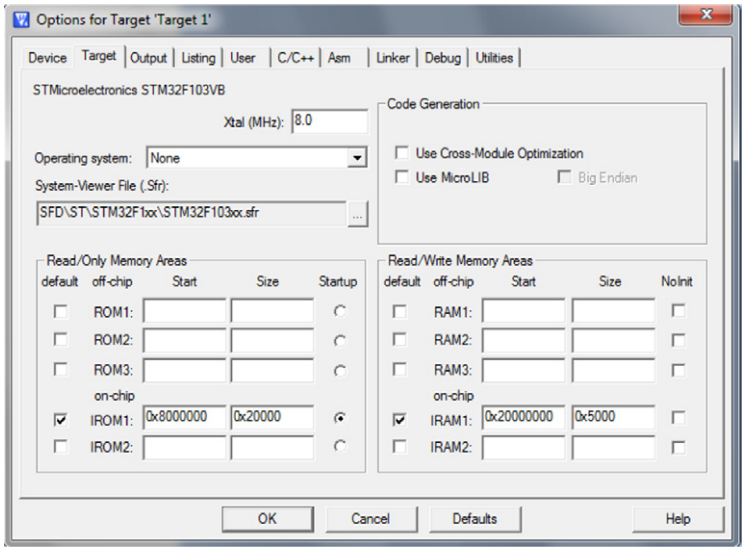
Exit the debugger with the Start/Stop Debug Session option.



Open the Options for Target dialog.

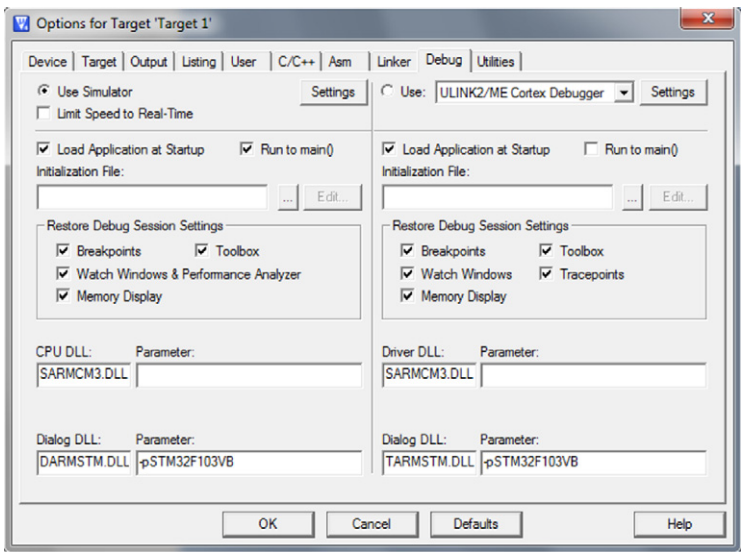


This can be done in the project menu by right clicking the project name and selecting Options for Target or by selecting the same option in the project menu from the main toolbar.



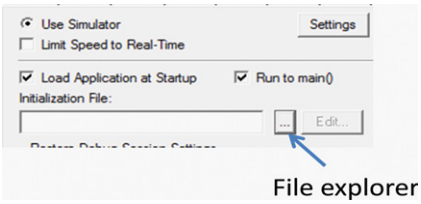
The Options for Target dialog holds all of the global project settings.

Now select the Debug tab.

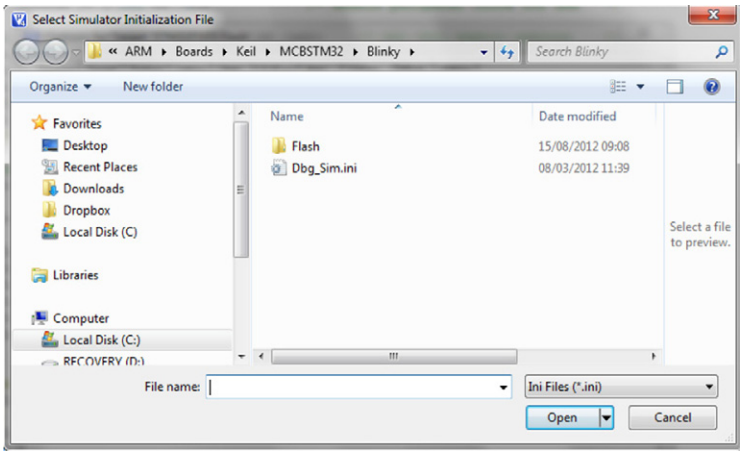


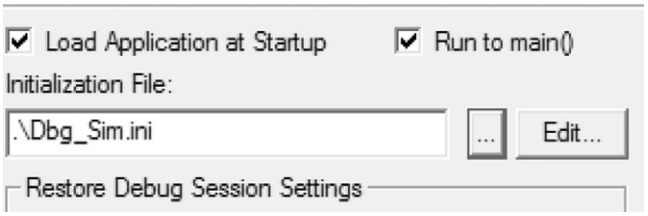
The Debug menu is split into two halves. The simulator options are on the left and the hardware debugger is on the right. Currently, the Use Simulator option is set.

Add the LCD simulation script.



Press the file explorer button and add the Dbg_sim.ini file, which is in the first project directory as the debugger initialization file.





The script file uses a C-like language to model the external hardware. All of the simulated microcontroller "pins" appear as virtual registers that can be read from and written to by the script. The debug script also generates a simulated voltage for the ADC. The script for this is shown below. This generates a signal that ramps up and down and it is applied to the virtual register ADC1_IN1, which is channel 1 of ADC convertor 1. The twatch function reads the simulated clock of the processor and halts the script for a specified number of cycles.

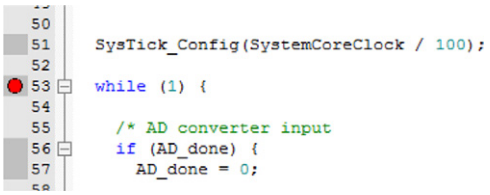
```
Signal void Analog (float limit) {
  float volts;
  printf ("Analog (%f) entered.\n", limit);
  while (1) {        /* forever */
```

```
    volts = 0;
    while (volts <= limit) {
      ADC1_IN1 = volts;        /* analog input-2 */
      twatch (250000);         /* 250000 Cycles Time-Break */
      volts += 0.1;          /* increase voltage */
    }
    volts = limit;
    while (volts >= 0.0) {
      ADC1_IN1 = volts;
      twatch (250000);         /* 250000 Cycles Time-Break */
      volts -= 0.1;         /* decrease voltage */
    }
  }
}
```

Click OK to close the Options for Target dialog.

Start the debugger.

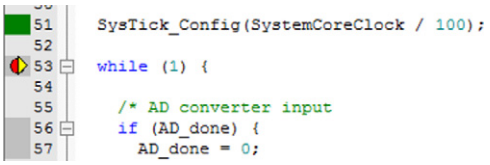
Set a breakpoint on the main while loop in blinky.c.



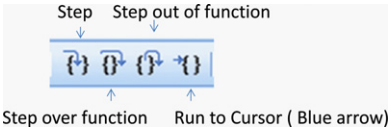
You can set a breakpoint by moving the mouse cursor into a dark gray block next to the line number and left clicking. A breakpoint is marked by a red dot.

Start executing the code.

With the simulation script in place we will be able to execute all of the LCD code and reach the breakpoint.



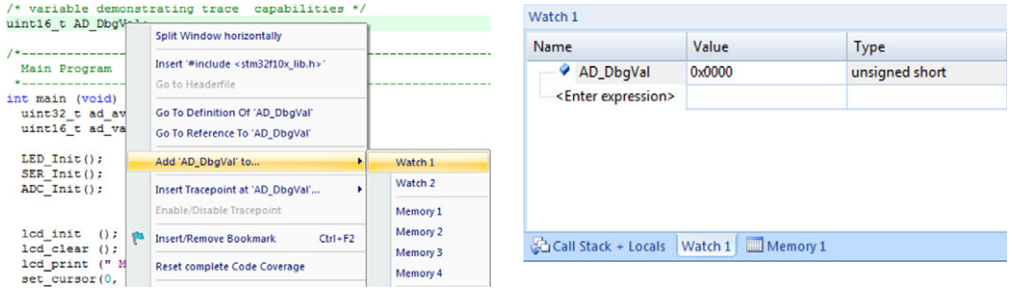
Now spend a few minutes exploring the debugger run control.



Use the single step commands, set a breakpoint, and start the simulator running at full speed. If you lose what is going on, exit the debugger by selecting debug/start/stop the debugger and then restart again.

Add a variable to the watch window.

Once you have finished familiarizing yourself with the run control commands within the debugger, locate the main() function within blinky.c. Just above main() is the declaration for a variable called AD_DbgVal. Highlight this variable, right click, and select Add AD_DbgVal to Watch 1.

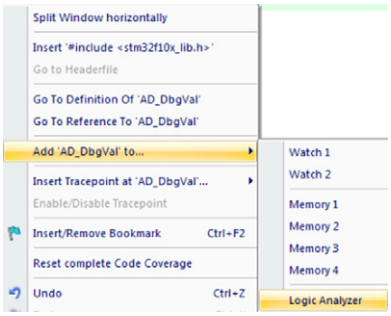


Now start running the code and you will be able to see the ADC variable updating in the watch window.

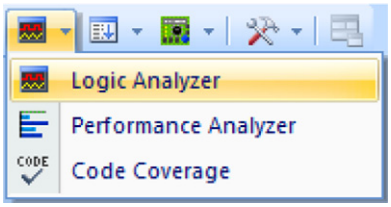
The simulation script is feeding a voltage to the simulated microcontroller ADC which in turn provides converted results to the application code.

Now add the same variable to the logic trace window.

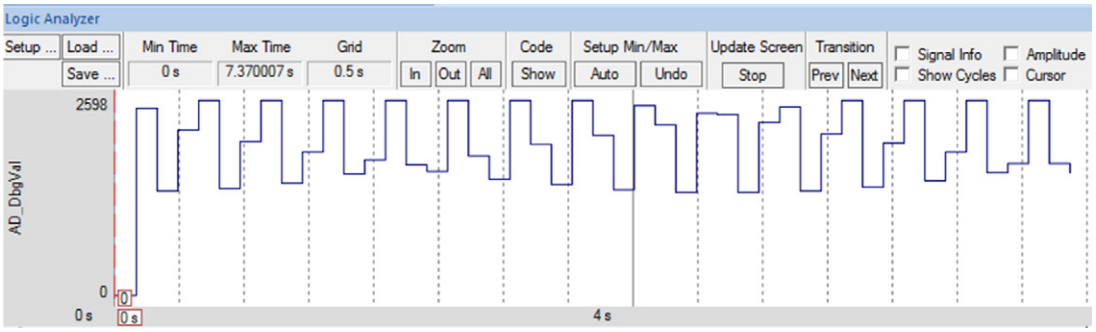
The μVision debugger also has a logic trace feature that allows you to visualize the historical values of a given global variable. In the watch window (or the source code window), highlight the AD_DbgVal variable name, right click, and select Add AD_DbgVal to Logic Analyzer.



If the logic analyzer window does not open automatically select it from the toolbar.



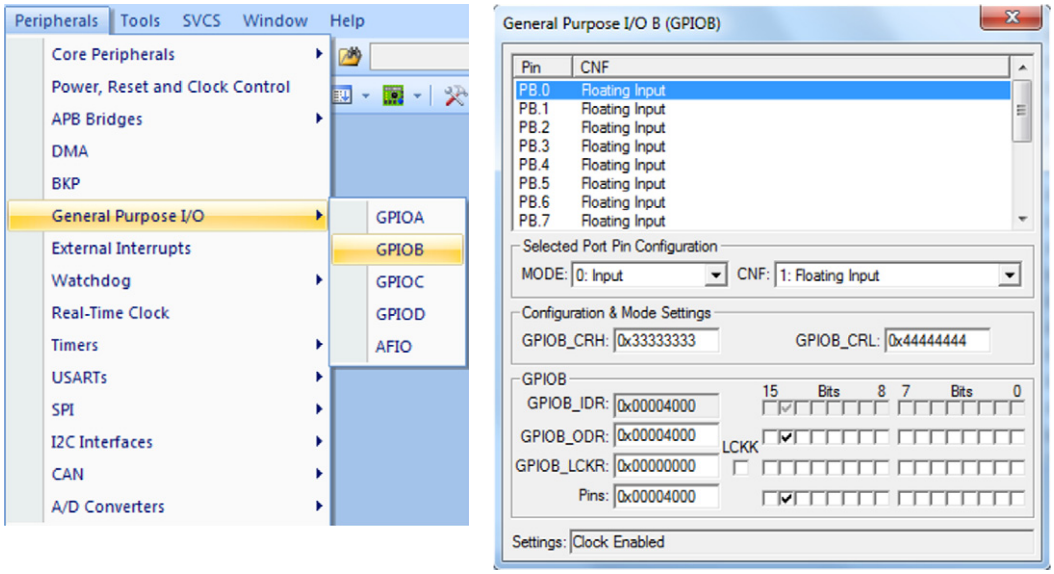
Now with the code running press the autoscale button, which will set the minimum and maximum values, and also click the zoom out button to get to a reasonable time scale. Once this is done, you will be able to view a trace of the values stored in the AD_DbgVal variable. You can add any other global variables or peripheral registers to the logic analyzer window.



Now view the state of the user peripherals.

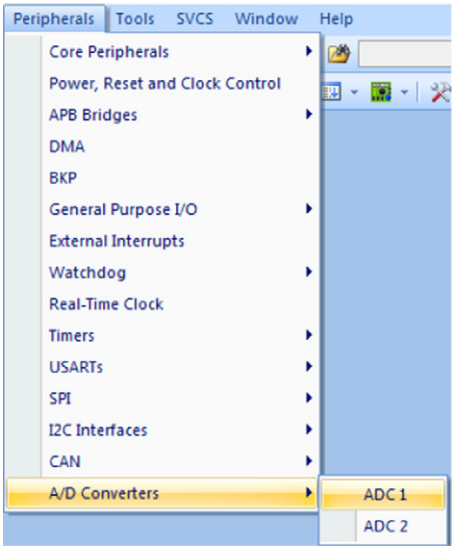
The simulator has a model of the whole microcontroller not just the Cortex-M processor, so it is possible to examine the state of the microcontroller peripherals directly.

Select Peripherals\General Purpose IO\GPIOB.

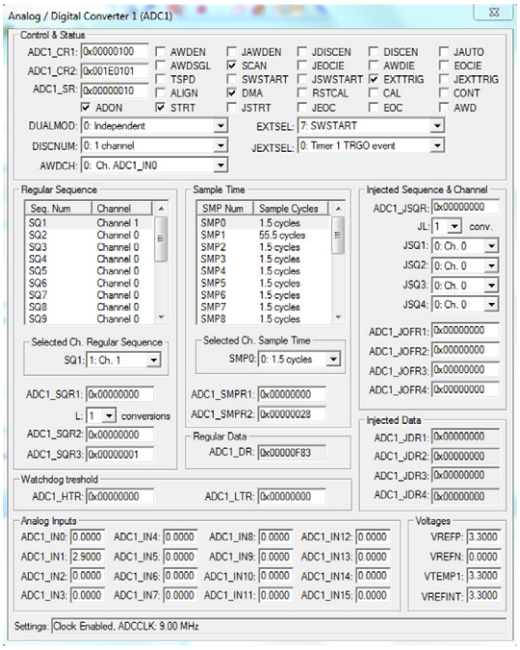


This will open a window that displays the current state of the microcontroller IO Port B. As the simulation runs, we can see the state of the port pins. If the pins are configured as inputs, we can manually set and clear them by clicking the individual Pins boxes.

You can do the same for the ADC by selecting ADC1.



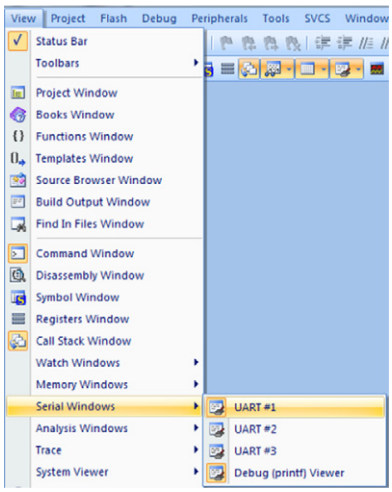
When the code is running it is possible to see the current configuration of the ADC and the conversion results. You can also manually set the input voltage by entering a fresh value in the Analog Inputs boxes.

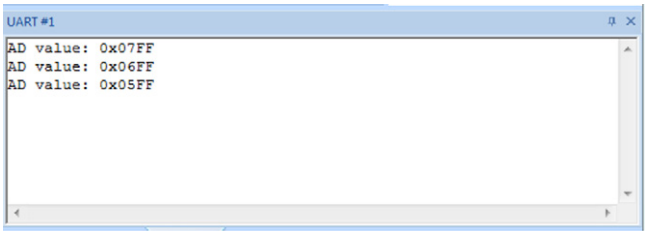


The simulator also provides a terminal that provides an I\O channel for the microcontroller's UARTS.

Select View\Serial Windows\UART #1.

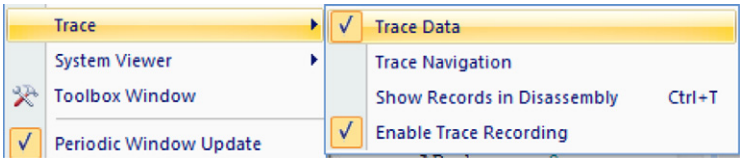
This opens a console-type window that displays the output from a selected UART and also allows you to input values.



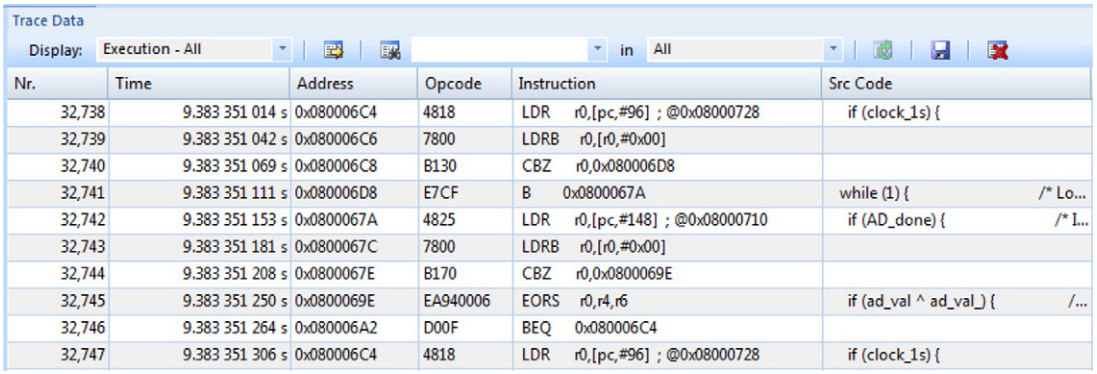


The simulator also boasts some advanced analysis tools including trace, code coverage, and performance analysis.

Open the View\Trace menu and select Trace Data and Enable Trace Recording.



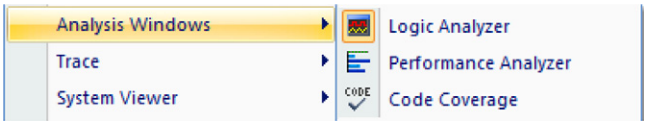
This will open the instruction trace window. This records a history of each instruction executed.



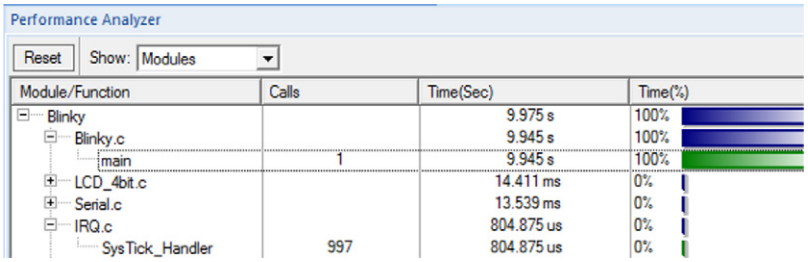
Nr.	Time	Address	Opcode	Instruction	Src Code
32,738	9.383 351 014 s	0x080006C4	4818	LDR r0,[pc,#96] ; @0x08000728	if (clock_1s) {
32,739	9.383 351 042 s	0x080006C6	7800	LDRB r0,[r0,#0x00]	
32,740	9.383 351 069 s	0x080006C8	B130	CBZ r0,0x080006D8	
32,741	9.383 351 111 s	0x080006D8	E7CF	B 0x0800067A	while (1) { /* Lo...
32,742	9.383 351 153 s	0x0800067A	4825	LDR r0,[pc,#148] ; @0x08000710	if (AD_done) { /* I...
32,743	9.383 351 181 s	0x0800067C	7800	LDRB r0,[r0,#0x00]	
32,744	9.383 351 208 s	0x0800067E	B170	CBZ r0,0x0800069E	
32,745	9.383 351 250 s	0x0800069E	EA940006	EORS r0,r4,r6	if (ad_val ^ ad_val_) { /...
32,746	9.383 351 264 s	0x080006A2	D00F	BEQ 0x080006C4	
32,747	9.383 351 306 s	0x080006C4	4818	LDR r0,[pc,#96] ; @0x08000728	if (clock_1s) {

Instruction trace is extremely valuable when tracking down complex bugs and validating software. In Chapter 8, we will see how to get instruction trace from a real microcontroller.

Now open the View/Analysis Windows/Code Coverage and View/Analysis Windows/ Performance Analyzer windows.

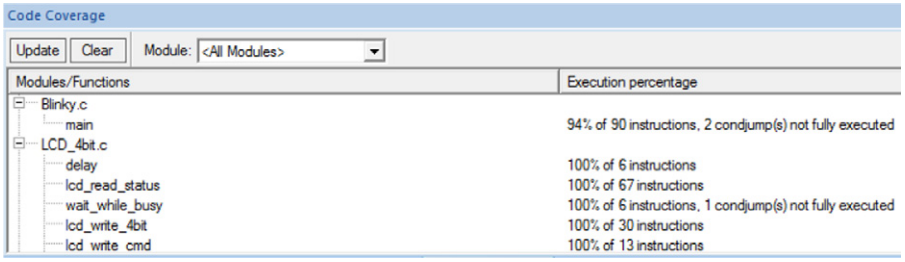


The performance analysis window shows the number of calls to a function and its cumulative runtime.



Module/Function	Calls	Time(Sec)	Time(%)
Blinky		9.975 s	100%
Blinky.c		9.945 s	100%
main	1	9.945 s	100%
LCD_4bit.c		14.411 ms	0%
Serial.c		13.539 ms	0%
IRQ.c		804.875 us	0%
SysTick_Handler	997	804.875 us	0%

The code coverage window provides a digest of the number of lines executed and partially executed in each function.

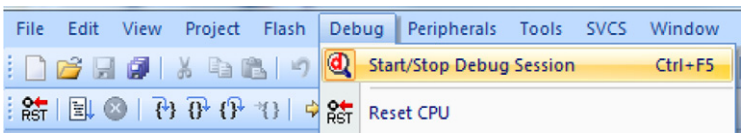


Both code coverage and performance analysis are essential for validating and testing software. In Chapter 8, we will see how this information can be obtained from a real microcontroller.

Project Configuration

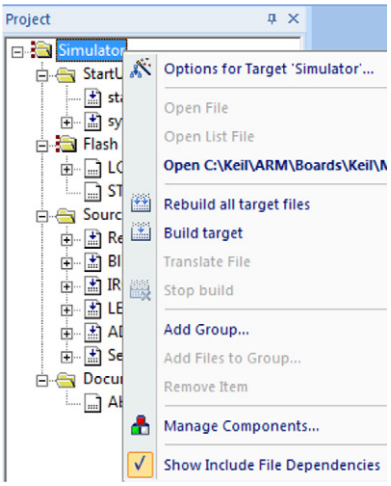
Now that you are familiar with the basic features of the debugger, we can look in more detail at how the project code is constructed.

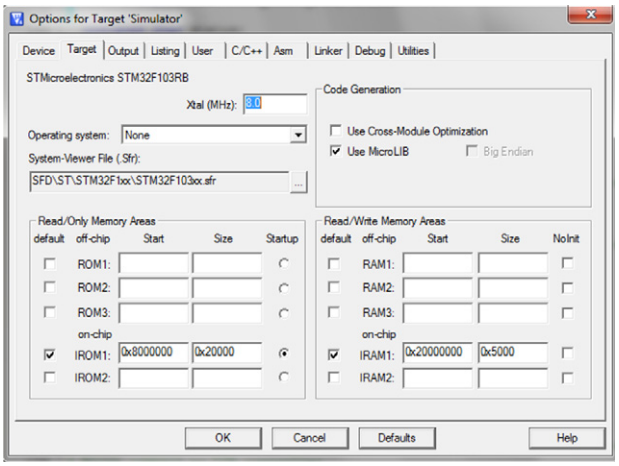
First quit the debugger by selecting Debug\Start/Stop Debug Session.



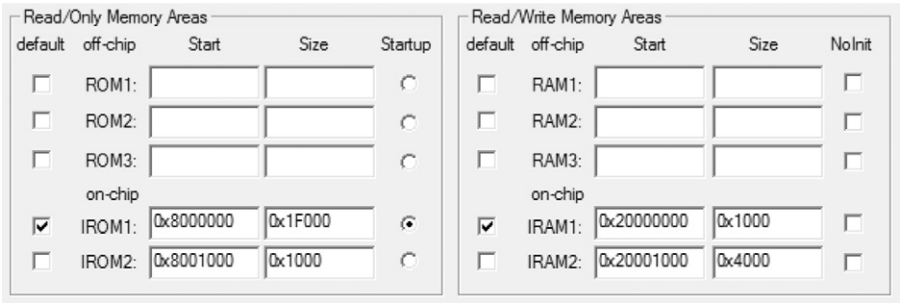
Open the Options for Target dialog.

All of the key project settings can be found in the Options for Target dialog.



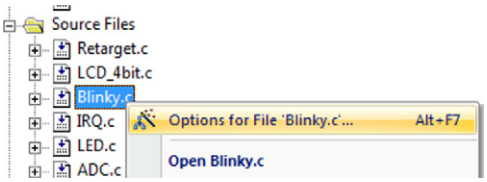


The target tab defines the memory layout of the project. A basic template is defined when you create the project. On this microcontroller, there are 128 K of internal flash memory and 20 K of SRAM. If you need to define a more complex memory layout, it is possible to create additional memory regions to subdivide the volatile and nonvolatile memories.

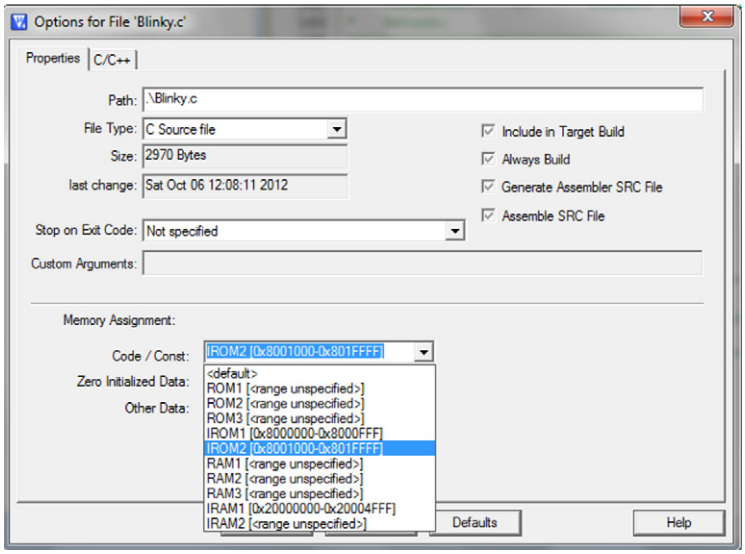


The more complex memory map above has split the internal flash into two blocks and defined the lower flash block as the default region for code and constant data. Nothing will be placed into the upper block unless you explicitly tell the linker to do this. Similarly, the SRAM has been split into two regions and the upper region is unused unless you explicitly tell the linker to use it. When the linker builds the project, it looks for the "RESET" code label. The linker then places the reset code at the base of the code region designated as the startup region. The initial startup code will write all of the internal SRAM to zero unless you tick the NoInit box for a given SRAM region. Then the SRAM will be left with its startup garbage values. This may be useful if you want to allow for a soft reset where some system data is maintained.

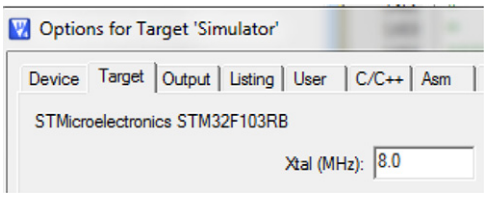
If you want to place objects (code or data) into an unused memory region, select a project module, right click, and open its local options.



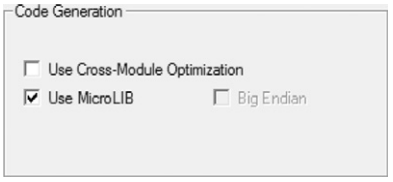
In its local options, the Memory Assignment boxes allow you to force the different memory objects in a module into a specific code region.



Back in the main Options for Target menu, there is an option to set the external crystal frequency used by the microcontroller.



Often this will be a standard value that can be multiplied by the internal phase locked loop oscillator of the microcontroller to reach the maximum clock speed supported by the microcontroller. This option is only used to provide the input frequency for the simulation model and nothing else.

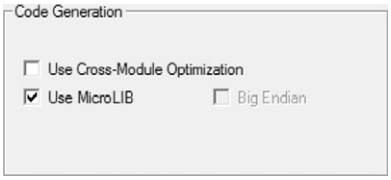


The Keil MDK-ARM comes with two ANSI library sets. The first is the standard library that comes with the ARM compiler. This is fully compliant with the current ANSI standard and as such has a large code footprint for microcontroller use. The second library set is the Keil MicroLIB; this library has been written to an earlier ANSI standard, that is, the C99 standard. This version of the ANSI standard is more in tune with the needs of microcontroller users.

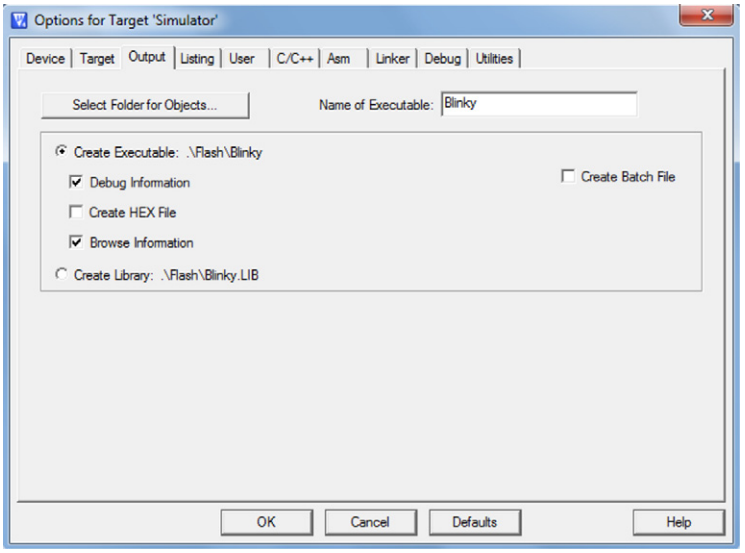
Table 2.3: Size Comparison between the Standard ARM ISO Libraries and the Keil Microlibrary

Processor	Object		Standard	MicroLib	% Savings
Cortex-M0(+)	Thumb	Library total	16,452	5996	64%
		RO total	19,472	9016	54%
Cortex-M3\M4	Thumb-2	Library total	15,018	5796	63%
		RO total	18,616	8976	54%

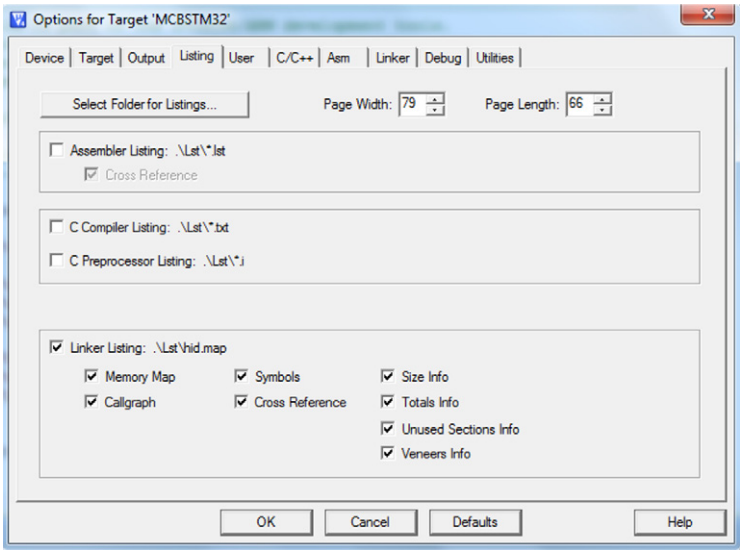
By selecting MicroLIB you will save at least 50% of the ANSI library code footprint verses the ARM compiler libraries. So try to use MicroLIB wherever possible. However, it does have some limitations, most notably it does not support all of the functions in standard library and double precision floating point calculations. In a lot of applications, you can live with this. Since most small microcontroller based embedded applications do not use these features you can generally use microLIB as your default library.



The Use Cross-Module Optimization tick box enables two pass linking process that fully optimizes your code. When you use this option, the code generation is changed and the code execution may no longer map directly to the C source code. So do not use this option when you are testing and debugging code as you will not be able to accurately follow it within the debugger source code window. We will look at the other options, system viewer file, and operating system in later chapters.



The Output menu allows you to control the final image of your project. Here, we can choose between generating a standalone program by selecting Create Executable and we can create a library that can be added to another project. The default is to create a standalone project with debug information. When you get toward the end of a project, you will need to select the Create HEX File option to generate a HEX32 file, which can be used with a production programmer. If you want to build the project outside of μVision, select Create Batch File and this will produce a <Project name>.bat DOS batch file that can be run from another program to rebuild the project outside of the IDE. By default, the name of the final image is always the same as your project name. If you want to change this, simply change the Name of Executable field. You can also select a directory to store all of the project, compiler, and linker generated files. This ensures that the original project directory only contains your original source code. This can make life easier when you are archiving projects.



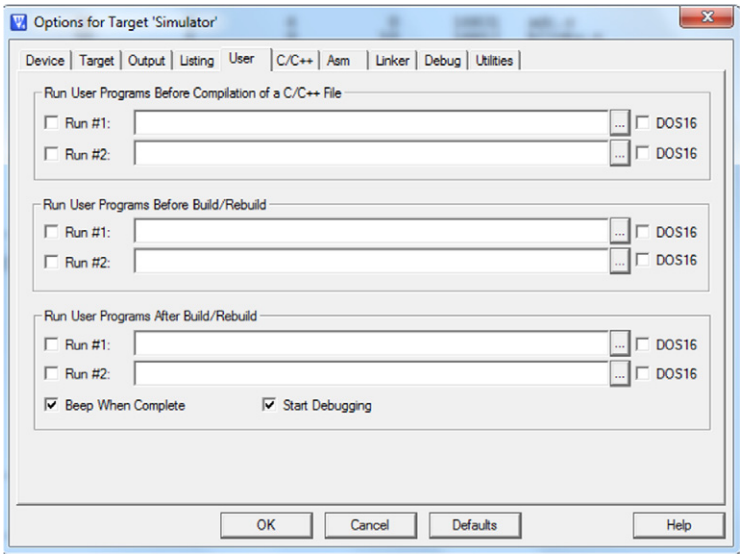
The Listing tab allows you to enable compiler listings and linker map files. By default, the linker map file is enabled. A quick way to open the map file is to select the project window and double click on the project root. The linker map file contains a lot of information, which seems incomprehensible at first, but there are a couple of important sections that you should learn to read and keep an eye on when developing a real project. The first is the "memory map of the image." This shows you a detailed memory layout of your project. Each memory segment is shown against its absolute location in memory. Here you can track which objects have been located to which memory regions. You can review the total amount of memory resources allocated, the location of the stack, and also if it has enabled the location of the heap memory.

```
Image Entry point : 0x000000cd

Load Region LR_IROM1 (Base: 0x00000000, Size: 0x00000d98, Max: 0x00040000, ABSOLUTE)

  Execution Region ER_IROM1 (Base: 0x00000000, Size: 0x00000d78, Max: 0x00040000, ABSOLUTE)

  Base Addr    Size         Type   Attr      Idx    E Section Name        Object

  0x00000000   0x000000cc   Data   RO            3    RESET               startup_lpc17xx.o
  0x000000cc   0x00000000   Code   RO          267  * .ARM.Collect$$$$00000000  mc_w.l(entry.o)
  0x000000cc   0x00000004   Code   RO          533    .ARM.Collect$$$$00000001  mc_w.l(entry2.o)
  0x000000d0   0x00000004   Code   RO          536    .ARM.Collect$$$$00000004  mc_w.l(entry5.o)
  0x000000d4   0x00000000   Code   RO          538    .ARM.Collect$$$$00000008  mc_w.l(entry7b.o)
  0x000000d4   0x00000008   Code   RO          539    .ARM.Collect$$$$00000009  mc_w.l(entry8.o)
  0x000000dc   0x00000004   Code   RO          534    .ARM.Collect$$$$00002712  mc_w.l(entry2.o)
  0x000000e0   0x00000024   Code   RO            5    .text               startup_lpc17xx.o
  0x00000104   0x00000030   Code   RO          100    .text               retarget.o
  0x00000134   0x00000080   Code   RO          127    .text               serial.o
  0x000001b4   0x000000ac   Code   RO          154    .text               led.o
  0x00000260   0x00000074   Code   RO          180    .text               irq.o
  0x000002d4   0x0000001e   Code   RO          542    .text               mc_w.l(llshl.o)
  0x000002f2   0x00000002   Code   RO          566    i.__scatterload_null  mc_w.l(handlers.o)
  0x000002f4   0x00000008   PAD
  0x000002fc   0x00000004   Code   RO            4    .ARM.__at_0x02FC    startup_lpc17xx.o
  0x00000300   0x00000310   Code   RO           15    .text               system_lpc17xx.o
  0x00000610   0x000000d8   Code   RO          207    .text               blinky.o
  0x000006e8   0x000000c4   Code   RO          239    .text               adc.o
  0x000007ac   0x00000062   Code   RO          270    .text               mc_w.l(uldiv.o)
  0x0000080e   0x00000020   Code   RO          544    .text               mc_w.l(llushr.o)
  0x0000082e   0x00000002   PAD
  0x00000830   0x00000024   Code   RO          557    .text               mc_w.l(init.o)
  0x00000854   0x00000020   Code   RO          479    i.__0printf$8       mc_w.l(printf8.o)
  0x00000874   0x00000028   Code   RO          481    i.__0sprintf$8      mc_w.l(printf8.o)
  0x0000089c   0x0000000e   Code   RO          565    i.__scatterload_copy  mc_w.l(handlers.o)
  0x000008aa   0x0000000e   Code   RO          567    i.__scatterload_zeroinit  mc_w.l(handlers.o)
  0x000008b8   0x00000420   Code   RO          486    i._printf_core      mc_w.l(printf8.o)
  0x00000cd8   0x00000026   Code   RO          487    i._printf_post_padding  mc_w.l(printf8.o)
  0x00000cfe   0x00000030   Code   RO          488    i._printf_pre_padding  mc_w.l(printf8.o)
  0x00000d2e   0x0000000a   Code   RO          490    i._sputc            mc_w.l(printf8.o)
  0x00000d38   0x00000020   Data   RO          155    .constdata          led.o
  0x00000d58   0x00000020   Data   RO          563    Region$$Table       anon$$obj.o


  Execution Region RW_IRAM1 (Base: 0x10000000, Size: 0x00000000, Max: 0x00004000, ABSOLUTE)

  **** No section assigned to this execution region ****


  Execution Region RW_IRAM2 (Base: 0x2007c000, Size: 0x00000230, Max: 0x00008000, ABSOLUTE)

  Base Addr    Size         Type   Attr      Idx    E Section Name        Object

  0x2007c000   0x00000004   Data   RW           16    .data               system_lpc17xx.o
  0x2007c004   0x00000008   Data   RW          101    .data               retarget.o
  0x2007c00c   0x0000000d   Data   RW          181    .data               irq.o
  0x2007c019   0x00000001   PAD
  0x2007c01a   0x00000004   Data   RW          240    .data               adc.o
  0x2007c01e   0x00000002   PAD
  0x2007c020   0x0000000a   Zero   RW          208    .bss                blinky.o
  0x2007c02a   0x00000006   PAD
  0x2007c030   0x00000200   Zero   RW            1    STACK               startup_lpc17xx.o
```

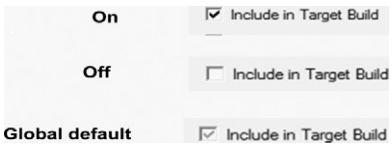
The second section gives you a digest of the memory resources required by each module and library in the project together with details of the overall memory requirements. The image memory usage is broken down into the code data size. The code data size is the amount of nonvolatile memory used to store the initializing values to be loaded into RAM variables on startup. In simple projects, this initializing data is held as a simple ROM table which is written into the correct RAM locations by the startup code. However, in projects with large amounts of initialized data, the compiler will switch strategies and use a compression algorithm to minimize the size of the initializing data. On startup, this table is decompressed before the data is written to the variable locations in memory. The RO data entry lists the amount of nonvolatile memory used to store code literals. The SRAM usage is split into initialized RW data and uninitialized ZI data.

```
Image component sizes

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name

       196         30          0          4          0      16831   adc.o
       216         50          0          0         10      16612   blinky.o
       116         20          0         13          0        641   irq.o
       172         16         32          0          0       1317   led.o
        48          0          0          8          0       2658   retarget.o
       128         18          0          0          0      17327   serial.o
        40         12        204          0        512        880   startup_lpc17xx.o
       784         76          0          4          0       5105   system_lpc17xx.o

    ----------------------------------------------------------------------
      1700        222        268         32        528      61371   Object Totals
         0          0         32          0          0          0   (incl. Generated)
         0          0          0          3          6          0   (incl. Padding)

    ----------------------------------------------------------------------

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Library Member Name

         0          0          0          0          0          0   entry.o
         8          4          0          0          0          0   entry2.o
         4          0          0          0          0          0   entry5.o
         0          0          0          0          0          0   entry7b.o
         8          4          0          0          0          0   entry8.o
        30          0          0          0          0          0   handlers.o
        36          8          0          0          0         68   init.o
        30          0          0          0          0         68   llshl.o
        32          0          0          0          0         68   llushr.o
      1224         62          0          0          0        504   printf8.o
        98          0          0          0          0         92   uldiv.o

    ----------------------------------------------------------------------
      1480         78          0          0          0        800   Library Totals
        10          0          0          0          0          0   (incl. Padding)

    ----------------------------------------------------------------------

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Library Name

      1470         78          0          0          0        800   mc_w.l

    ----------------------------------------------------------------------
      1480         78          0          0          0        800   Library Totals

    ----------------------------------------------------------------------

==============================================================================

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug

      3180        300        268         32        528      61027   Grand Totals
      3180        300        268         32        528      61027   ELF Image Totals
      3180        300        268         32          0          0   ROM Totals

==============================================================================

    Total RO  Size (Code + RO Data)                 3448 (   3.37kB)
    Total RW  Size (RW Data + ZI Data)               560 (   0.55kB)
    Total ROM Size (Code + RO Data + RW Data)       3480 (   3.40kB)
```

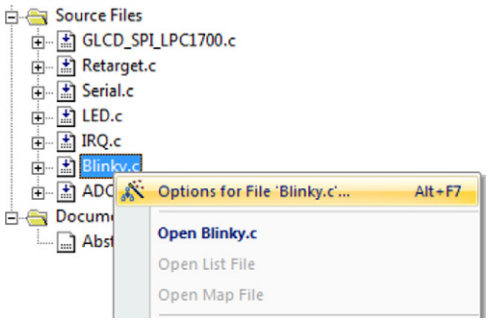
The next tab is the User tab. This allows you to add external utilities to the build process. The menu allows you to run a utility program to pre- or postprocess files in the project. A utility program can also be run before each module is compiled. Optionally, you can also start the debugger once the build process has finished.

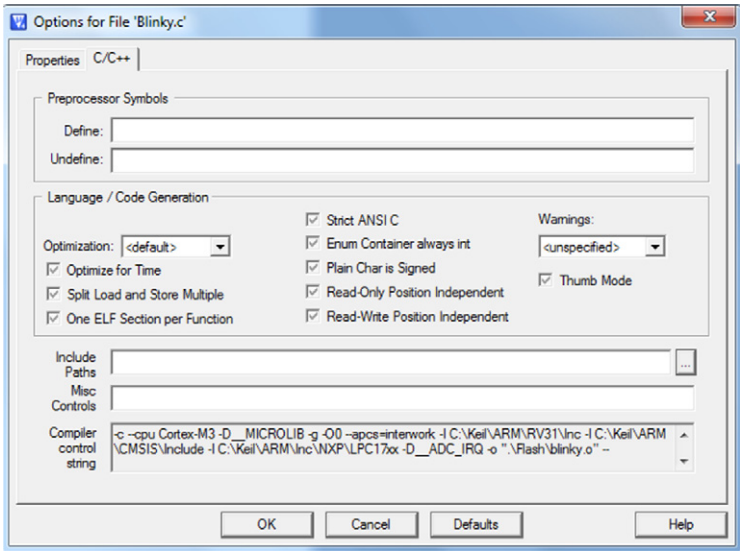


The code generated by the compiler is controlled by the C/C++ tab. This controls the code generation for the whole project. However, the same menu is available in the local options for each source module. This allows you to have global build options and then different build options for selected modules. In the local options menu, the option tick boxes are a bit unusual in that they have three states.

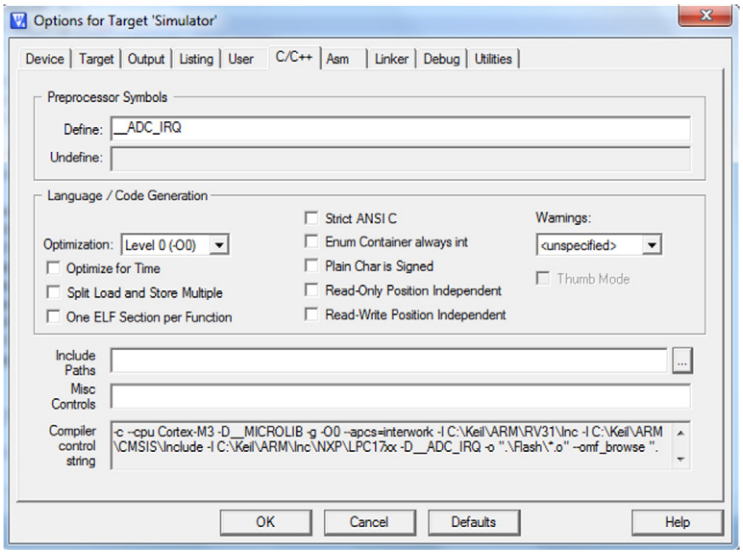


They can be unchecked, checked with a solid black tick, or checked with a gray tick. Here, the gray tick means "inherit the global options" and this is the default state.

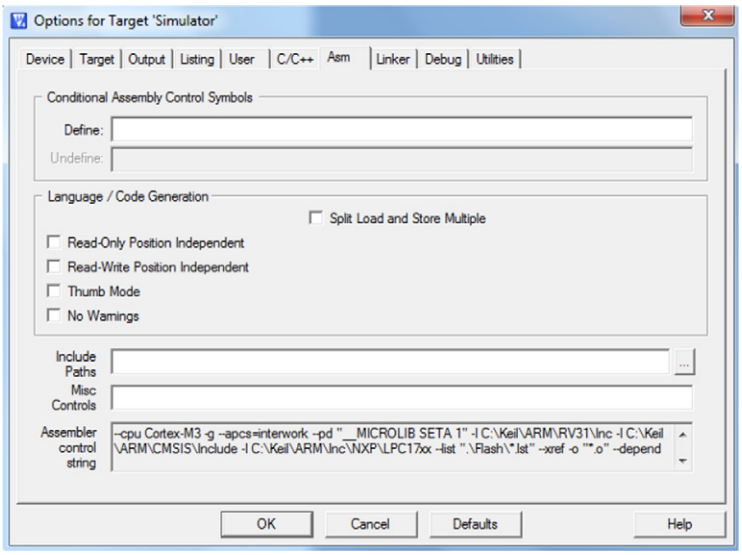




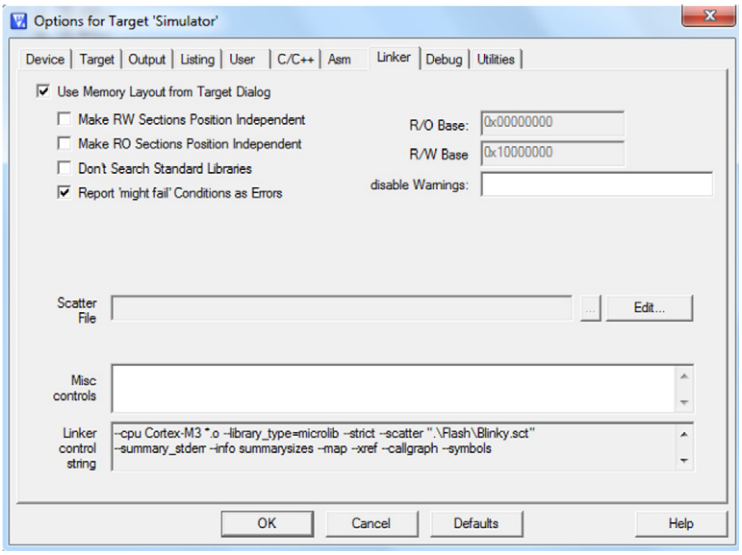
The most important option in this menu is the optimization control. During development and debugging you should leave the optimization level at zero. Then, the generated code maps directly to the high level “C” source code and it is easy to debug. As you increase the optimization level, the compiler will use more and more aggressive techniques to optimize the code. At the high optimization level, the generated code no longer maps closely to the original source code, which then makes using the debugger very difficult. For example, when you single step the code, its execution will no longer follow the expected path through the source code. Setting a breakpoint can also be hit and miss as the generated code may not exist on the same line as the source code. By default, the compiler will generate the smallest image. If you need to get the maximum performance, you can select the Optimize for Time option. Then, the compiler strategy will be changed to generate the fastest executable code.



The compiler menu also allows you to enter any #defines that you want to pass to the source module when it is being compiled. If you have structured your project over several directories, you may also add local include paths to directories with project header files. The Misc Controls text box allows you to add any compiler switches that are not directly supported in the main menu. Finally, the full compiler control string is displayed. This includes the CPU options, project include paths and library paths, and the make dependency files.



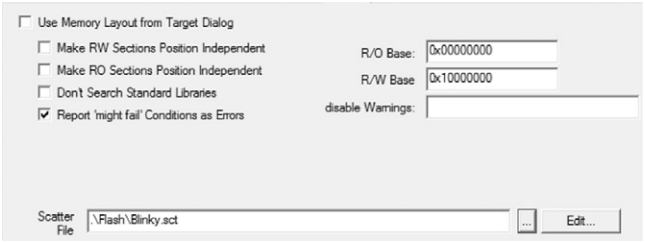
There is also an assembler options window that includes many of the same options as the C/C++ menu. However, most Cortex-M projects are written completely in C/C++, so with luck you will never have to use this menu!



By default, the Linker menu imports the memory layout from the Target menu. This memory layout is converted into a linker "scatter" file. The scatter file provides a text description of the memory layout to the linker so it can create a final image. An example of a scatter file is shown below.

```
*******************************************************************
; *** Scatter-Loading Description File generated by µVision ***
; *******************************************************************
LR_IROM1 0×00000000 0×00040000 {  ; load region size_region
  ER_IROM1 0×00000000 0×00040000 {  ; load address = execution address
    *.o (RESET, +First)
    *(InRoot$$Sections)
    .ANY (+RO)
  }
  RW_IRAM1 0×10000000 0×00004000 {  ; RW data
    .ANY (+RW +ZI)
  }
  RW_IRAM2 0×2007C000 0×00008000 {
    .ANY (+RW +ZI)
  }
}
```

The scatter file defines the ROM and RAM regions and the program segments that need to be placed in each segment. In the above example, the scatter file first defines a ROM region of 256 K. All of this memory is allocated in one bank. The scatter file also tells the linker to place the reset segment containing the vector table at the start of this section. Next, the scatter file then tells the linker to place all the remaining nonvolatile segments in this region. The scatter file then defines two banks of RAM of 16 and 32 K. The linker is then allowed to use both pages of RAM for initialized and uninitialized variables. This is a simple memory layout that maps directly onto the microcontroller's memory. If you need to use a more sophisticated memory layout, you can add extra memory regions in the Target menu and this will be reflected in the scatter file. If, however, you need a complex memory map which cannot be defined through the Target menu, then you will need to write your own scatter file. The trick here is to get as close as you can with the Target menu and then hand edit the scatter file.

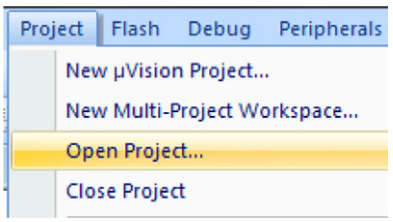


If you are using your own scatter file, you must then uncheck the Use Memory Layout from Target Dialog box and then manualy select the new scatter file using the Scatter File text box.

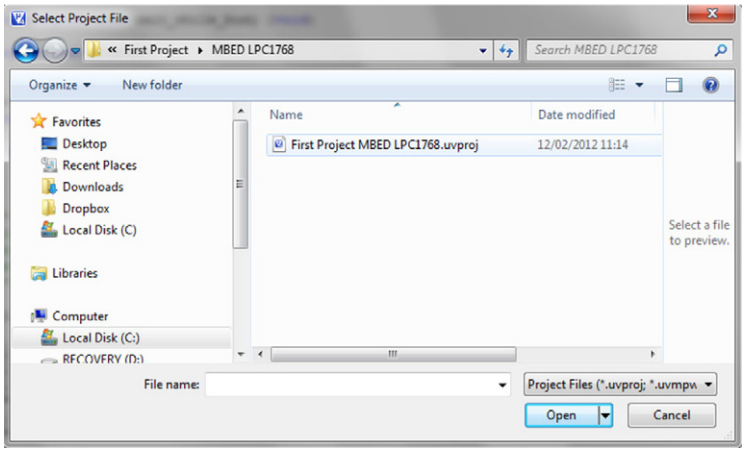
Hardware Debug

If you have downloaded an example set for a specific hardware module, the first project directory will contain a subdirectory named after the module you are using.

In μVision, select Project\Open Project.



Open the project located in the hardware module directory.

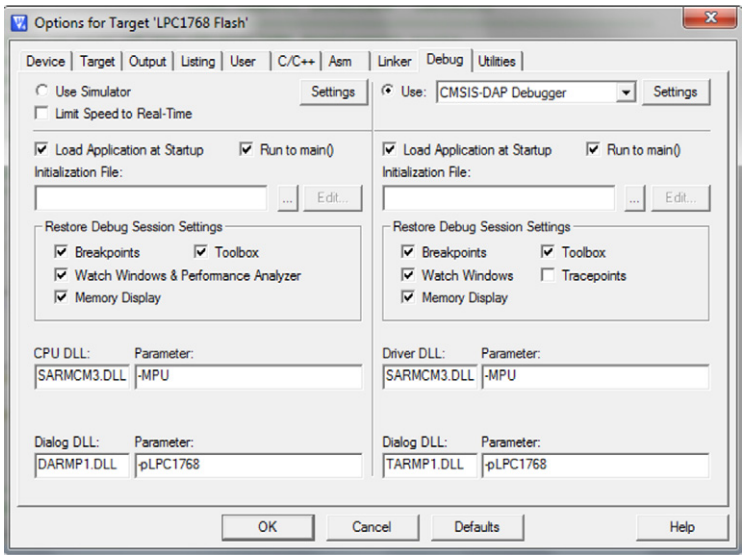


The project shown here is for the ARM MBED module based on the NXP LPC1768.

Build the project.

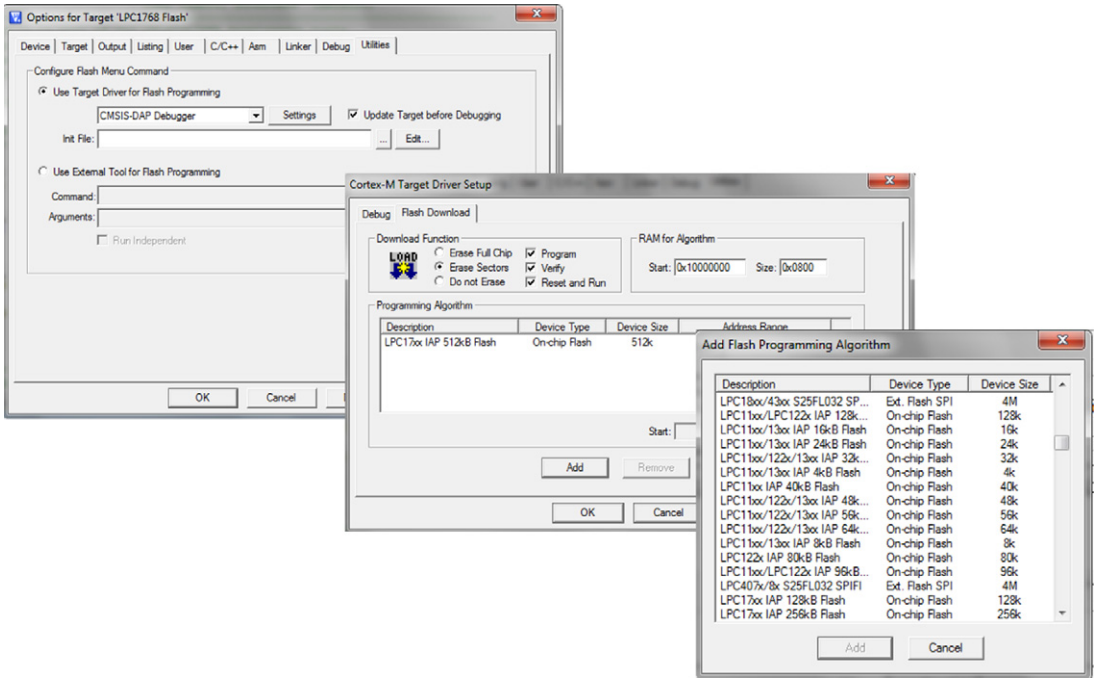
The project is created and builds in exactly the same way as the simulator version.

Open the Options for Target dialog and the Debug tab.



In the Debug menu, the Use option has been switched to select a hardware debugger rather than the simulator.

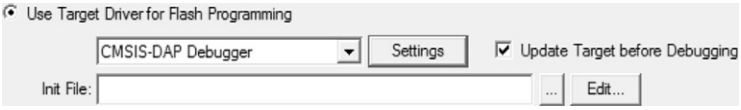
Now open the Utilities menu.



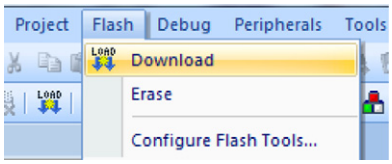
The Utilities menu allows you to select a tool to program the microcontroller flash memory. This will normally be the same as the debugger interface selected in the Debug menu. Pressing the setting button allows you to add the flash algorithm for the microcontroller.

The most common flash programming problems are listed below.

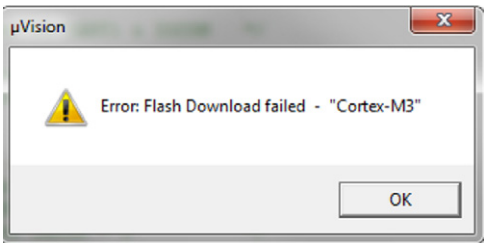
One point worth noting in the Utility menu is the Update Target before Debugging tick box.



When this option is ticked, the flash memory will be reprogrammed when the debugger starts. If it is not checked, then you must manually reprogram the flash by selecting Flash\Download from the main toolbar.



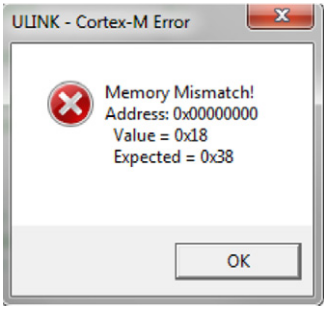
If there is a problem programming the flash memory, you will get the following error window pop-up.



The build output window will report any further diagnostic messages. The most common error is a missing flash algorithm. If you see the following message, check if the Options for Target\Utilities menu is configured correctly.

```
No Algorithm found for: 00000000H - 000032A3H
Erase skipped!
```

When the debugger starts, it will verify the contents of the flash against an image of the program. If the flash does not match the current image, you will get a memory mismatch error and the debugger will not start. This means that the flash image is out of date and the current version needs to be downloaded into the flash memory.



Select Cancel to close both of these dialogs without making any changes.

Start the debugger.

When the debugger starts, it is now connected to the hardware and will download the code into the flash memory of the microcontroller and allow the debugger interface to control the real microcontroller in place of the simulation model.

Experiment with the debugger interface now that it is connected to the real hardware.

You will note that some of the features available in the simulator are not present when using the hardware module. These are the instruction trace, code coverage, and performance analysis windows. These features are available with hardware debug, but you need a more advanced hardware interface to get them.