CHAPTER 8

Debugging with CoreSight

This chapter details the CoreSight debug architecture and in particular its real-time debug features.

Introduction

Going back to the dawn of microcontrollers, development tools were quite primitive. The application code was written in assembler and tested on an erasable programmable read-only memory (EPROM) version of the target microcontroller.

Figure 8.1
The electrical EPROM was the forerunner of today's flash memory.

To help debugging, the code was "instrumented" by adding additional lines of code to write debug information out of the UART or to toggle an IO pin. Monitor debugger programs were developed to run on the target microcontroller and control execution of the application code. While monitor debuggers were a big step forward, they consumed resources on the microcontroller and any bug that was likely to crash the application code would also crash the monitor program just at the point you needed it.

Figure 8.2
An in-circuit emulator provides nonintrusive real-time debug for older embedded microcontrollers.

If you had the money, an alternative was to use an in-circuit emulator. This was a sophisticated piece of hardware that replaced the target microcontroller and allowed full control of the program execution without any intrusion on the CPU runtime or resources. In the late 1990s, the more advanced microcontrollers began to feature various forms of on-chip debug unit. One of the most popular on-chip debug units was specified by the Joint Test Action Group and is known by the initials JTAG. The JTAG debug interface provides a basic debug connection between the microcontroller CPU and the PC debugger via a low-cost hardware interface unit.

Figure 8.3
Today, a low-cost interface unit allows you to control devices fitted with on-chip debug hardware.

JTAG allows you to start and stop running the CPU. It also allows you to read and write to memory locations and insert instructions into the CPU. This allows the debugger designer to halt the CPU, save the state of the processor, run a series of debug commands and then restore the state of the CPU, and restart execution of the application program. While this process is transparent to the user, it means that the PC debugger program has run control of the CPU (reset, run, halt, and set breakpoint) and memory access (read/write to user memory and peripherals). The key advantage of JTAG is that it provides a core set of debug features

with the reliability of an emulator at a much lower cost. The disadvantage of JTAG is that you have to add a hardware socket to the development board and the JTAG interface uses some of the microcontroller pins. Typically, the JTAG interface requires 5 GPIO pins, which may also be multiplexed with other peripherals. More importantly, the JTAG interface needs to halt the CPU before any debug information can be provided to the PC debugger. This run/stop style of debugging becomes very limited when you are dealing with a real-time system such as a communication protocol or motor control. While the JTAG interface was used on ARM7/9-based microcontrollers, a new debug architecture called CoreSight was introduced by ARM for all the Cortex-M/R- and Cortex-A-based processors.

CoreSight Hardware

When you first look at the datasheet of a Cortex-M-based microcontroller, it is easy to miss the debug features available or assume it has a form of JTAG interface. However, the CoreSight debug architecture provides a very powerful set of debug features that go way beyond what can be offered by JTAG. First of all on the practical side, a basic CoreSight debug connection only requires two pins, serial in and serial out.

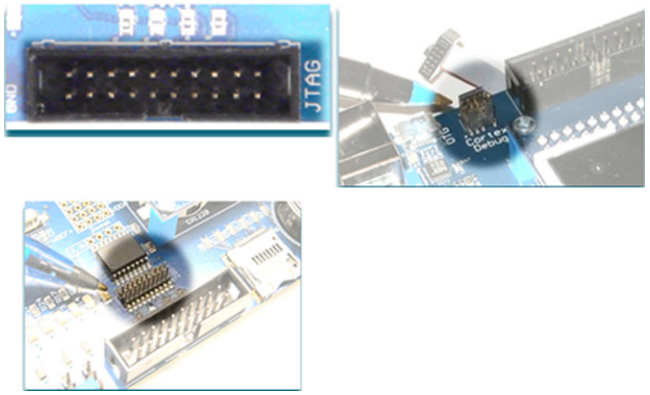


Figure 8.4
The CoreSight debug architecture replaces the JTAG berg connector with two styles of subminiature connector.

The JTAG hardware socket is a 20-pin berg connector that often has a bigger footprint on the PCB than the microcontroller that is being debugged. CoreSight specifies two connectors: a 10-pin connector for the standard debug features and a 20-pin connector for the standard debug features and instruction trace. We will talk about trace options later but if your microcontroller supports instruction trace then it is recommended to fit the larger 20-pin socket so you have access to the trace unit even if you do not initially intend to use it. A complex bug can be sorted out in hours with a trace tool where with basic run/stop debugging it could take weeks.

Table 8.1: CoreSight Debug Sockets

Socket	Samtec	Don Connex
10-pin standard debug	FTSH-105-01-L-DV-K	C42-10-B-G-1
20-pin standard + ETM trace	FTSH-110-01-L-DV-K	C42-20-B-G-1

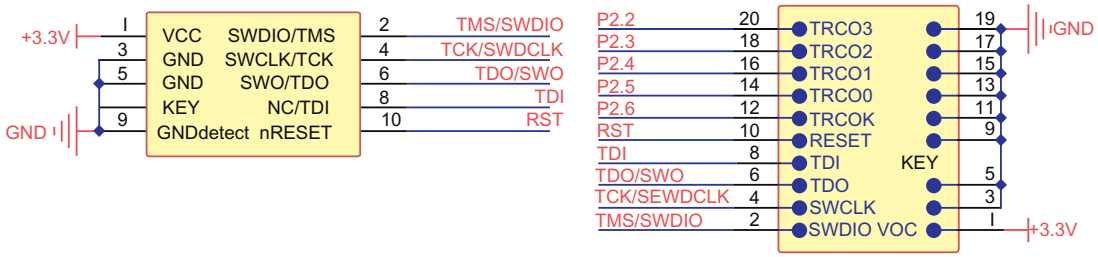


Figure 8.5
The two debug connectors require a minimum number of processor pins for hardware debug.

This standard debug system uses the 10-pin socket. This requires a minimum of two of the microcontroller pins, serial wire IO (SWIO) and serial wire clock (SWCLK), plus the target Vcc, ground, and reset. As we will see below, the Cortex-M3 and Cortex-M4 are fitted with two trace units that require an extra pin called serial wire out (SWO). Some Cortex-M3 and Cortex-M4 devices are fitted with an additional instruction trace unit. This is supported by the 20-pin connector and uses an additional four processor pins for the instruction trace pipe.

Debugger Hardware

Once you have your board fitted with a suitable CoreSight socket you will need a hardware debugger unit that plugs into the socket and is connected to the PC, usually through a USB or Ethernet connection. An increasing number of low-cost development boards also incorporate a USB debug interface. However, many of these use legacy JTAG and do not offer the full range of CoreSight functionality.

CoreSight Debug Architecture

There are several levels of debug support provided over the Cortex-M family. In all cases, the debug system is independent of the CPU and does not use processor resources or runtime.

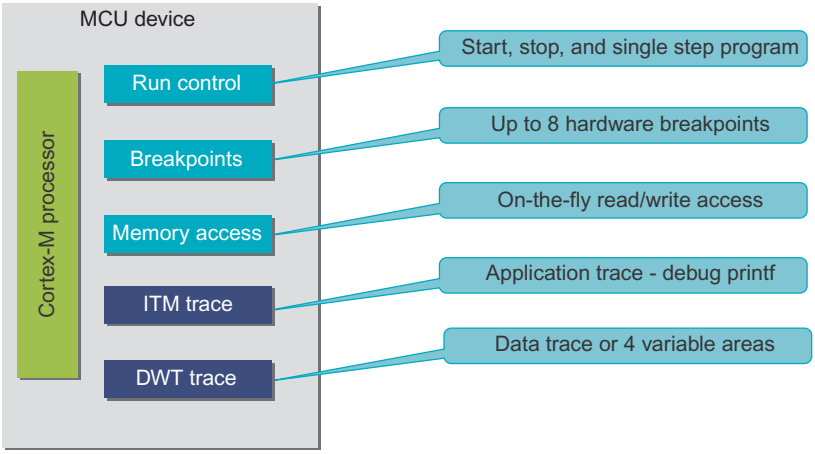


Figure 8.6
In addition to run control the Cortex-M3 and Cortex-M4 basic debug system includes two trace units: a data trace and an instrumentation trace.

The minimal debug system available on the Cortex-M3 and Cortex-M4 consists of the SW interface connected to a debug control system, which consists of a run control unit, breakpoint unit, and memory access unit. The breakpoint unit supports up to eight hardware breakpoints; the total number actually available will depend on the number specified by the silicon manufacturer when the chip is designed. In addition to the debug control units, the standard CoreSight debug support for Cortex-M3 and Cortex-M4 includes two trace units, a data watch trace and an instrumentation trace. The data watch trace allows you to view internal RAM and peripheral locations "on the fly" without using any CPU cycles. The data watch unit allows you to visualize the behavior of your application's data.

Exercise: CoreSight Debug

For most of the examples in this book, I have used the simulator debugger, which is part of the μVision IDE. In this exercise, however, I will run through setting up the debugger to work with the Ulink2 CoreSight debug hardware. While there are a plethora of evaluation boards available for different Cortex devices, the configuration of the hardware debug interface is essentially the same for all boards. There are hardware example sets for the ST Discovery boards, the NXP LPC1768 MBED and the Freescale Freedom boards. The instructions for this exercise customized for each board are included as a PDF in the project directory for each board.

Hardware Configuration

The ULINK2 debugger is connected to the development board through the 10-pin CoreSight debug socket and is in turn connected to the PC via USB.

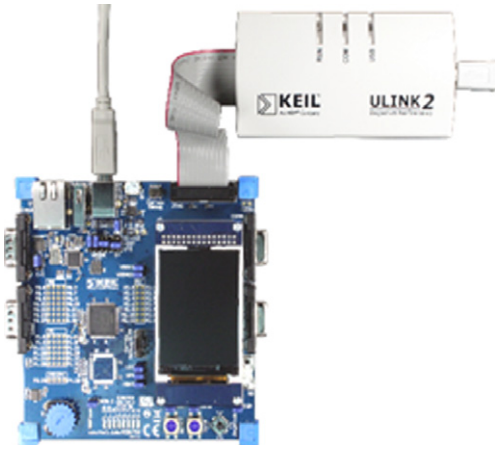


Figure 8.7
A typical evaluation board provides a JTAG/CoreSight connector. The evaluation board must also have its own power supply. Often this is provided via a USB socket.

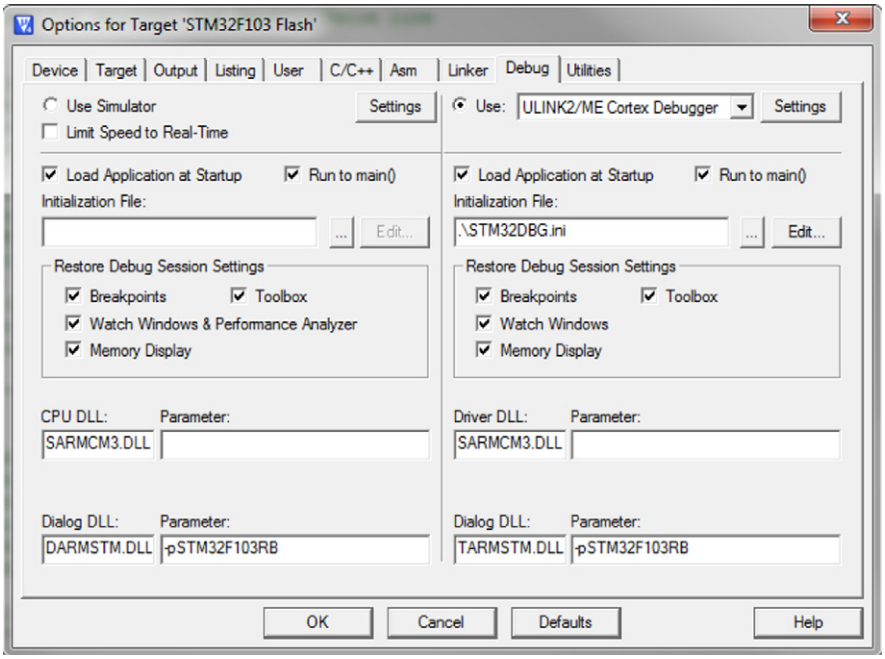
It is important to note here that the development board also has its own power supply. The Ulink2 will sink some current into the development board, enough to switch on LEDs on the board and give the appearance that the hardware is working. However, not enough current is provided to allow the board to run reliably; always ensure that the development board is powered by its usual power supply.

Software Configuration

Open the project in exercises\ulink2.

This is a version of the blinky example for the Keil MCBSTM32 evaluation board.

Now open the Options for Target\Debug tab.



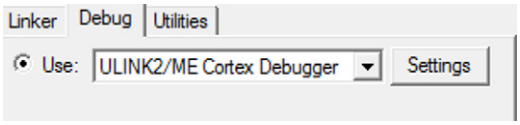
It is possible to switch from using the simulator to the CoreSight debugger by clicking on the Use ULINK2/ME Cortex Debugger radio button on the right-hand side of the menu. Like the simulator it is possible to run a script when the microvision debugger is started. While the debugger will start successfully without this script, the script is used to program a configuration register within the microcontroller debug system that allows us to configure any unique options for the microcontroller in use.

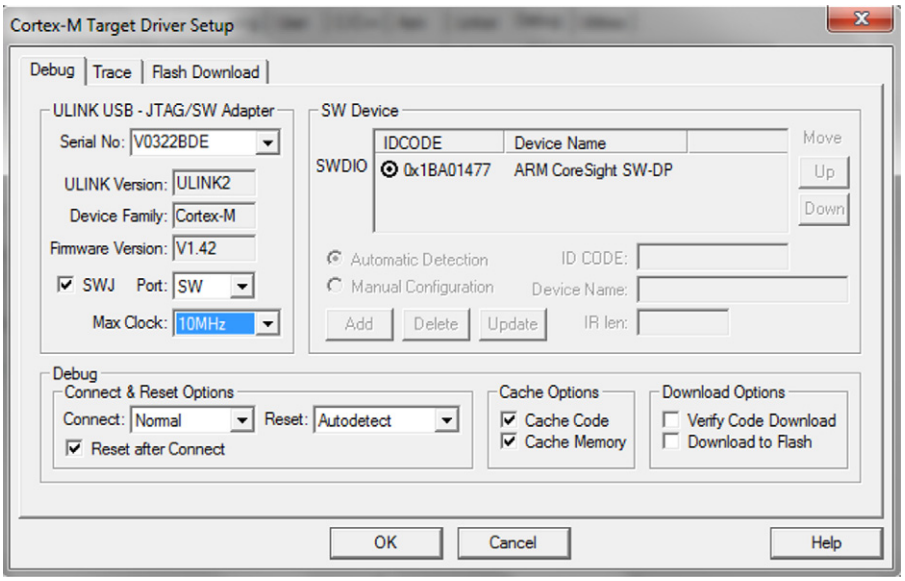
In the case of this microcontroller, the manufacturer has implemented some options to allow some of the user peripherals to be halted when the Cortex CPU is halted by the debug system. In this case, the user timers, watchdogs, and Controller Area Network (CAN) module may be frozen when the CPU is halted. The script also allows us to enable debug support when the Cortex CPU is placed into low-power modes. In this case, the sleep

modes designed by the microcontroller manufacturer can be configured to allow the clock source to the CoreSight debug system to keep running while the rest of the microcontroller enters the requested low-power mode. Finally, we can configure the serial trace pin. It must be enabled by selecting TRACE_IOEN with TRACE_MODE set to Asynchronous. When the debugger starts, the script will write the custom config value to the MCU debug register.

```
_WDWORD(0xE0042004, 0x00000027);
```

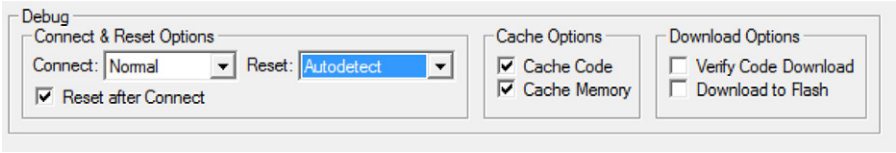
Now press the Settings button.





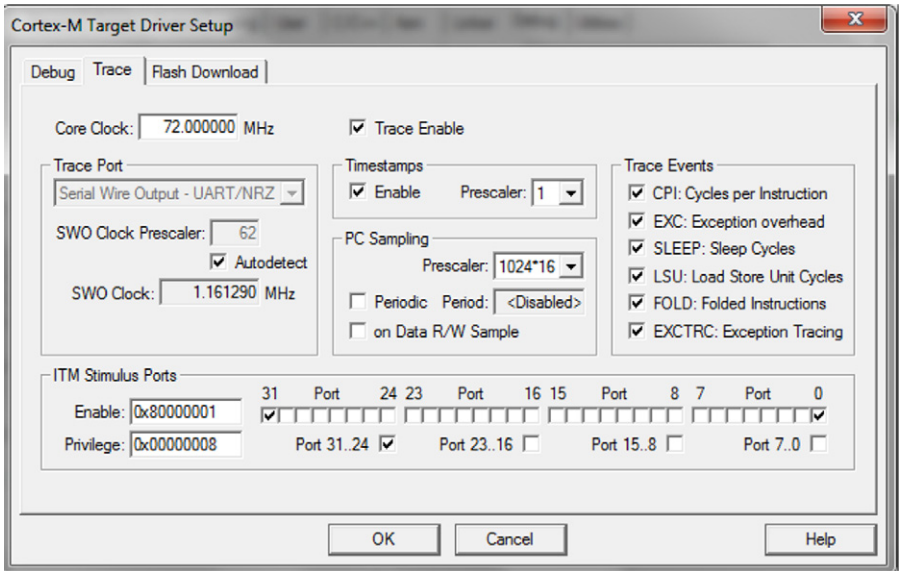
This will open the Ulink2 configuration window. The two main panes in this dialog show the connection state of the ULINK2 to the PC and to the microcontroller. The ULINK2 USB pane shows the ULINK2 serial number and firmware version. It also allows you to select which style of debug interface to use when connecting to the microcontroller. For Cortex-M microcontrollers, you should normally always use SW but JTAG is also available should you need it. The SWJ tick box allows the debugger to select the debug interface style dynamically.

When you are connected to a microcontroller, the SW device dialog will display all of the available debug interfaces (on some microcontrollers there may be more than one). Although you normally do not need to do any configuration to select a debug port, this information is useful in that it tells you that the microcontroller is running. When you are bringing up a new board for the first time it can be useful to check this screen when the board is first connected to get some confidence that the microcontroller is working.



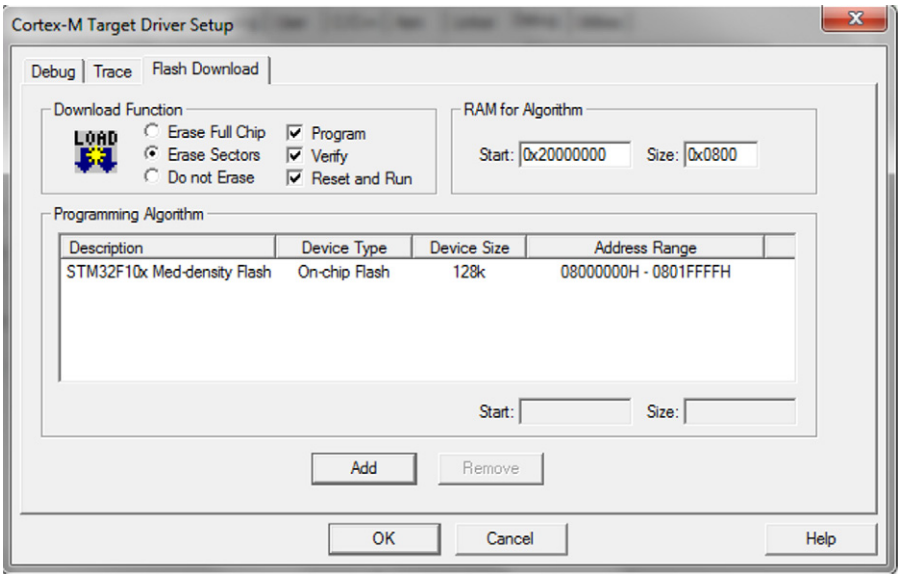
The debug dialog in the settings allows you to control how the debugger connects to the hardware. The Connect option defines if a reset is applied to the microcontroller when the debugger connects; you also have the option to hold the microcontroller in reset. Remember that when you program a code in the flash and reset the processor, the code will run for many cycles before you connect the debugger. If the application code does something to disturb the debug connection such as placing the processor into a sleep mode, then the debugger may fail to connect. The Ulink2 can be configured to hold the Cortex-M processor under reset at the start of a debug session to eliminate such problems. The reset method may also be controlled, and this can be a hardware reset applied to the whole microcontroller or a software reset caused by writing to the SYSRESET or VECTRESET registers in the NVIC. In the case of the SYSRESET option, it is possible to do a "warm" reset, that is, resetting the Cortex CPU without resetting the microcontroller peripherals. The cache options are affected when the physical memory is read and displayed. If the code is cached then the debugger will not read the physical memory but hold a cached version of the program image in the PC memory. If you are writing self-modifying code, you should uncheck this option. A cache of the data memory is also held; if used the debugger data cache is only updated once when the code is halted. If you want to see the peripheral registers update while the code is halted, you should uncheck this option.

Now click on the Trace tab.



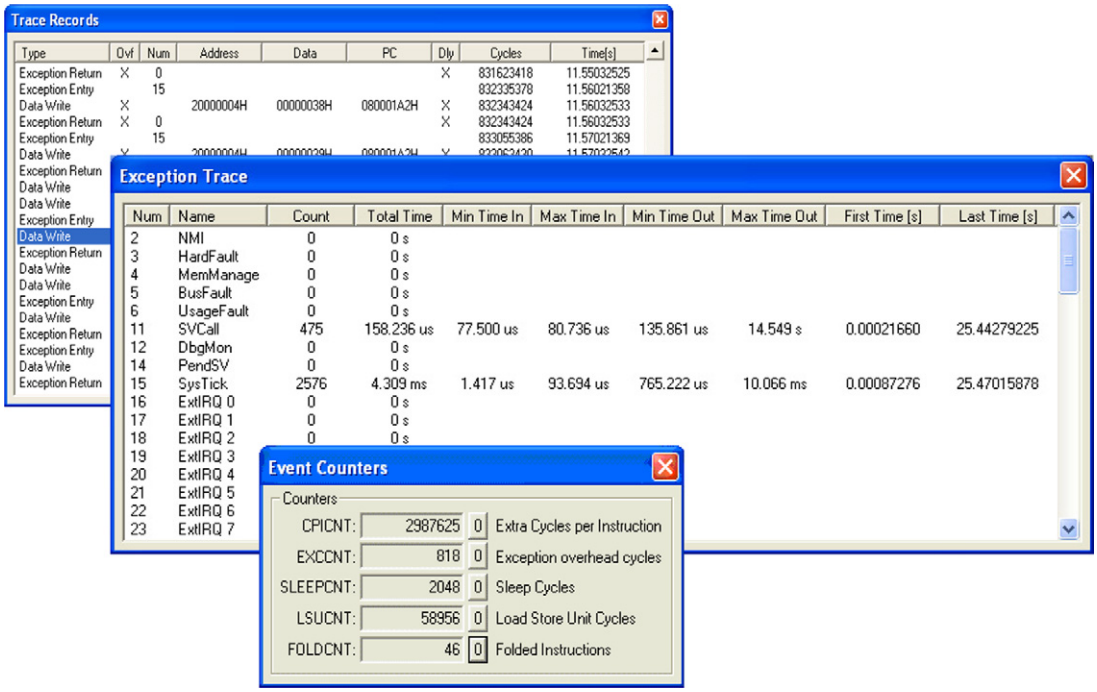
This dialog allows us to configure the internal trace units of the Cortex-M device. To ensure accurate timing, the core clock frequency must be set to the Cortex processor CPU clock frequency. The trace port options are configured for the SW interface using the UART communication protocol. In this menu, we can also enable and configure the data watch trace and enable the various trace event counters. This dialog also configures the instrumentation trace (ITM) and we will have a look at this later in this chapter.

Now click the Flash Download tab.



This dialog allows you to set the flash download algorithm for the microcontroller flash memory. This will normally be configured automatically when the project is defined. This menu allows you to update or add algorithms to support additional external parallel or serial flash memory.

Once configured you can start the debugger as normal and it will be connected to the hardware in place of the simulation model. This allows you to exercise the code on the real hardware through the same debugger interface we have been using for the simulator. With the data watch trace enabled you can see the current state of your variables in the watch and memory windows without having to halt the application code. It is also possible to add global variables to the logic analyzer to trace the values of a variable over time. This can be incredibly useful when working with real-time data.



The data watch trace windows give some high-level information about the runtime performance of the application code. The data watch trace windows provide a raw high-level trace exception and data access. An exception trace is also available, which provides detailed information of exception and interrupt behavior. The CoreSight debug architecture also contains a number of counters that show the performance of the Cortex-M processor. The extra cycles per instruction count is the number of wait states the processor has encountered waiting for the instructions to be delivered from the flash memory. This is a good indication at how efficiently the processor is running.

Debug Limitations

When the PC debugger is connected to the real hardware, there are some limitations compared to the simulator. Firstly, you are limited to the number of hardware breakpoints provided by the silicon manufacturer, and there will be a maximum of eight breakpoints. This is not normally a limitation but when you are trying to track down a bug or test code, it is easy to run out. The basic trace units do not provide any instruction trace or timing information. This means that the code coverage and performance analysis features are disabled.

Instrumentation Trace

In addition to the data watch trace, the basic debug structure on the Cortex-M3 and Cortex-M4 includes a second trace unit called the instrumentation trace. You can think of the instrumentation trace as a serial port that is connected to the debugger. You can then add code to your application that writes custom debug messages to the instrumentation trace (ITM) that are then displayed within the debugger. By instrumenting your code this way, you can send complex debug information to the debugger. This can be used to help locate obscure bugs but it is also especially useful for software testing.

The ITM is slightly different from the other two trace units in that it is intrusive on the CPU, i.e., it does use a few CPU cycles. The ITM is best thought of as a debug UART that is connected to a console window in the debugger. To use it, you need to instrument your code by adding simple send and receive hooks. These hooks are part of the CMSIS standard and are automatically defined in the standard microcontroller header file. The hooks consist of three functions and one variable.

```
static __INLINE uint32_t ITM_SendChar (uint32_t ch);    //Send a character to the ITM
volatile int ITM_RxBuffer = ITM_RXBUFFER_EMPTY;         //Receive buffer for the ITM
static __INLINE int ITM_CheckChar (void);               //Check to see if a character has
                                                          been sent from the debugger
static __INLINE int ITM_ReceiveChar (void);            //Read a character from the ITM
```

The ITM is actually a bit more complicated in that it has 32 separate channels. Currently, channel 31 is used by the RTOS kernel to send messages to the debugger for the kernel aware debug windows. Channel 0 is the user channel, which can be used by your application code to send printf() style messages to a console window within the debugger.

Exercise: Setting Up the ITM

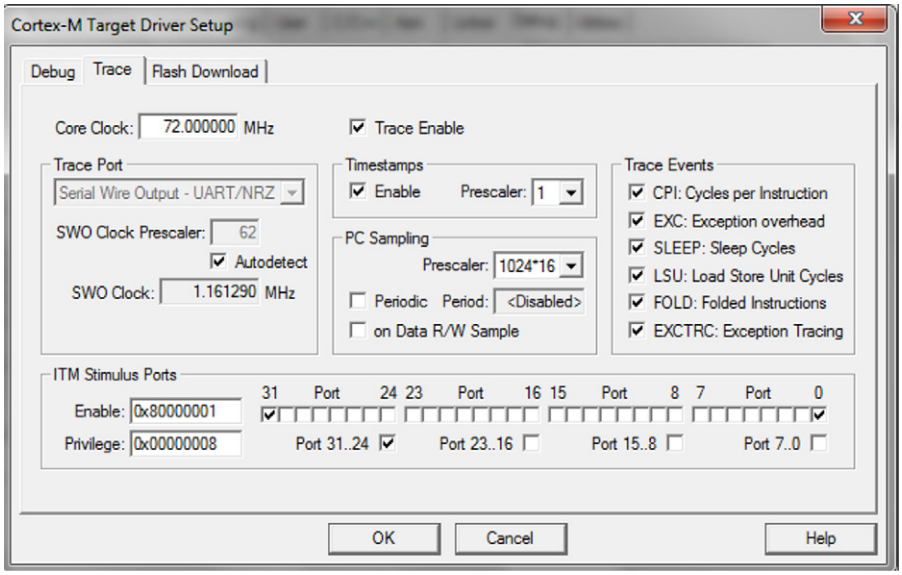
In this exercise, we will look at configuring the instrumentation trace to send and receive bytes between the microcontroller and the PC debugger.

Open the project in exercises\CoreSight ITM.

First we need to configure the debugger to enable the application ITM channel.

Open the Options for Target\Debug dialog.

Press the Ulink2 Settings button and select the Trace tab.



To enable the ITM, the core clock must be set to the correct frequency as discussed in the last exercise and the trace must be enabled.

In the ITM Stimulus Ports menu port 31 will be enabled by default. To enable the application port we must enable port 0. In order to use the application ITM port from privileged and unprivileged mode, the Privilege Port 7..0 box is unchecked.

Once the trace and ITM settings are configured, click OK and return back to the editor.

Now we have to modify the source code to use the ITM in place of a UART.

Open the file Serial.c.

```
void Ser_Init(void);
int SER_PutChar(int c);
int SER_GetChar(void);
int SER_CheckChar(void);
```

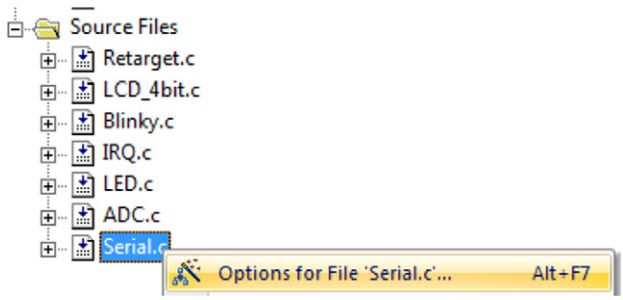
This module contains the low-level serial functions that are called by printf() and direct the serial data stream to one of the microcontroller's UARTs. It is also possible to redirect the serial data to the instrumentation trace using the CMSIS Core debug functions.

```
#ifdef __DBG_ITM
volatile int ITM_RxBuffer = ITM_RXBUFFER_EMPTY; /* CMSIS Debug Input */
#endif
```

```
int SER_PutChar (int c) {
#ifdef __DBG_ITM
  ITM_SendChar(c);
#else
  while (!(UART->LSR & 0x20));
  UART->THR = c;
#endif
  return (c);
}
```

For example, the SER_PutChar() function normally writes a character to a UART but if we create the #define __DBG_ITM the output will be directed to the application ITM port.

Open the local options for Serial.c.

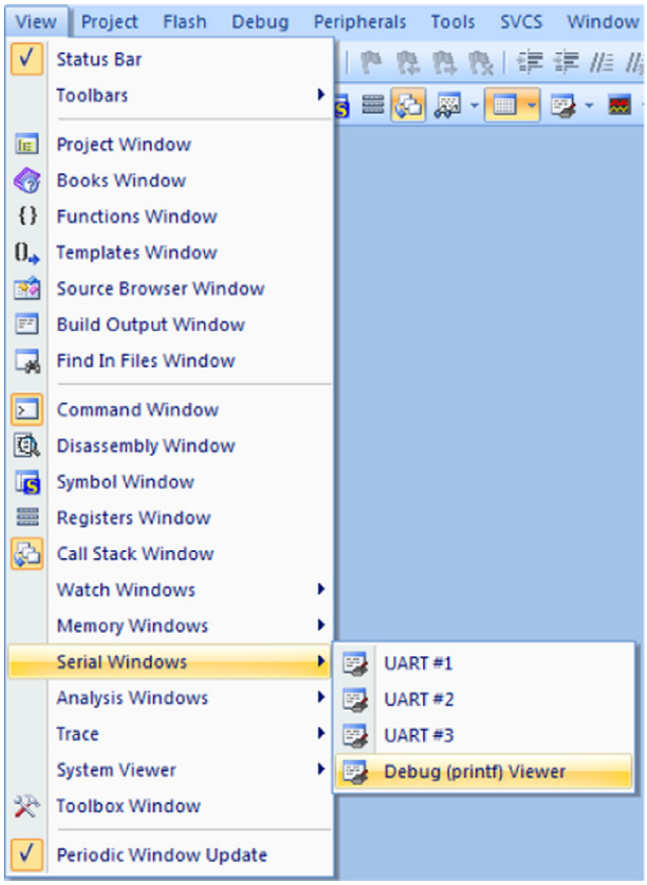


Now add the symbol ___DBG_ITM as a local #define.

Close the options menu and rebuild the application code.

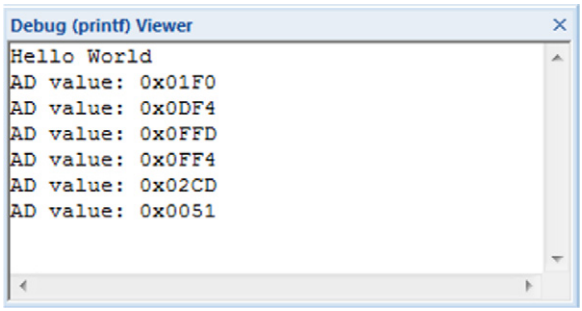
Start the hardware debugger.

Open View\Serial Windows\Debug (printf) Viewer.



Now start running the code and the serial IO will be redirected to the debugger window rather than the UART.

```
SysTick_Config(SystemCoreClock/100);
printf("Hello World\n\r");
  while (1) {
.....
if (clock_1s) {
  clock_1s = 0;
  printf("AD value: %s\r\n", text);
  }
.....
}
```



It is worth adding ITM support early in your project as it can provide a useful user interface to the running application code. Later in the project, the ITM can be used for software testing.

Software Testing Using the ITM with RTX RTOS

A typical RTX application will consist of a number of threads that communicate via various RTX methods such as main queues and event flags. As the complexity of the application grows, it can be hard to debug and test. For example, to test one region of the code it may be necessary to send several serial command packets to make one task enter a desired state to run the code that needs testing. Also if runtime errors occur we may not pick up on them until the code crashes and by then it may be too late to get meaningful debug information.

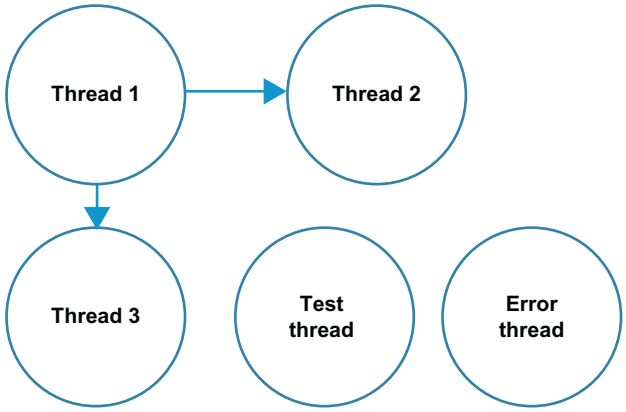


Figure 8.8
Two low-priority threads can be added to an application for error trapping and software testing.

Error Task

In an application, we can introduce two additional threads, an error thread and a test thread. The error thread consists of a switch statement that is triggered by the setting of its signal flags. Each RTX task has 16 error flags. Any other task can set these flags to trigger the task into activity. In the case of the error task, we can use the error flags to send an error code, i.e., a 16-bit number.

```
void errorHandler (void){
uint32_t errorCode;
while(1){
osSignalWait(0xFFFF,osWaitForever);    //wait here until error condition occurs
errorCode = osSignalGet(Error_t);      //read error code
switch(errorCode){                     //switch to run error code code
  case (THREAD2_MAILQUEUE_ERROR)
  ITM_SendChar('[');                  //send an error code to the debugger enclosed in [ ]
  ITM_SendChar('0');
  ITM_SendChar(']');
    break;
  }}}
```

Basically, the task waits until any of its event flags are set. It then reads the state of the error flags to get the "error code." A switch statement then uses the error code to execute some recovery code, and log the error to a file. During development we can also write a message to the debugger via the ITM. So in the main application code we can add statements to trap runtime errors.

```
Error = osMessagePut(Q_LED,0x1,osWaitForever);  //Send a mail message
if(Error! = osOK){  //If the message fails throw an error
  osSignalSetet(t_errorHandler, THREAD2_MAILQUEUE_ERROR);
}
```

In this case, we check for a free message slot before sending a message. If the mail queue is full then the error task is signaled by setting a group of error flags that can be read as an error code. During debug, critical runtime errors can be trapped with this method. Once the code is mature the error task becomes a central location for recovery code and fault logging.

Software Test Task

One of the strengths in using an RTOS is that each of the threads is encapsulated with well-defined inputs and outputs. This allows us to test the behavior of individual threads without having to make changes to the application code, i.e., commenting out functions, creating a test harness, and so on. Instead we can create a test thread. This runs at a low priority so it does not intrude on the overall performance of the application code. Like the error thread, the test thread is a big switch statement but rather than waiting to be triggered by event flags in the application it is waiting for an input from the instrumentation trace.

```
void testTask (void){
unsigned int command;
while(FOREVER){
while (ITM_CheckChar()! = 1) __NOP(); //wait for a character to be sent from the debugger
command = ITM_ReceiveChar();       //read the ITM character
switch(command){                  //switch to the test code
  case ('1'):                      /inject a message into the thread2 mail queue
  test_func_send_thread2_message(CMD_START_MOTOR,0x07,
  0x01,0x07,0,0,0,0);  //send a message to thread2 break;
  case ('2'):                     //halt thread1
  osThreadTerminate(t_thread1);
  default:
  break;
}}}
```

In this case, the test thread is waiting for a character to arrive from the debugger. The switch statement will then execute the desired test code. So for example we could halt a section of the application by deleting the thread, and another debug command could restart it. Then we can inject messages into the RTX mail queues or set event flags to trigger the behavior of the different tasks. This technique is simple and quick to implement and helps to debug application code as it becomes more complex. It is also a basis for the functional testing of mature application code. Because we are using the ITM we only need to access the debug header, and minimal resources are used on the microcontroller. All of the instrumented code is held in one module so it can be easily removed from the project once testing is finished.

Exercise: Software Testing with the ITM

This exercise is designed to run on the MBED and Discovery modules.

Program the project into the microcontroller's flash memory.

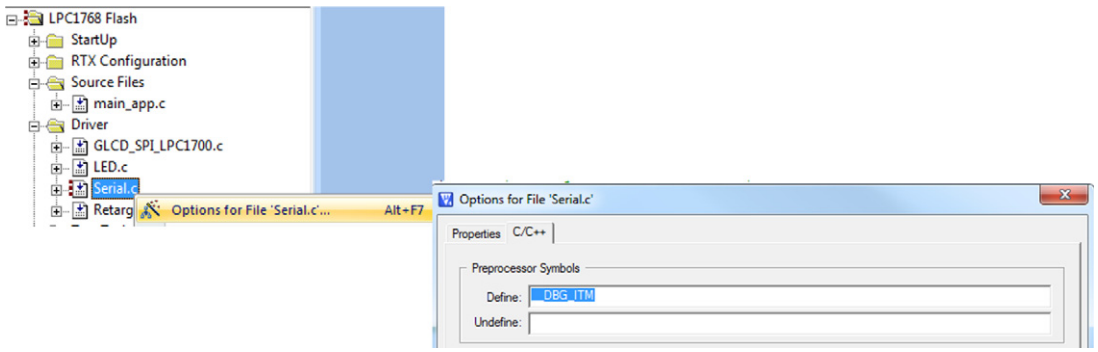
Start the debugger and let the code run.

You should see the LEDs on the board flash in a symmetrical pattern.

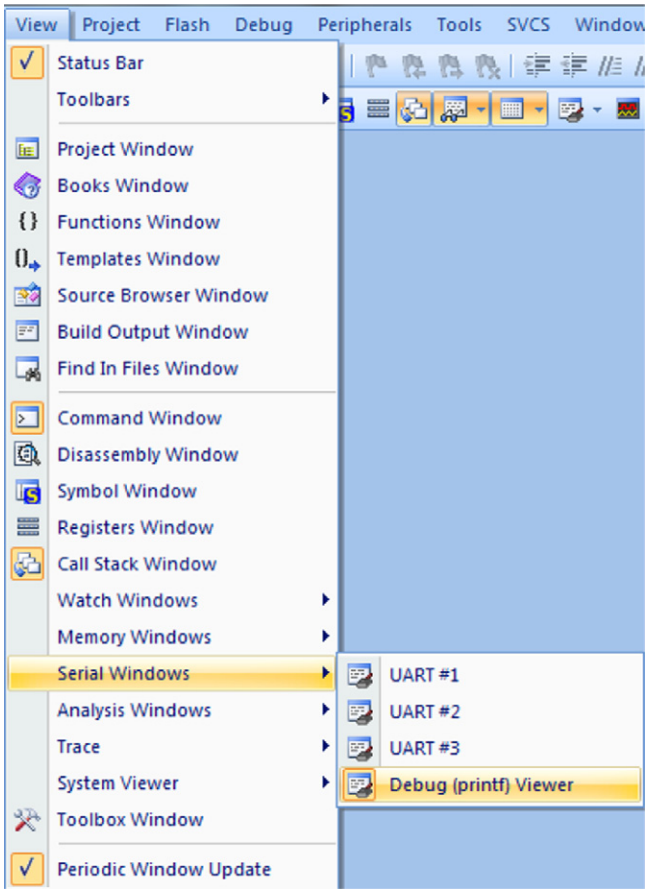
The application code is located in main_app.c; take a few minutes to look through this to familiarize yourself with this simple application. The CMSIS core layer, which includes the ITM functions, is brought into the project with the microcontroller include file:

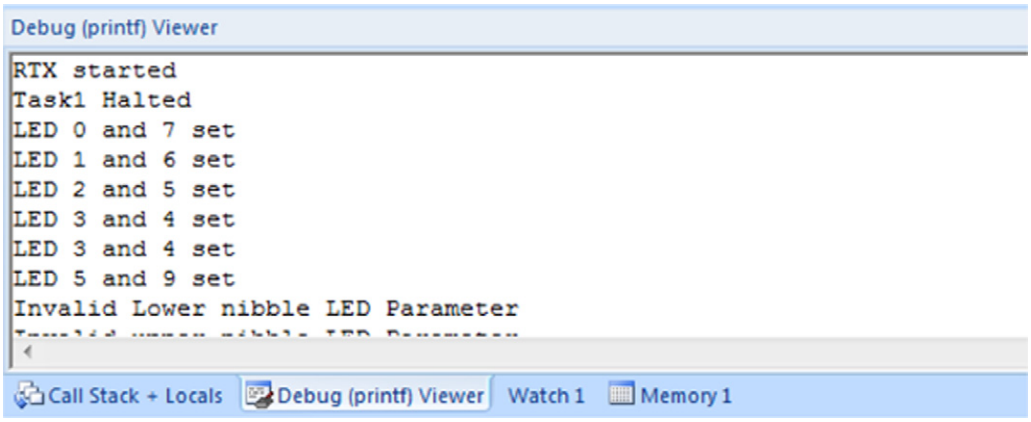
```
#include <lpc1768xx.h>
```

The files error.c and test_functions.c provide the error handling and test functions, respectively. For clarity these functions use printf to send test messages to the ITM. In a real application, more lightweight messaging could be used to minimize the function size and number of CPU cycles used. The file Serial.c contains the low-level functions used by printf(). By default these use a standard UART but output can be redirected to the ITM with the #define __DBG_ITM. This is located in the local options for Serial.c.



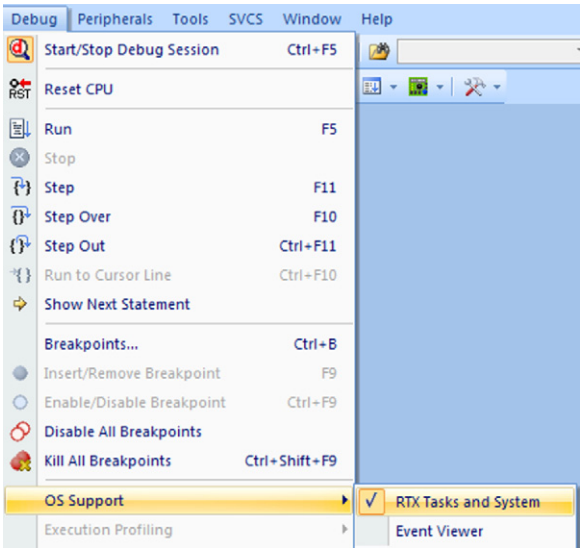
To access the ITM from within the debugger, select View\Serial Windows\Debug (printf) Viewer.





This opens a console style window that allows you to send data to the test task via the ITM and view messages sent from the error task.

You can also view the state of the RTX tasks by opening the Debug\OS Support\RTX Tasks and System window. The event viewer also provides a task trace, which is very useful for seeing the loading on the CPU and the interaction between the different tasks.



ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Load
255	os_idle_demon	0	Ready				32%
6	testTask	1	Running				0%
5	errorHandler	1	Wait_OR		0x0000	0xFFFF	32%
4	Task3	10	Wait_OR		0x0000	0x0001	32%
2	Task2	10	Wait_MBX				36%
1	Task1	10	Wait_DLY	5			32%

To use the ITM test task, make sure the debug (printf) viewer is the active window. Then the various test functions can be triggered by typing the characters 0–9 on the PC keyboard. The test task provides the following functions.

1—Halt Thread1

2–6—Inject a message into the Thread2 mail queue to switch the LEDs

7—Inject a message with faulty parameters

8—Start Thread1

9—Stop Thread2

0—Start Thread2

Here if you stop Thread2 it will no longer read messages from the mail queue. If Thread1 is running it will continue to write into the mail queue. This will cause the queue to overflow and this will be reported by the error task. Without this kind of error trapping, it is likely that you would end up on the hard fault exception vector and have no idea what had gone wrong with the program. Once the project has been fully developed, the test_functions.c module can be removed and the error task can be modified to take remedial action if an error occurs. This could be logging fault codes to flash, running code to try and recover the error, or worst case resetting the processor.

Instruction Trace with the ETM

The Cortex-M3 and Cortex-M4 may have an optional debug module fitted by the silicon manufacturer when the microcontroller is designed. The ETM is a third trace unit that provides an instruction trace as the application code is executed on the Cortex-M processor. The ETM is normally fitted to higher end Cortex-M3- and Cortex-M4-based microcontrollers. When selecting a device it will be listed as a feature of the microcontroller in the datasheet.

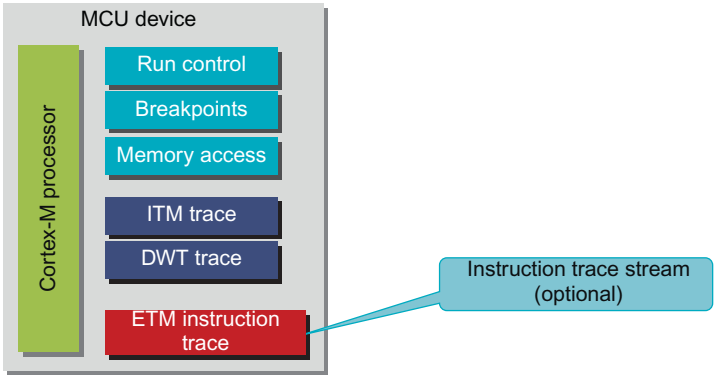


Figure 8.9
The Cortex-M3 and Cortex-M4 may optionally be fitted with a third trace unit. The ETM supports instruction trace. This allows you to quickly find complex bugs. The ETM also enables code coverage and performance analysis tools which are essential for software validation.

The ETM trace pipe requires four additional pins that are brought out to a larger 20-pin socket that incorporates the SW debug pins and the ETM trace pins. Standard JTAG/CoreSight debug tools do not support the ETM trace channel so you will need a more sophisticated debug unit.

Figure 8.10
An ETM trace unit provides all the features of the standard hardware debugger plus instruction trace.

At the beginning of this chapter, we looked at various debug methods that have been used historically with small microcontrollers. For a long time, the only solution that would provide any kind of instruction trace was the in-circuit emulator. The emulator hardware would capture each instruction executed by the microcontroller and store it in an internal trace buffer. When requested, the trace could be displayed within the PC debugger as assembly or a high-level language, typically C. However, the trace buffer had a finite size and it was only possible to capture a portion of the executed code before the trace buffer was full. So while the trace buffer was very useful, it had some serious limitations and took some experience to use correctly. In contrast, the ETM is a streaming trace that outputs compressed trace information. This information can be captured by a CoreSight trace tool; the more sophisticated units will stream the trace data directly to the hard drive of the PC without the need to buffer it within the debugger hardware.

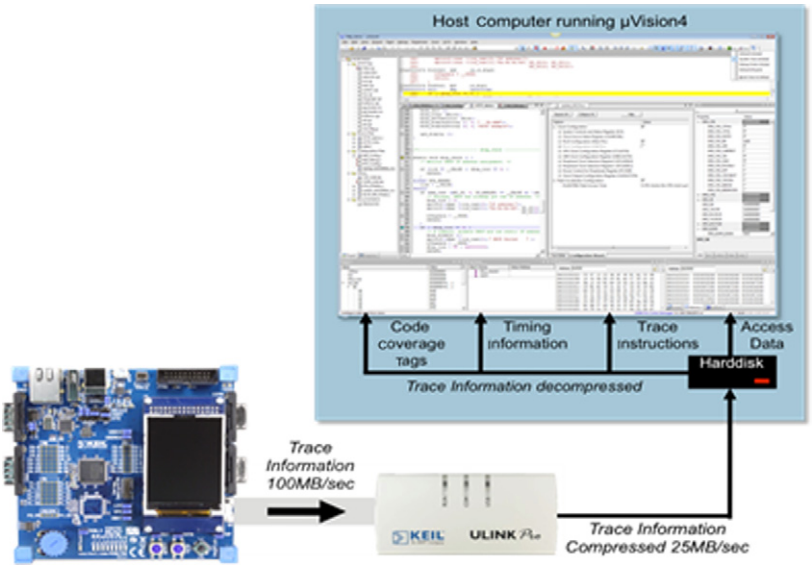


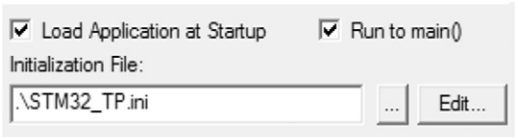
Figure 8.11
A "streaming" trace unit is capable of recording every instruction directly onto the hard drive of the PC. The size of the trace buffer is only limited by the size of your hard disk.

This streaming trace allows the debugger software to display 100% of the instructions executed along with execution times. Along with a trace buffer that is only limited by the size of the PC hard disk it is also possible to analyze the trace information to provide accurate code coverage and performance analysis information.

Exercise: Using the ETM Trace

Open the project in exercises\ETM.

Open the Options for Target\Debug menu.

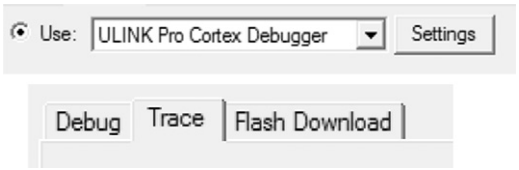


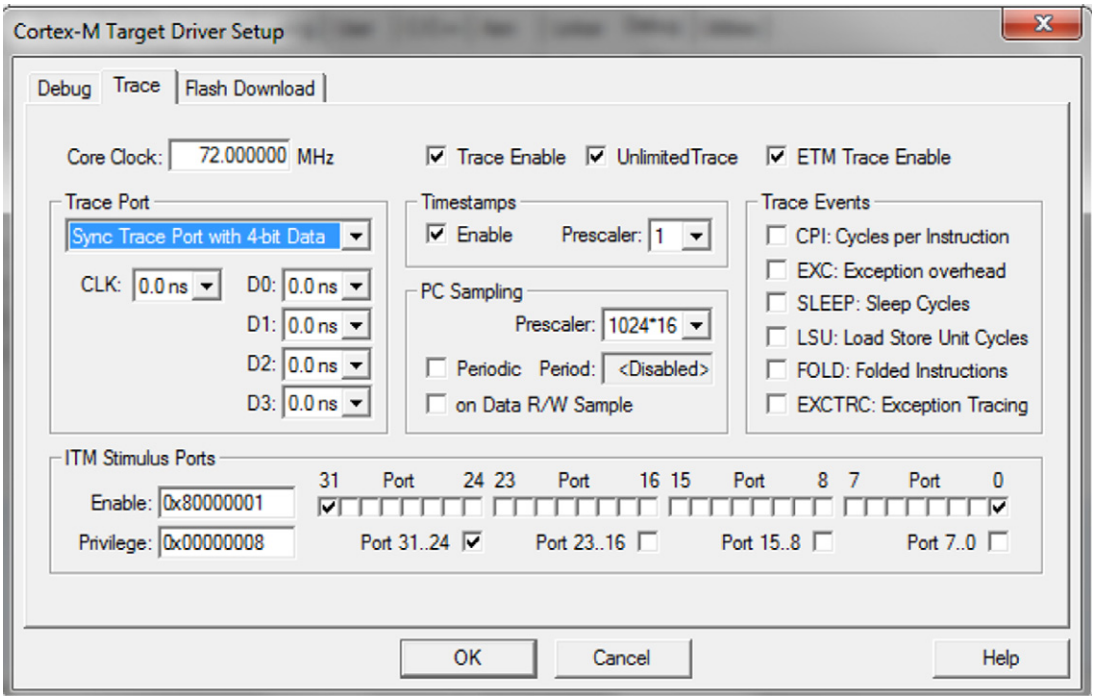
Open the STM32_TP.ini initialization file.

This is the same script file that was used with the standard Ulink2 debugger. This time the TRACE_MODE has been set for 4-bit synchronous trace data. This will enable the ETM trace pipe and switch the external microcontroller pins from GPIO to debug pins.

Now press the ULINK Pro Settings button.

Select the Trace tab.





When the ULINK2 trace tool is connected, you have the option to enable the ETM trace. The unlimited trace option allows you to stream every instruction executed to a file on your PC hard disk. The trace buffer is then only limited by the size of the PC hard disk. This makes it possible to trace the executed instructions for days if necessary, yes days.

Click OK to quit back to the μVision editor.

Start the debugger.

Now the debugger has an additional instruction trace window.

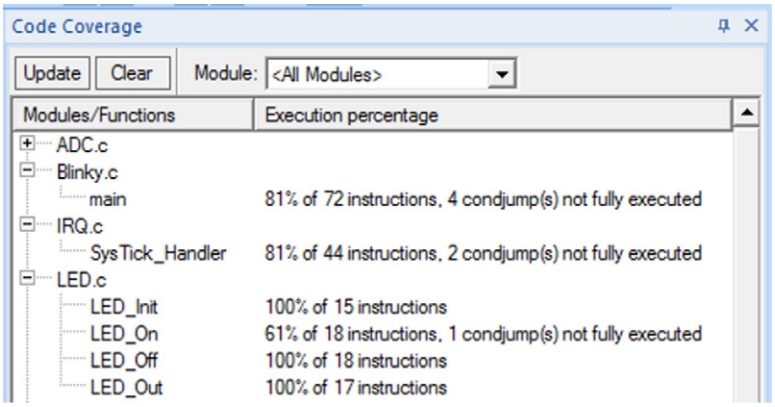
Trace Data

Display: Execution - All | in All

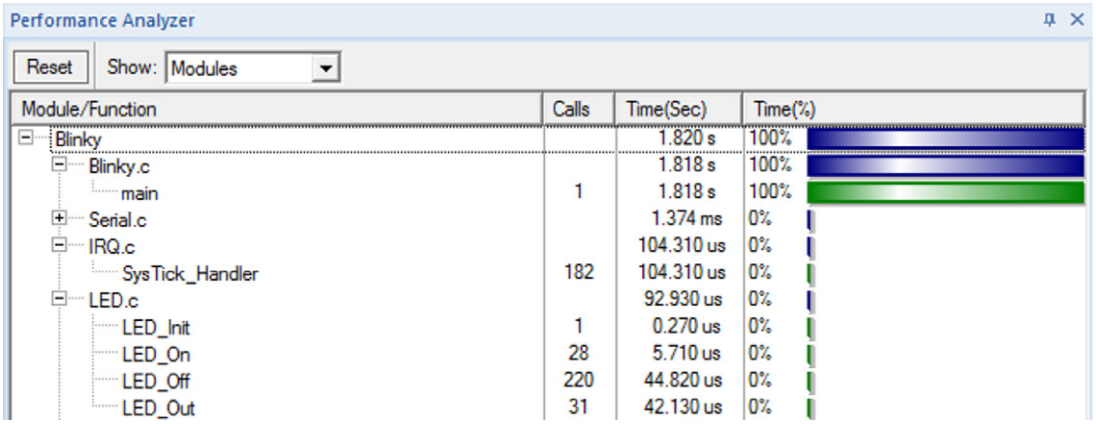
Nr.	Time	Address	Opcode	Instruction	Src Code
32,739	0.038 093 547 s	0x00000684	B168	CBZ r0,0x000006A2	
32,740	0.038 093 577 s	0x000006A2	EA950006	EORS r0,r5,r6	if (ad_val ^ ad_val_) { /...
32,741	0.038 093 587 s	0x000006A6	D005	BEQ 0x000006B4	
32,742	0.038 093 617 s	0x000006B4	480D	LDR r0,[pc,#52] ; @0x000006EC	if (clock_1s) {
32,743	0.038 093 637 s	0x000006B6	7800	LDRB r0,[r0,#0x00]	
32,744	0.038 093 657 s	0x000006B8	B130	CBZ r0,0x000006C8	
32,745	0.038 093 687 s	0x000006C8	E7DA	B 0x00000680	while (1) { /* Lo...
32,746	0.038 093 717 s	0x00000680	4815	LDR r0,[pc,#84] ; @0x000006D8	if (AD_done) { /* I...
32,747	0.038 093 737 s	0x00000682	7800	LDRB r0,[r0,#0x00]	
32,748	0.038 093 757 s	0x00000684	B168	CBZ r0,0x000006A2	
32,749	0.038 093 787 s	0x000006A2	EA950006	EORS r0,r5,r6	if (ad_val ^ ad_val_) { /...
32,750	0.038 093 797 s	0x000006A6	D005	BEQ 0x000006B4	
32,751	0.038 093 827 s	0x000006B4	480D	LDR r0,[pc,#52] ; @0x000006EC	if (clock_1s) {
32,752	0.038 093 847 s	0x000006B6	7800	LDRB r0,[r0,#0x00]	
32,753	0.038 093 867 s	0x000006B8	B130	CBZ r0,0x000006C8	

In addition to the trace buffer the ETM also allows us to show the code coverage information that was previously only available in the simulator.

```
60
61      /* AD converter input
62      if (AD_done) {
63        AD_done = 0;
64
65        ad_avg += AD_last << 8;
66        ad_avg ++;
67        if ((ad_avg & 0xFF) == 0x10) {
68          ad_val = (ad_avg >> 8) >> 4;
69          ad_avg = 0;
70        }
71      }
72
73      if (ad_val ^ ad_val_) {
74        ad_val_ = ad_val;
75
76        sprintf(text, "0x%04X", ad_val);
```



Similarly, timing information can be captured and displayed alongside the C code or as a performance analysis report.



System Control Block Debug Support

The CoreSight debugger interface allows you to control execution of your application code and examine values in the memory and peripheral registers. Combined with the various trace units, this provides you with a powerful debug system for normal program development. However, as we saw in Chapter 3, the Cortex-M processors have up to four fault exceptions that will be triggered if the application code makes incorrect use of the Cortex processor or the microcontroller hardware.

Table 8.2: Fault Exceptions

Fault Exception	Priority	Cortex Processor
Hard fault	−1	Cortex-M0, Cortex-M0+ , Cortex-M3, Cortex-M4
Bus fault	Programmable	Cortex-M3, Cortex-M4
Usage fault	Programmable	Cortex-M3, Cortex-M4
Memory manager fault	Programmable	Cortex-M3, Cortex-M4 (Optional)

When this happens your program will be trapped on the default fault handler in the startup code. It can be very hard to work out how you got there. If you have an instruction trace tool, you can work this out in seconds. If you do not have access to instruction trace then resolving a runtime crash can take a long, long time. In this section, we will look at configuring the fault exceptions and then look at how to track back to the source of a fault exception.

The behavior of the fault exceptions can be configured by registers in the system control block. The key registers are listed below.

Table 8.3: Fault Exception Configuration Registers

Register	Processor	Description
Configuration and control	M3, M4	Enable additional fault exception features
System handler control and state	M3, M4	Enable and pending bits for fault exceptions
Configurable fault status register	M3, M4	Detailed fault status bits
Hard fault status	M3, M4	Reports a hard fault or fault escalation
Memory manager fault address	M3, M4	Address of location that caused the memory manager fault
Bus fault address	M3, M4	Address of location that caused the bus fault

When the Cortex-M processor comes out of reset, only the hard fault handler is enabled. If a usage, bus, or memory manager fault is raised and the exception handler for these faults is not enabled, then the fault will "escalate" to a hard fault. The hard fault status register provides two status bits that indicate the source of the hard fault.

Table 8.4: Hard Fault Status Register

Name	Bit	Use
FORCED	30	Reached the hard fault due to fault escalation
VECTTBL	1	Reached the hard fault due to a faulty read of the vector table

The system handler control and state register contains enable, pending, and active bits for the bus usage and memory manager exception handlers. We can also configure the behavior of the fault exceptions with the configuration and control register.

Table 8.5: Configuration and Control Register

Name	Bit	Use
STKALIGN	9	Configures 4- or 8-byte stack alignment
BFHFMIGN	8	Disables data bus faults caused by load and store instructions
DIV_0_TRP	4	Enables a usage fault for divide by zero
UNALIGNTRP	3	Enables a usage fault for unaligned memory access

The divide by zero can be a useful trap to enable, particularly during development. The remaining exceptions should be left disabled unless you have a good reason to switch them on. When a memory manager fault exception occurs, the address of the instruction that attempted to access a prohibited memory region will be stored in the memory fault address register, similarly when a bus fault is raised the address of the instruction that caused the fault will be stored in the bus fault address register. However, under some conditions, it is not always possible to write the fault addresses to these registers. The configurable fault status register contains an extensive set of flags that report the Cortex-M processor error conditions that help you track down the cause of a fault exception.

Tracking Faults

If you have arrived at the hard fault handler, first check the hard fault status register. This will tell you if you have reached the hard fault due to fault escalation or a vector table read error. If there is a fault escalation, next check the system handler control and state register to see which other fault exception is active. The next port of call is the configurable fault status register. This has a wide range of flags that report processor error conditions.

Table 8.6: Configurable Fault Status Register

Name	Bit	Use
DIVBYZERO	25	Divide by zero error
UNALIGNED	24	Unaligned memory access
NOCP	19	No processor present
INVPC	18	Invalid PC load
INVSTATE	17	Illegal access to the execution program status register (EPSR)
UNDEFINSTR	16	Attempted execution of an undefined instruction
BFARVALID	15	Address in bus fault address register is valid
STKERR	12	Bus fault on exception entry stacking
UNSTKERR	11	Bus fault on exception exit on stacking
IMPRECISERR	10	Data bus error. Error address not stacked
PRECISERR	9	Data bus error. Error address stacked
IBUSERR	8	Instruction bus error
MMARVALID	7	Address in the memory manager fault address register is valid
MSTKERR	4	Stacking on exception entry caused a memory manager fault
MUNSTKERR	3	Stacking on exception exit caused a memory manager fault
DACCVIOL	1	Data access violation flag
IACCVIOL	0	Instruction access violation flag

When the processor fault exception is entered, a normal stack frame is pushed onto the stack. In some cases, the bus frame will not be valid and this will be indicated by the flags in the configurable fault status register. When a valid stack frame is pushed, it will contain the PC address of the instruction that generated the fault. By decoding the stack frame, you can retrieve this address and locate the problem instruction. The system control block provides a memory and bus fault address register that depending on the cause of the error may hold the address of the instruction that caused the error exception.

Exercise: Processor Fault Exceptions

Open the project in c:\exercises\fault exception

In this project, we will generate a fault exception and look at how it is handled by the NVIC and how it is possible to trace the fault back to the line of code that caused the problem.

```
volatile uint32_t op1;
int main (void)
{
int op2 = 0x1234,op3 = 0;
  SCB->CCR = 0x0000010;  //enable divide by zero usage fault
  op1 = op2/op3;  //perform a divide by zero to generate a usage exception
while(1);
}
```

The code first enables the divide by zero usage fault and then divides by zero to cause the exception.

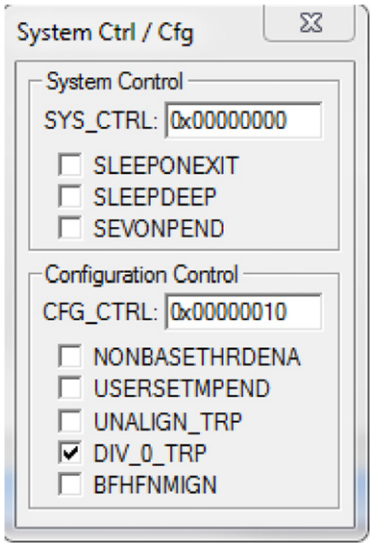
Build the code and start the debugger.

Set a breakpoint on the line of code that contains the divide instruction.

```
10   SCB->SHCSR = 0x00060000;
11   SCB->CCR = 0x0000010;
12   op1 = op2/op3;
```

Run the code until it hits this breakpoint.

Open the Peripherals\Core Peripherals\System Control and Configuration window and check that the divide by zero trap has been enabled.

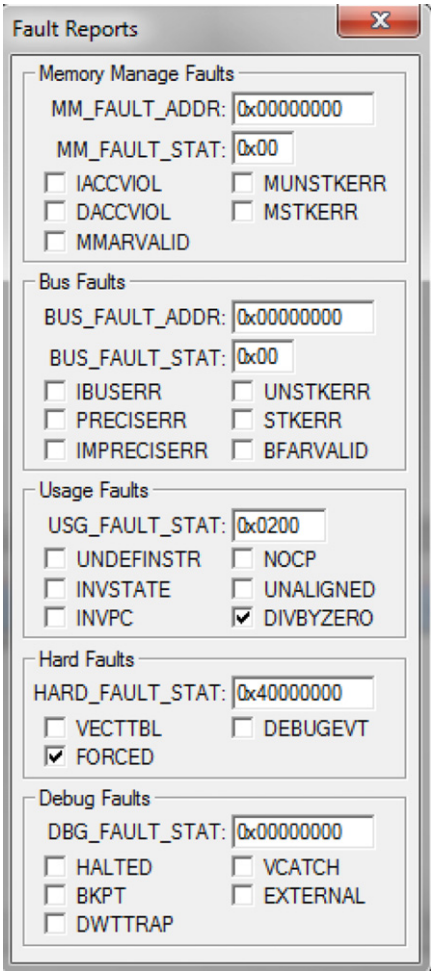


Single step the divide instruction to cause a divide by zero error.

```
128  HardFault_Handler\
129                  PROC
130                  EXPORT  HardFault_Handler         [WEAK]
131                  B       .
132                  ENDP
```

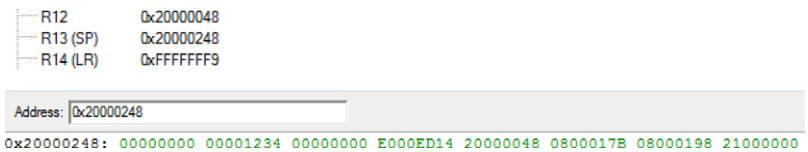
A usage fault exception will be raised. We have not enabled the usage fault exception vector so the fault will elevate to a hard fault.

Open the Peripherals\Core Peripherals\Fault Reports window.



This window shows that the hard fault has been forced by another fault exception. Also the divide by zero flag has been in the usage fault status register.

In the register window, read the contents of R13, the main stack pointer, and open a memory window at this location.



Read the PC value saved in the stack frame and open the disassembly window at this location.

```
0x08000198 FB91F2F0  SDIV     r2,r1,r0
0x0800019C 4B08      LDR      r3,[pc,#32]   ; @0x080001C0
```

This takes us back to the SDIV instruction that caused the fault.

Exit the debugger and add the line of code given below to the beginning of the program.

```
SCB->SHCSR = 0x00060000;
```

This enables the usage fault exception in the NVIC.

Now add a C level usage fault exception handler.

```
void UsageFault_Handler (void)
{
  error_address = (uint32_t *)(__get_MSP());  // load the current base address of the
stack pointer
  error_address = error_address + 6;  // locate the PC value in the last stack frame
  while(1);
}
```

Build the project and start the debugger.

Set a breakpoint on the while loop in the exception function.

```
19  void UsageFault_Handler (void)
20  {
21    error_address = (uint32_t *)(__get_MSP());
22    error_address = error_address + 6;
23    while(1);
24  }
```

Run the code until the exception is raised and the breakpoint is reached.

When a usage fault occurs this exception routine will be triggered. It reads the value stored in the stack pointer and extracts the value of the PC stored in the stack frame.

CMSIS SVD

The CMSIS SVD format is designed to provide silicon manufacturers a method of creating a description of the peripheral registers in their microcontrollers. This description can be passed to third party tool manufacturers so that compiler includes files and debugger peripheral view windows can be created automatically. This means that there will be no lag

in software development support when a new family of devices is released. As a developer you will not normally need to work with these files but it is useful to understand how the process works so that you can fix any errors that may inevitably occur. It is also possible to create your own additional peripheral debug windows. This would allow you to create a view of an external memory mapped peripheral or provide a debug view of a complex memory object.

When the silicon manufacturer develops a new microcontroller, they also create an XML description of the microcontroller registers. A conversion utility is then used to create a binary version of the file that is used by the debugger to automatically create the peripheral debug windows.

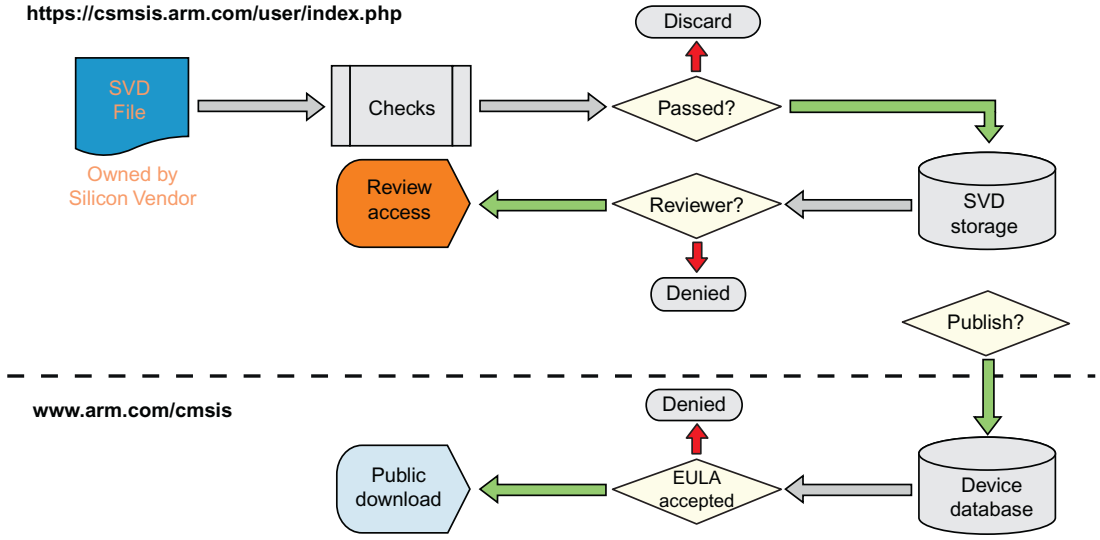


Figure 8.12
ARM has created a repository of "SVD" files. This enables tools suppliers to have support for new devices as they are released.

Alongside the XML description definition, ARM has introduced a submission and publishing system for new SVD files. When a silicon manufacturer designs a new microcontroller, its system description file is submitted via the CMSIS Web site. Once it has been reviewed and validated, it is then published for public download on the main ARM Web site.

Exercise: CMSIS SVD

In this exercise, we will take a look at a typical system viewer file to make an addition and to rebuild the debugger file, and then check the updated version in the debugger.

For this exercise, you will need an XML editor. If you do not have one, then download a trial or free tool from the Internet.

Open your web browser and go to www.arm.com/cmsis.

If you do not have a user account on the ARM Web site, you will need to create one and login.

On the CMSIS page, select the CMSIS-SVD tab.

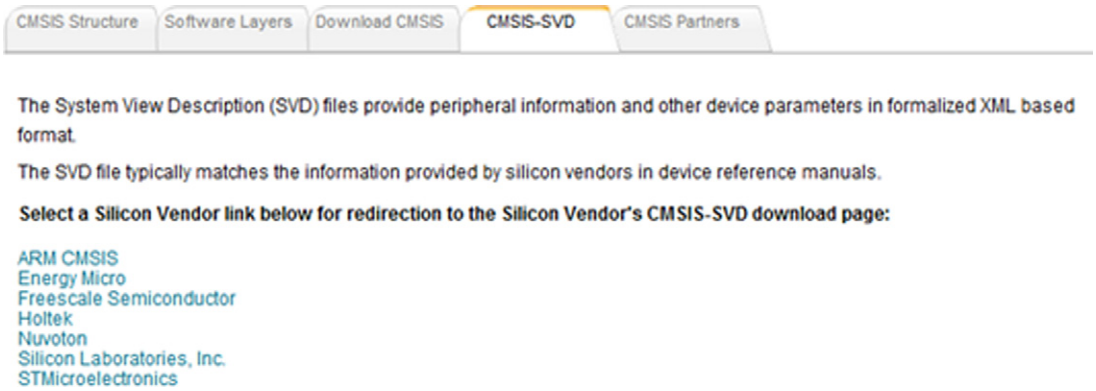
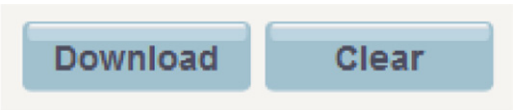


Figure 8.13
The CMSIS Web site has public links to the current CMSIS specification and the CMSIS-SVD repository.

Select the STMicroelectronics link.

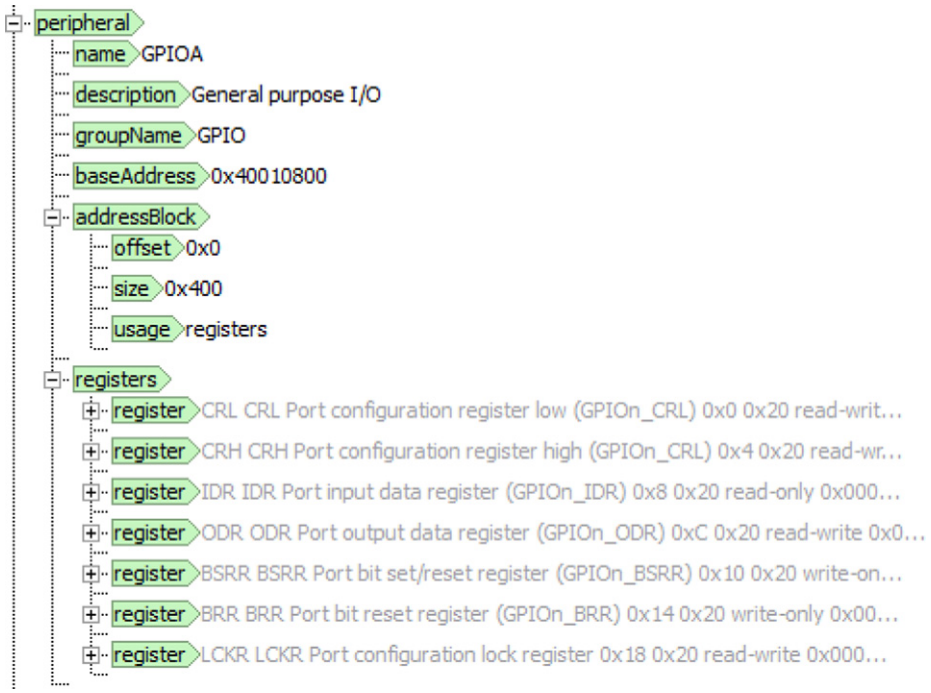
In the ST window, select the SVD download that supports the STM32F103RB.

Then click the download button at the bottom of the page.

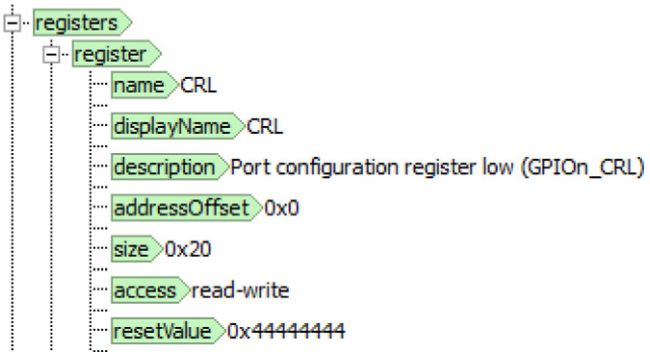


Unzip the compressed file into a directory.

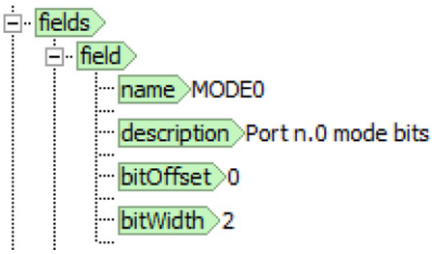
Open the STM32F103xx.svd file with the XML editor.



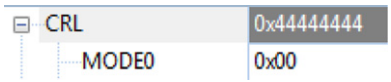
Each of the peripheral windows is structured as a series of XML tags that can be edited or a new peripheral pane can be added. This would allow you to display the registers of an external peripheral interfaced onto an external bus. Each peripheral window starts with a name, description, and group name followed by the base address of the peripheral.



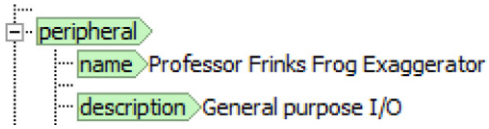
Once the peripheral details have been defined, a description of each register in the peripheral is added. This consists of the register name and description of its offset from the peripheral base address along with its size in bits, access type, and reset value.



It is also possible to define bit fields within the register. This allows the debugger window to expand the register view and display the contents of the bit field.



Make a small change to the XML file and save the results.



Generate the SFR file by using the SVDConv.exe utility.

This utility is located in c:\keil\arm\cmsis\svd.

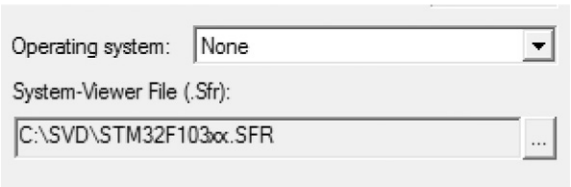
The convertor must be invoked with the following command line:

SVDConv STM32F103xx.svd–generate = sfr

This will create STM32F103xx.SFR.

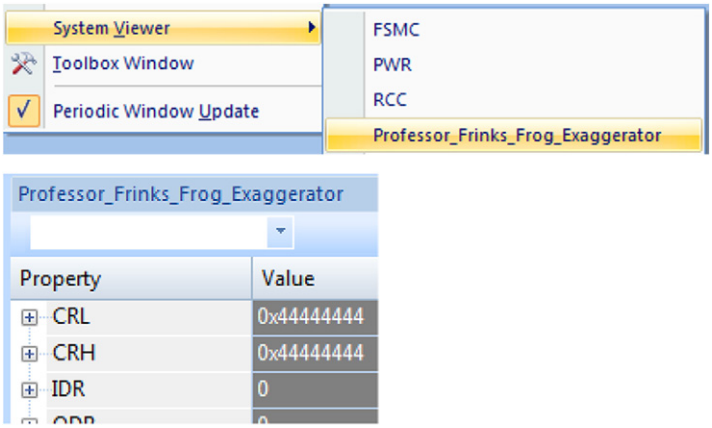
Now start μVision and open a previous exercise.

Open the Options for Target menu and select STM32F103xx.sfr as the system viewer file.



Build the project and start the debugger.

Open the View\System Viewer selection and view the updated peripheral window.



End of book craziness aside, it is useful to know how to add and modify the peripheral windows so you can correct mistakes or add in your own specific debug support.

CMSIS DAP

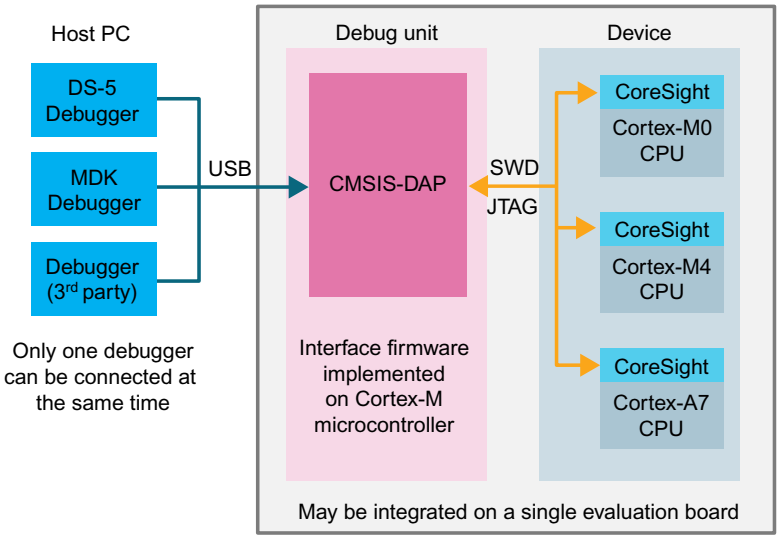


Figure 8.14
The CMSIS DAP specification is designed to support interoperability between different debugger hardware and debugger software.

The CMSIS DAP specification defines the interface protocol between the CoreSight debugger hardware and the PC debugger software. This creates a new level of

interoperability between different vendors' software and hardware debuggers. The CMSIS DAP firmware is designed to operate on even the most basic debugger hardware. This allows even the most basic evaluation modules to host a common debug interface that can be used with any CMSIS-compliant tool chain.

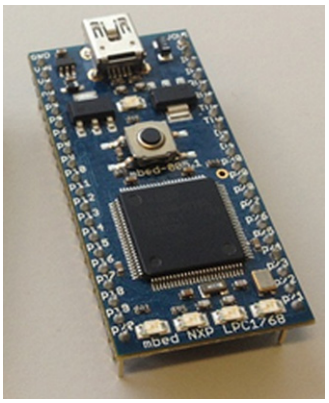


Figure 8.15
The MDED module is the first to support the CMSIS DAP specification.

The CMSIS DAP specification is designed to support a USB interface between the target hardware and the PC. This allows many simple modules to be powered directly from the PC USB port.

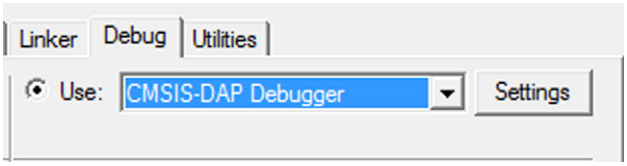


Figure 8.16
The CMSIS DAP driver must be selected in the debugger menu.

The CMSIS DAP interface can be selected in the debug menu in place of the proprietary Ulink2. The configuration options are essentially the same as the Ulink2 but the options available will depend on the level of firmware implemented by the device manufacturer. The CMSIS DAP specification supports all of the debug features found in the CoreSight debug architecture including the Cortex-M0+ microtrace buffer (MTB).

Cortex-M0+ MTB

While the ETM is available for the Cortex-M3 and Cortex-M4, no form of instruction trace is currently available for the Cortex-M0. However, the Cortex-M0+ has a simple form of instruction trace buffer called the MTB. The MTB uses a region of internal SRAM that is allocated by the developer. When the application code is running, a trace of executed

instructions is recorded into this region. When the code is halted, the debugger can read the MTB trace data and display the executed instructions. The MTB trace RAM can be configured as a circular buffer or a one-shot recording. While this is a very limited trace, the circular buffer will allow you to see "what just happened" before the code halted. The one-shot mode can be triggered by the hardware breakpoints to start and stop allowing you to track down more elusive bugs.

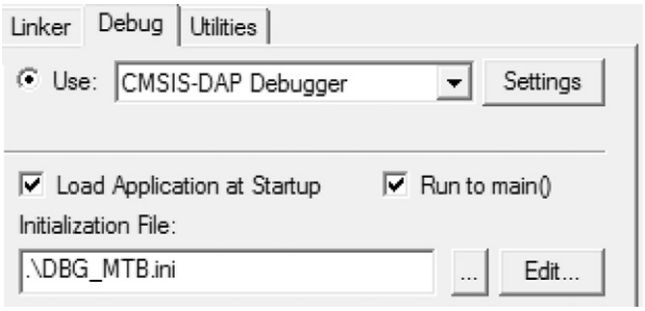
Exercise: MTB

This exercise is based on the Freescale Freedom board for the MKL25Z microcontroller. This was the first microcontroller available to use the Cortex-M0+ .

Connect the Freedom board via its USB cable to the PC.

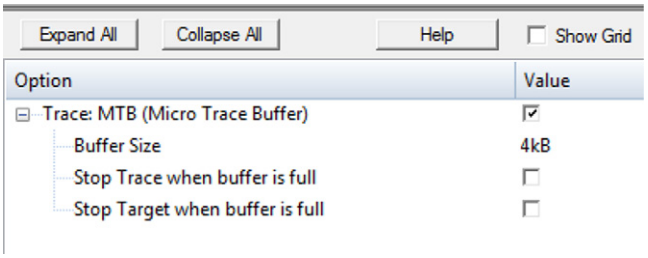
Open the project in c:\exercises\CMSIS DAP.

Open the Options for Target\Debug menu.



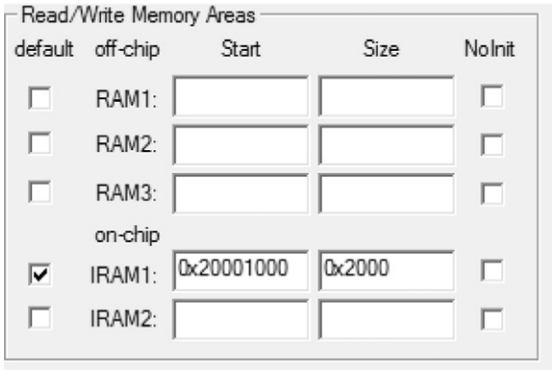
Here the CMSIS DAP interface is selected along with an initializing file for the MTB.

The initializing script file has a wizard that allows you to configure the size and MTB.



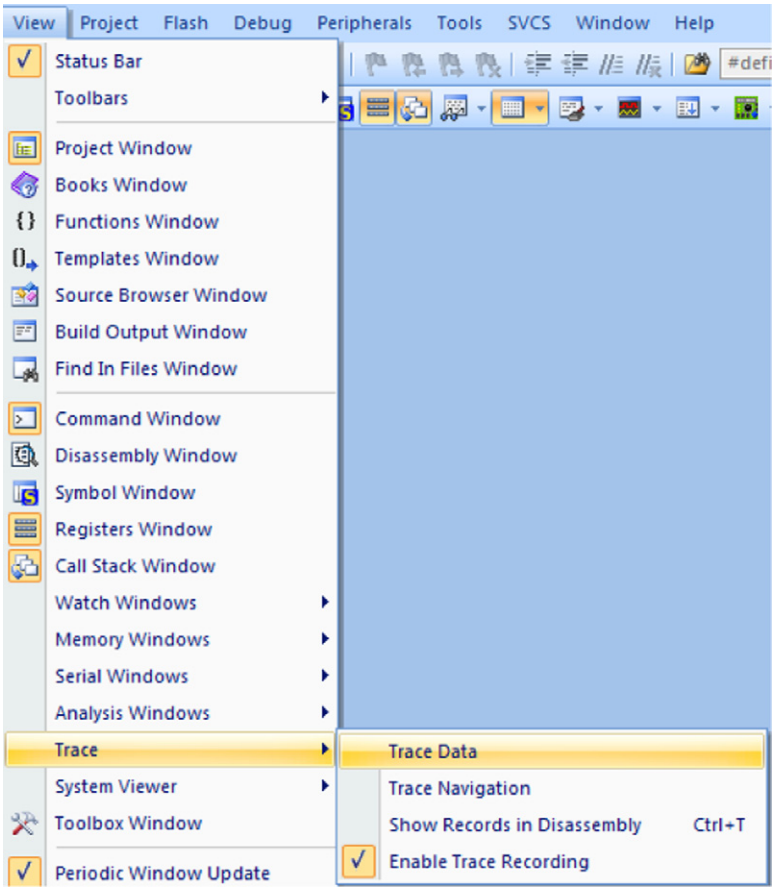
Here we can select the amount of internal SRAM that is to be used for the trace. It is also possible to configure different debugger actions when the trace is full: either halt trace recording or halt execution on the target.

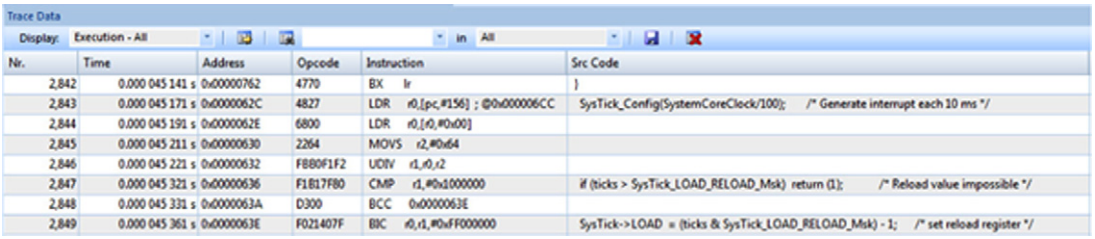
By default the MTB is located at the start of the internal SRAM (0x20000000). So it is necessary to offset the start of user SRAM by the size of memory allocated to the MTB. This is done in the Options for Target\Target dialog.



Now start the debugger and execute the code.

Halt the debugger and open the trace window.





While this is a limited trace buffer, it can be used by very low-cost tools and provides a means of tracking down runtime bugs that would be time consuming to find any other way.

Debug Features Summary

Table 8.7: Summary of Cortex-M Debug Features

Feature	Cortex-M0 Cortex-M0+	Cortex-M3 Cortex-M4
Debug interface	Legacy JTAG or SW	Legacy JTAG or SW
"On the fly" memory access	Yes	Yes
Hardware breakpoint	4	6 Instruction + 2 Literal
Data watchpoint	2	4
Software breakpoint	Yes	Yes
ETM instruction trace	No	Yes (optional)
Data trace	No	Yes (optional)
Instrumentation trace	No	Yes
SW viewer	No	Yes
MTB	Yes (Cortex-M0+ only)	No