CHAPTER 5

Advanced Architecture Features

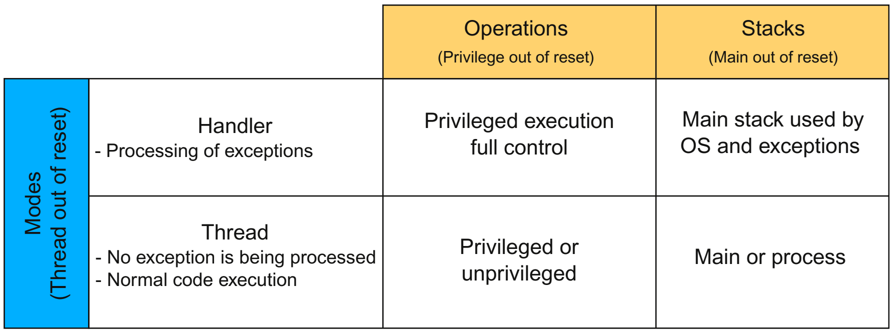
Introduction

In the last few chapters, we have covered most of what you need to know to develop a Cortex-M-based microcontroller. In the remaining chapters in this book, we will look at some of the more advanced features of the Cortex-M processors. In this chapter, we will look at the different operating modes built into each of the Cortex-M processors and some additional instructions that are designed to support the use of an RTOS. We will also have a look at the optional MPU, which can be fitted to the Cortex-M3, Cortex-M4, and Cortex-M0+, and how this can partition the memory map. This provides controlled access to different regions of memory depending on the running processor mode. To round the chapter off, we will have a look at the bus interface between the Cortex-M processor and the microcontroller system.

Cortex Processor Operating Modes

When the Cortex-M processor comes out of reset, it is running in a simple "flat" mode where all of the application code has access to the full processor address space and unrestricted access to the CPU and NVIC registers. While this is OK for many applications, the Cortex-M processor has a number of features that let you place the processor into a more advanced operating mode that is suitable for high-integrity software and also supports an RTOS.

As a first step to understanding the more advanced operating modes of the Cortex-M processor, we need to understand its operating modes. The CPU can be running in two different modes, thread mode and handler mode. When the processor is executing background code (i.e., noninterrupt code), it is running in thread mode. When the processor is executing interrupt code, it is running in handler mode.



Modes (Thread out of reset)		Operations (Privilege out of reset)	Stacks (Main out of reset)
	Handler - Processing of exceptions	Privileged execution full control	Main stack used by OS and exceptions
	Thread - No exception is being processed - Normal code execution	Privileged or unprivileged	Main or process

Figure 5.1
Each Cortex-M processor has two execution modes—(interrupt) handler and (background) thread. It is possible to configure these modes to have privileged and unprivileged access to memory regions. It is also possible to configure a two-stack operating mode.

When the processor starts to run out of reset, there is no operating difference between thread and handler modes. Both modes have full access to all features of the CPU. This is known as privileged mode. By programming the Cortex-M processor CONTROL register, it is possible to place the thread mode in unprivileged mode by setting the thread privilege level bit.

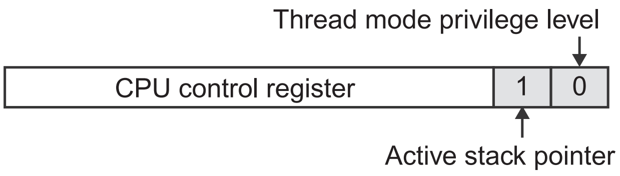


Figure 5.2
The CONTROL register is a CPU register that can only be accessed by the MRS and MSR instructions. It contains two bits that configure the thread mode privilege level and activation of the process.

In unprivileged mode, the MRS, MSR, and Change Processor State (CPS) instructions are disabled for all special CPU registers, except the APSR. This prevents the Cortex-M processor from accessing the CONTROL, FAULTMASK, and PRIMASK registers, and the program status register (except the APSR). In unprivileged mode, it is also not possible to access the systick timer registers, NVIC, or the system control block. These limits attempt to prevent unprivileged code from accidentally disturbing the operation of the Cortex-M processor. If thread mode has been limited to unprivileged access, it is not possible to clear the thread mode privilege level bit even if the CPU is running in handler mode with privilege access. Once the TLP bit has been set, the application code running in thread mode can no longer influence the operation of the Cortex-M processor. When the processor responds to an exception or interrupt, it moves into handler mode, which always executes code in privileged mode regardless of the contents of the CONTROL register. The CONTROL register also contains an additional bit, the active stack pointer selection (ASPEL). Setting this bit enables an additional stack called the process stack. The CONTROL register is a CPU register rather than a memory-mapped register and can only be accessed by the MRS and MSR instructions. The CMSIS core specification provides dedicated functions to read and write to the CONRTOL register.

```
    void  __set_CONTROL(uint32_t value);
uint32_t  __get_CONTROL(void);
```

The process stack is a banked R13 stack pointer, which is used by code running in thread mode. When the Cortex-M processor responds to an exception it enters handler mode. This causes the CPU to switch stack pointers. This means that the handler mode will use the MSP while thread mode uses the process stack pointer.

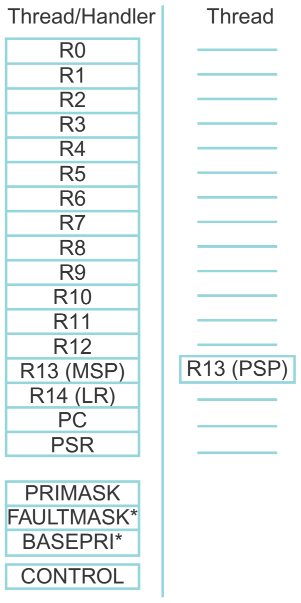


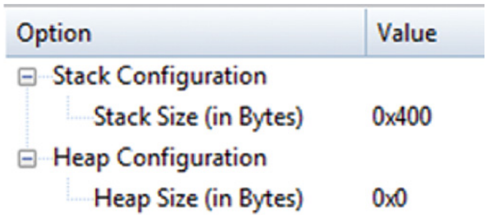
Figure 5.3
At reset R13 is the MSP and is automatically loaded with the initial stack value. The CPU CONTROL register can be used to enable a second banked R13 register. This is the process stack that is used in thread mode. The application code must load an initial stack value into this register.

As we have seen in Chapter 3, at reset the MSP will be loaded with the value stored in the first 4 bytes of memory. However, the application stack is not automatically initialized and must be set up by the application code before it is enabled. Fortunately, the CMSIS core specification contains a function to configure the process stack.

```
void  __set_PSP(uint32_t TopOfProcStack);
uint32_t  __get_PSP(void);
```

So if you have to manually set the initial value of the process stack, what should it be? There is not an easy way to answer this, but the compiler produces a report file that details the static calling tree for the project. This file is created each time the project is built and is called <project name>.htm. The report file includes a value for the maximum stack usage and a calling tree for the longest call chain.

```
Stack_Size  EQU  0x00000400
  AREA  STACK, NOINIT, READWRITE, ALIGN = 3
Stack_Mem  SPACE  Stack_Size
__initial_sp
```



Option	Value
⊟ Stack Configuration	
Stack Size (in Bytes)	0x400
⊟ Heap Configuration	
Heap Size (in Bytes)	0x0

Figure 5.4
The stack size allocated to the MSP is defined in the startup code and can be configured through the configuration wizard.

This calling tree is likely to be for background functions and will be the maximum value for the process stack pointer. This value can also be used as a starting point for the MSP.

Exercise: Stack Configuration

In this exercise, we will have a look at configuring the operating mode of the Cortex-M processor so the thread mode is running with unprivileged access and uses the process stack pointer.

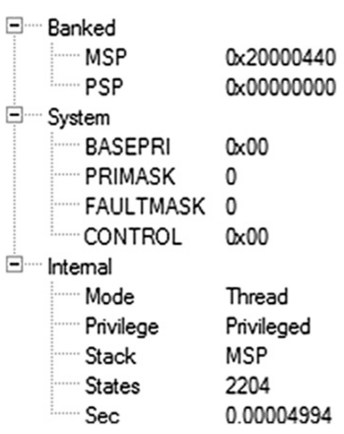
Open the project in the exercise\operating mode folder.

Build the code and start the debugger.

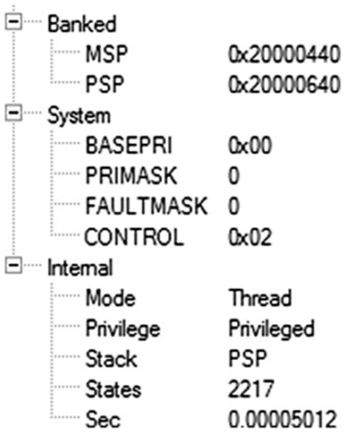
This is a version of the blinky project we used earlier with some code added to configure the processor operating mode. The new code includes a set of #defines.

```
#define USE_PSP_IN_THREAD_MODE       (1<<1)
#define THREAD_MODE_IS_UNPRIVILIGED   1
#define PSP_STACK_SIZE                0x200
```

The first two declarations define the location of the bits that need to be set in the control register to enable the process stack and switch the thread mode into unprivileged access. Then, we define the size of the process stack space. At the start of main, we can use the CMSIS functions to configure and enable the process stack. We can also examine the operating modes of the processor in the register window.



```
Banked
   MSP         0x20000440
   PSP         0x00000000
System
   BASEPRI     0x00
   PRIMASK     0
   FAULTMASK   0
   CONTROL     0x00
Internal
   Mode        Thread
   Privilege   Privileged
   Stack       MSP
   States      2204
   Sec         0.00004994
```

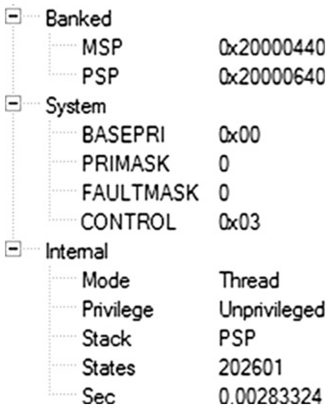


```
Banked
   MSP         0x20000440
   PSP         0x20000640
System
   BASEPRI     0x00
   PRIMASK     0
   FAULTMASK   0
   CONTROL     0x02
Internal
   Mode        Thread
   Privilege   Privileged
   Stack       PSP
   States      2217
   Sec         0.00005012
```

```
_initalPSPValue = __get_MSP() + PSP_STACK_SIZE;
_set_PSP(initalPSPValue);
__set_CONTROL(USE_PSP_IN_THREAD_MODE);
__ISB();
__ISB();
```

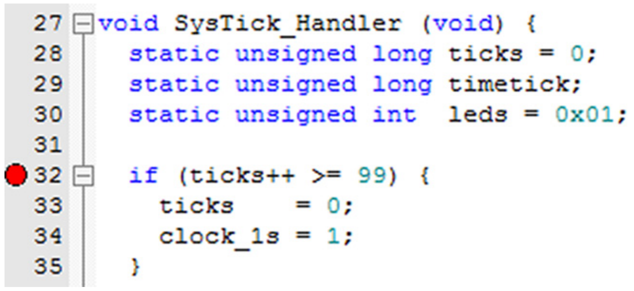
When you reach the main() function, the processor is in thread mode with full privileged access to all features of the microcontroller. Also only the MSP is being used. If you step through the three configuration lines, the code first reads the contents of the MSP. This will be at the top of the main stack space. To get the start address for the process stack, we simply add the desired stack size in bytes. This value is written by the process stack before enabling it in the control register. Always configure the stack before enabling it in case there is an active interrupt that could occur before the stack is ready. Next, we need to execute any code that needs to configure the processor before switching the thread mode to unprivileged access. The ADC_Init function accesses the NVIC to configure an interrupt, and the systick timer is also configured. Accessing these registers will be prohibited when we switch to unprivileged mode. Again an instruction barrier is used to ensure the code completes before execution continues.

```
ADC_Init();
  SysTick_Config(SystemCoreClock / 100);
    __set_CONTROL(USE_PSP_IN_THREAD_MODE
                     |THREAD_MODE_IS_UNPRIVILIGED);
    __ISB();
```

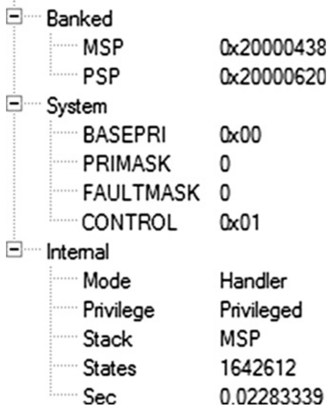


```
Banked
   MSP         0x20000440
   PSP         0x20000640
System
   BASEPRI     0x00
   PRIMASK     0
   FAULTMASK   0
   CONTROL     0x03
Internal
   Mode        Thread
   Privilege   Unprivileged
   Stack       PSP
   States      202601
   Sec         0.00283324
```

Next, set a breakpoint in the IRQ.c module, line 32. This is in the SysTick interrupt handler routine. Now run the code and it will hit the breakpoint when the systick handler interrupt is raised.



```
27 void SysTick_Handler (void) {
28   static unsigned long ticks = 0;
29   static unsigned long timetick;
30   static unsigned int  leds = 0x01;
31
32   if (ticks++ >= 99) {
33     ticks     = 0;
34     clock_1s = 1;
35   }
```



```
Banked
    MSP          0x20000438
    PSP          0x20000620
System
    BASEPRI      0x00
    PRIMASK      0
    FAULTMASK    0
    CONTROL      0x01
Internal
    Mode         Handler
    Privilege    Privileged
    Stack        MSP
    States       1642612
    Sec          0.02283339
```

Now that the processor is serving an interrupt, it has moved into interrupt handler mode with privileged access to the Cortex processor and is using the main stack.

Supervisor Call

Once configured, this more advanced operating mode provides a partition between the exception/interrupt code running in handler mode and the background application code running in thread mode. Each operating mode can have its own code region, RAM region, and stack. This allows the interrupt handler code full access to the chip without the risk that it may be corrupted by the application code. However, at some point, the application code will need to access features of the Cortex-M processor that are only available in the handler mode with its full privileged access. To allow this to happen, the Thumb-2 instruction set has an instruction called Supervisor Call (SVC). When this instruction is executed, it raises a supervisor exception that moves the processor from executing the application code in thread\unprivileged mode to an exception routine in handler/privileged mode. The SVC has its own location within the vector table and behaves like any other exception.

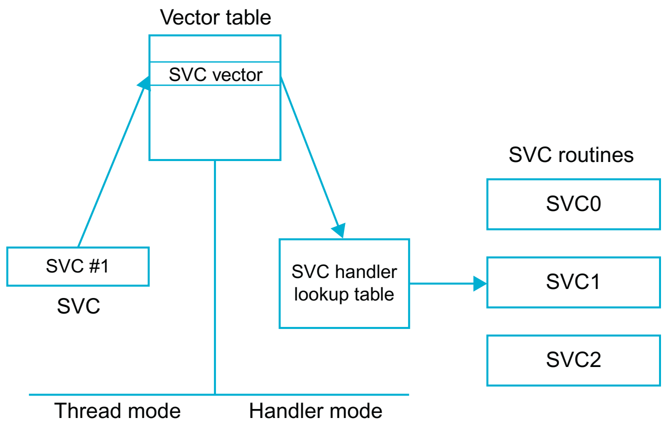


Figure 5.5
The SVC allows execution to move from unprivileged thread mode to privileged handler mode and gain full unrestricted access to the Cortex processor. The SVC instruction is used by RTOS API calls.

The supervisor instruction may also be encoded with an 8-bit value called an ordinal. When the SVC is executed, this ordinal value can be read and used as an index to call one of 256 different supervisor functions.

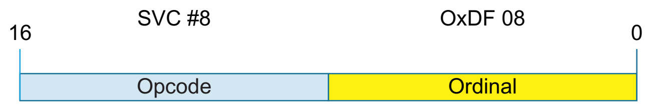
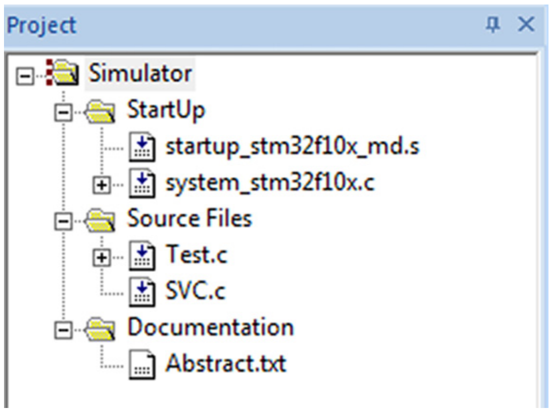


Figure 5.6
The unused portion of the SVC instruction can be encoded with an ordinal number. On entry to the SVC handler, this number can be read to determine which SVC functions to execute.

The toolchain provides a dedicated SVC support function that is used to extract the ordinal value and call the appropriate function. First, we need to find the value of the PC, which is the last value stored on the stack. First, the SVC support function reads the link register to determine the operating mode; then it reads the value of the saved PC from the appropriate stack. We can then read the memory location holding the SVC instruction and extract the ordinal value. This number is then used as an index into a lookup table to load the address of the function that is being called. We can then call the supervisor function that is executed in privileged mode before we return back to the application code running in unprivileged thread mode. This mechanism may seem an overly complicated way of calling a function, but it provides the basis of a supervisor/user split where an OS is running in privileged mode and acts as a supervisor to application threads running in unprivileged thread mode. This way the individual threads do not have access to critical processor features except by making API calls to the OS.

Exercise: SVC

In this exercise, we will look at calling some functions with the SVC instruction rather than branching to the routine as in a standard function call. Open the project in the exercise/SVC folder. First, let us have a look at the project structure.



The project consists of the standard project startup file and the initializing system file. The application source code is in the file Test.c. There is an additional source file SVC.c that provides support for handling SVC exceptions. The SVC.c file contains the SVC exception handler; this is a standard support file that is provided with the ARM compiler. We will have a closer look at its operation later. The application code in Test.c is calling two simple functions that in turn call routines to perform basic arithmetic operations.

```
int main (void) {
    test_a();
    test_t();
while(1);
}
  void test_a (void) {
    res = add (74, 27);
    res + = mul4(res);
}
void test_t (void) {
    res = div (res, 10);
    res = mod (res, 3);
}
```

Each of the arithmetic functions is designed to be called with an SVC instruction so that all of these functions run in handler mode rather than thread mode. In order to convert the arithmetic functions from standard functions to software interrupt functions, we need to change the way the function prototype is declared. The way this is done will vary between compilers, but in the ARM compiler there is a function qualifier __svc. This is used as shown below to convert the function to an SVC and allows you to pass up to four parameters and get a return value. So the add function is declared as follows:

```
int __svc(0) add (int i1, int i2);
int __SVC_0  (int i1, int i2) {
  return (i1+ i2);
}
```

The __svc qualifier defines this function as an SVC and defines the ordinal number of the function. The ordinals used must start from zero and grow upward contiguously to a maximum of 256. To enable each ordinal, it is necessary to build a lookup table in the SVC.c file.

```
; Import user SVC functions here.
  IMPORT  __SVC_0
  IMPORT  __SVC_1
  IMPORT  __SVC_2
  IMPORT  __SVC_3
```

SVC_Table

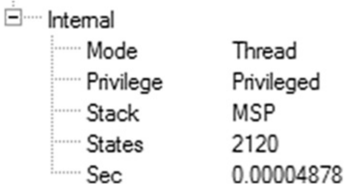
```
; Insert user SVC functions here
  DCD  __SVC_0
  DCD  __SVC_1  ;
  DCD  __SVC_2  ;
  DCD  __SVC_3  ;
```

You must import the label used for each supervisor function and then add the function labels to the SVC table. When the code is compiled, the labels will be replaced by the entry address of each function.

In the project build the code and start the simulator. Step the code until you reach line 61, the call to the add function.

The following code is displayed in the disassembly window, and in the register window we can see that the processor is running in thread mode.

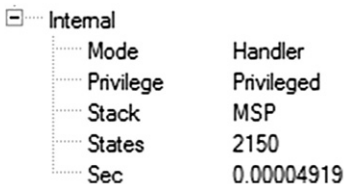
```
  61: res = add (74, 27);
0x0800038A 211B  MOVS  r1,#0x1B
0x0800038C 204A  MOVS  r0,#0x4A
0x0800038E DF03  SVC  0x03
```



```
Internal
    Mode        Thread
    Privilege   Privileged
    Stack       MSP
    States      2120
    Sec         0.00004878
```

The function parameters are loaded into the parameter passing registers R0 and R1, and the normal branch instruction is replaced by an SVC instruction. The SVC instruction is encoded with an ordinal value of 3. If you make the disassembly window the active window and step through these instructions, the SVC exception will be raised and you will enter the SVC_Handler in SVC.c. In the registers window, you can also see that the processor is now running in handler mode.

```
__asm void SVC_Handler (void) {
  PRESERVE8
    TST  LR,#4               ; Called from Handler Mode?
    MRSNE  R12,PSP           ; Yes, use PSP
    MOVEQ  R12,SP            ; No, use MSP
    LDR  R12,[R12,#24]       ; Read Saved PC from Stack
    LDRH  R12,[R12,#-2]      ; Load Halfword
    BICS  R12,R12,#0xFF00    ; Extract SVC Number
```



```
Internal
    Mode        Handler
    Privilege   Privileged
    Stack       MSP
    States      2150
    Sec         0.00004919
```

The first section of the SVC_Handler code works out which stack is in use and then reads the value of the program counter saved on the stack. The program counter value is the return address, so the load instruction deducts 2 to get the address of the SVC instruction. This will be the address of the SVC instruction that raised the exception. The SVC instruction is then loaded into R12 and the ordinal value is extracted. The code is using R12 because the ARM binary interface standard defines R12 as the "intra Procedure call scratch register"; this means it will not contain any program data and is free for use.

```
PUSH {R4,LR}                 ; Save Registers
  LDR  LR,=SVC_Count
  LDR  LR,[LR]
  CMP  R12,LR
  BHS  SVC_Dead              ; Overflow
  LDR  LR,=SVC_Table
  LDR  R12,[LR,R12,LSL #2]   ; Load SVC Function Address
  BLX  R12                   ; Call SVC Function
```

The next section of the SVC exception handler prepares to jump to the add() function. First, the link register and R4 are pushed onto the stack. The size of the SVC table is loaded into the link register. The SVC ordinal is compared to the table size to check whether it is less than the SVC table size and hence a valid number. If it is valid, the function address is loaded into R12 from the SVC table and the function is called. If the ordinal number has not been added to the table, the code will jump to a trap called SVC_DEAD. Although R4 is not used in this example, it is preserved on the stack as it is possible for the called function to use it.

```
POP  {R4,LR}
TST  LR,#4
MRSNE  R12,PSP
MOVEQ  R12,SP
STM  R12,{R0-R3}  ; Function return values
BX  LR  ; RETI
```

Once the SVC function has been executed, it will return back to the SVC exception handler to clean up before returning to the background code in handler mode.

Pend_SVC Exception

To extend the support for OSes, there is a second software interrupt called Pend_SVC. The purpose of the Pend_SVC exception is to minimize the latency experienced by interrupt service routines running alongside the OS. A typical OS running on the Cortex-M processor will use the SVC exception to allow user threads to make calls to the OS API, and the systick timer will be used to generate a periodic exception to run the OS scheduler.

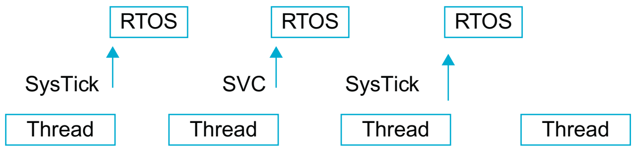


Figure 5.7
A typical RTOS will use the systick timer to provide a periodic interrupt to run the kernel scheduler. Calls to RTOS functions will use the SVC instruction to raise an exception.

While this works fine, problems can occur when the systick timer is running at a high priority. If we enter a peripheral interrupt and start to run the ISR and then the systick timer interrupt is triggered, it will preempt the peripheral interrupt and delay servicing of the peripheral interrupt while the OS scheduler examines the state of the user threads and performs a context switch to start a new thread running. This process can delay execution of the peripheral interrupt by several hundred cycles.

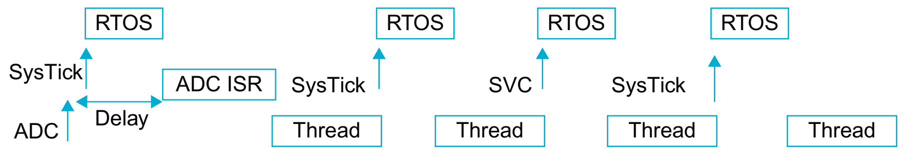


Figure 5.8
If the RTOS systick interrupt is running at a high priority, it can delay peripheral interrupts making interrupt handling nondeterministic.

The Pend_SVC is used to minimize this problem. When the systick timer occurs, it will check the NVIC registers to see if any peripheral interrupt is pending. If this is the case, it will cause a Pend_SVC exception by setting the Pend bit in the NVIC Pend_SVC register. The OS scheduler then exits. Now we have two active interrupts, the original peripheral interrupt and the Pend_SVC exception. The Pend_SVC is set to a low priority so that the peripheral interrupt ISR will resume execution. As soon as the peripheral interrupt has finished, the Pend_SVC exception will be served, which will continue with the OS scheduler to finish the thread context switch. This technique maintains the performance of the OS while keeping the intrusion on the peripheral interrupts to a minimum.

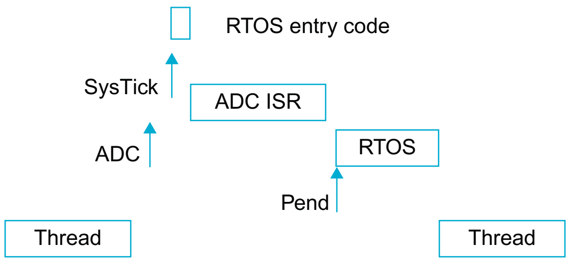


Figure 5.9
When execution enters the RTOS scheduler, it can check if any other interrupts are pending. If this is the case, the low-priority Pend_SVC can be set to pend and the RTOS scheduler suspended. The peripheral interrupt will now be served. When it is finished the Pend_SVC exception will be served, allowing the RTOS scheduler to resume processing.

Example: Pend_SVC

In this example, we will examine how to use the Pend_SVC system service call interrupt.

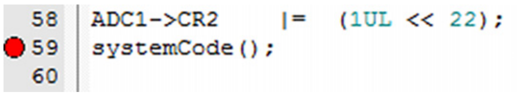
Open the project in c:\examples\Pendsvc.

Build the project and start the debugger.

The code initializes the ADC and enables its end of conversion interrupt. It also changes the Pend_SVC and ADC interrupt priority from their default options. The SVC has priority zero (highest) while the ADC has priority 1, and the Pend_SVC interrupt has priority 2 (lowest). The systemCode routine uses an SVC instruction to raise an exception and move to handler mode.

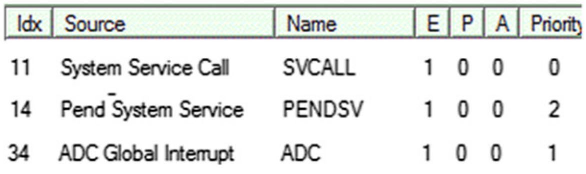
```
NVIC_SetPriority(PendSV_IRQn,2);  //set interrupt priorities
NVIC_SetPriority(ADC1_2_IRQn,1);
NVIC_EnableIRQ(ADC1_2_IRQn);      //enable the ADC interrupt
ADC1->CR1  |= (1UL << 5);         //switch on the ADC and start a conversion
ADC1->CR2  |= (1UL << 0);
ADC1->CR2  |= (1UL << 22);
systemCode();                     //call some system code with an SVC interrupt
```

Set a breakpoint on the systemCode( ) function and run the code.



```
ADC1->CR2     |=  (1UL << 22);
systemCode();
```

Open the Peripherals\Core Peripherals\Nested Vector Interrupt Controller window and check the priority levels of the SVC, Pend_SVC, and ADC interrupts.



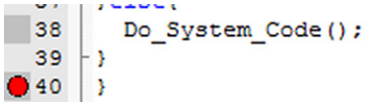
Idx	Source	Name	E	P	A	Priority
11	System Service Call	SVCALL	1	0	0	0
14	Pend System Service	PENDSV	1	0	0	2
34	ADC Global Interrupt	ADC	1	0	0	1

Now step into the systemCode routine (F11) until you reach the C function.

```
void __svc(0) systemCode (void);
void __SVC_0 (void) {
unsigned int i, pending;
  for(i = 0;i < 100;i++);
pending = NVIC_GetPendingIRQ(ADC1_2_IRQn);
if(pending = = 1){
    SCB->ICSR |= 1<<28;           //set the pend pend
}else{
  Do_System_Code();
}
}
```

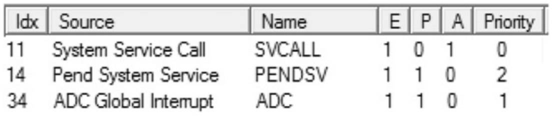
Inside the system code routine, there is a short loop that represents the critical section of code that must be run. While this loop is running, the ADC will finish conversion, and as it has a lower priority than the SVC interrupt it will enter a pending state. When we exit the loop, we test the state of any critical interrupts by reading their pending bits. If a critical interrupt is pending, then the remainder of the system code routine can be delayed. To do this, we set the PENDSVSET bit in the interrupt control and state register and quit the SVC handler.

Set a breakpoint on the exit brace (}) of the systemCode routine and run the code.



```
    Do_System_Code();
}
}
```

Now use the NVIC debug window to examine the state of the interrupts.

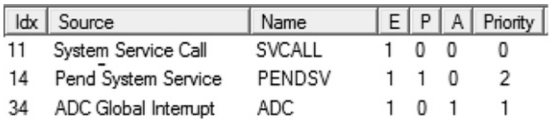


Idx	Source	Name	E	P	A	Priority
11	System Service Call	SVCALL	1	0	1	0
14	Pend System Service	PENDSV	1	1	0	2
34	ADC Global Interrupt	ADC	1	1	0	1

Now the SVC call is active with the ADC and Pend_SVC system service call in a pending state.

Single Step (F11) to the end of the system service call. Continue to single step so that you exit the system service routine and enter the next pending interrupt.

Both of the pending interrupts will be tail chained on to the end of the system service call. The ADC has the highest priority so it will be served next.



Idx	Source	Name	E	P	A	Priority
11	System Service Call	SVCALL	1	0	0	0
14	Pend System Service	PENDSV	1	1	0	2
34	ADC Global Interrupt	ADC	1	0	1	1

Step out of the ADC handler and you will immediately enter the Pend_SVC system service interrupt, which allows you to resume execution of the system code that was requested to be executed in the system service call interrupt.

Interprocessor Events

The Cortex-M processors are designed so that it is possible to build multiprocessor devices. An example would be to have a Cortex-M4 and a Cortex-M0 within the same microcontroller. The Cortex-M0 will typically manage the user peripherals, while the Cortex-M4 runs the intensive portions of the application code. Alternatively, there are devices that have a Cortex-A8 that can run Linux and manage a complex user interface;

on the same chip there are two Cortex-M4s that manage the real-time code. These more complex system-on-chip designs require methods of signaling activity between the different processors. The Cortex-M processors can be chained together by an external event signal. The event signal is set by using a set event instruction. This instruction can be added to your C code using the __SEV() intrinsic provided by the CMSIS core specification. When a SEV instruction is issued it will wake up the target processor if it has entered a low-power mode using the WFE instruction. If the target processor is running, the event latch will be set so that when the target processor executes the WFE instruction it will reset low-power mode.

Exclusive Access

One of the key features of an RTOS is multitasking support. As we will see in the next chapter, this allows you to develop your code as independent threads that conceptually are running in parallel on the Cortex-M processor. As your code develops, the program threads will often need to access common resources, be it SRAM or peripherals. An RTOS provides mechanisms called semaphores and mutexes that are used to control access to peripherals and common memory objects.

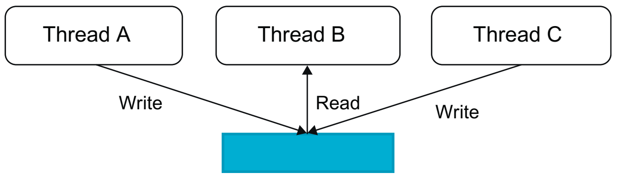


Figure 5.10
In a multiprocessor or multithread environment, it is necessary to control access to shared resources, or errors such as read before write can occur.

While it is possible to design "memory lock" routines on any processor, the Cortex-M3 and Cortex-M4 provide a set of instructions that can be used to optimize exclusive access routines.

Table 5.1: Exclusive Access Instructions

__LDREXB	Load exclusive (8 bits)
__LDREXH	Load exclusive (16 bits)
__LDREXW	Load exclusive (32 bits)
__STREXB	Store exclusive (8 bits)
__STREXH	Store exclusive (16 bits)
__STREXW	Store exclusive (32 bits)
__CLREX	Remove exclusive lock

In earlier ARM processors like the ARM7 and ARM9, the problem of exclusive access was answered by a swap instruction that could be used to exchange the contents of two registers. This instruction took four cycles but it was an atomic instruction, meaning that once started it could not be interrupted and was guaranteed exclusive access to the CPU to carry out its operation. As Cortex-M processors have multiple busses, it is possible for read and write accesses to be carried out on different busses and even by different bus masters, which may themselves be additional Cortex-M processors. On the Cortex-M processor, the new technique of exclusive access instructions has been introduced to support multitasking and multiprocessor environments.

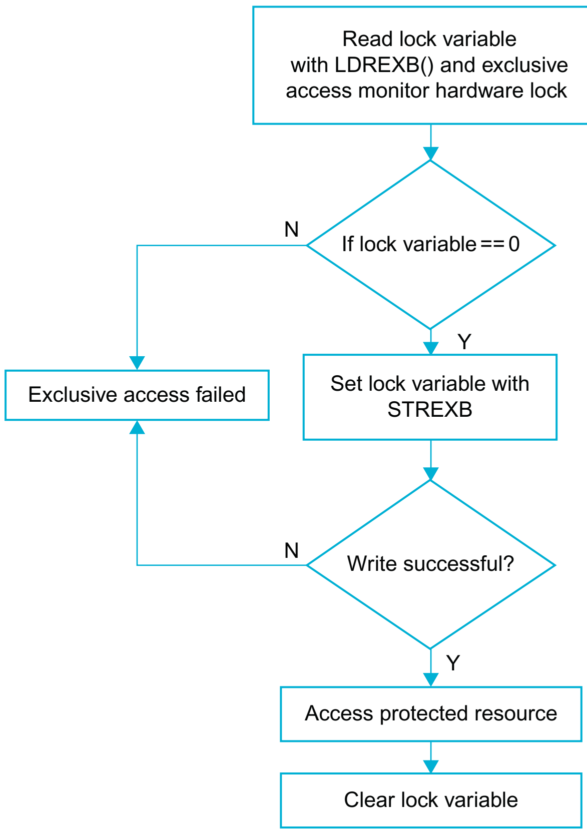

Read lock variable with LDREXB() and exclusive access monitor hardware lock
If lock variable == 0
N
Y
Exclusive access failed
Set lock variable with STREXB
Write successful?
N
Y
Access protected resource
Clear lock variable

Figure 5.11
The load and store exclusive instructions can be used to control access to a memory resource. They are designed to work with single and multiprocessor devices.

The exclusive access system works by defining a lock variable to protect the shared resource. Before the shared resource can be accessed, the locked variable is checked using the exclusive read instruction; if it is zero, then the shared resource is not currently being accessed. Before we access the shared resource, the lock variable must be set using the exclusive store instruction. Once the lock variable has been set, we now have control of the shared resource and can write to it. If our process is preempted by an interrupt or another thread that also performs an exclusive access read, then a hardware lock in the exclusive access monitor is set, preventing the original exclusive store instruction from writing to the lock variable. This gives exclusive control to the preempting process. When we are finished with the shared resource, the lock variable must be written to zero; this clears the variable and also removes the lock. If your code starts the exclusive access process but needs to abandon it, there is a clear exclusive (CLREX) instruction that can be used to remove the lock. The exclusive access instructions control access between different processes running on a single Cortex-M processor, but the same technique can be extended to a multiprocessor environment provided that the silicon designer includes the additional monitor hardware bus signals.

Exercise: Exclusive Access

In this exercise, we will create an exclusive access lock that is shared between a background thread process and an SVC handler routine to demonstrate the lock and unlock process.

Open the exercise in c:\exercises\exclusive access.

Build the code and start the debugger.

```
int main (void) {
if(__LDREXB( &lock_bit) = = 0){
  if (!__STREXB(1,&lock_bit = = 0)){
    semaphore++;
    lock_bit = 0;
      }
  }
```

The first block of code demonstrates a simple use of the exclusive access instructions. We first test the lock variable with the exclusive load instruction. If the resource is not locked by another process, we set the lock bit with the exclusive store instruction. If this is successful, we can then access the shared memory resource called a semaphore. Once this variable has been updated, the lock variable is written to zero, and the clear exclusive instruction releases the hardware lock.

Step through the code to observe its behavior.

```
if(__LDREXB( &lock_bit)= =0){
      thread_lock();
  if (!__STREXB(1,&lock_bit)){
      semaphore++;
    }
```

The second block of code does exactly the same thing except between the exclusive load and exclusive store the function is called thread_lock(). This is an SVC routine that will enter handler mode and jump to the SVC0 routine.

```
void __svc(0) thread_lock (void);
void __SVC_0  (void) {
__LDREXB( &lock_bit);
}
```

The SVC routine simply does another exclusive read of the lock variable that will set the hardware lock in the exclusive access monitor. When we return to the original routine and try to execute the exclusive store instruction, it will fail because any exception that happens between LDREX and STREX will cause the STREX to fail. The local exclusive access monitor is cleared automatically at exception entry/exit.

Step through the second block of code and observe the lock process.

Memory Protection Unit

The Cortex-M3/4 and Cortex-M0+ processors have an optional memory protection hardware unit that may be included in the processor core by the silicon manufacturer when the microcontroller is designed. The MPU allows you to extend the privileged/ unprivileged code model. If it is fitted, the MPU allows you to define regions within the memory map of the Cortex-M processor and grant privileged or unprivileged access to these regions. If the processor is running in unprivileged mode and tries to access an address within a privileged region, a memory protection exception will be raised and the processor will vector to the memory protection ISR. This allows you to detect and correct runtime memory errors.

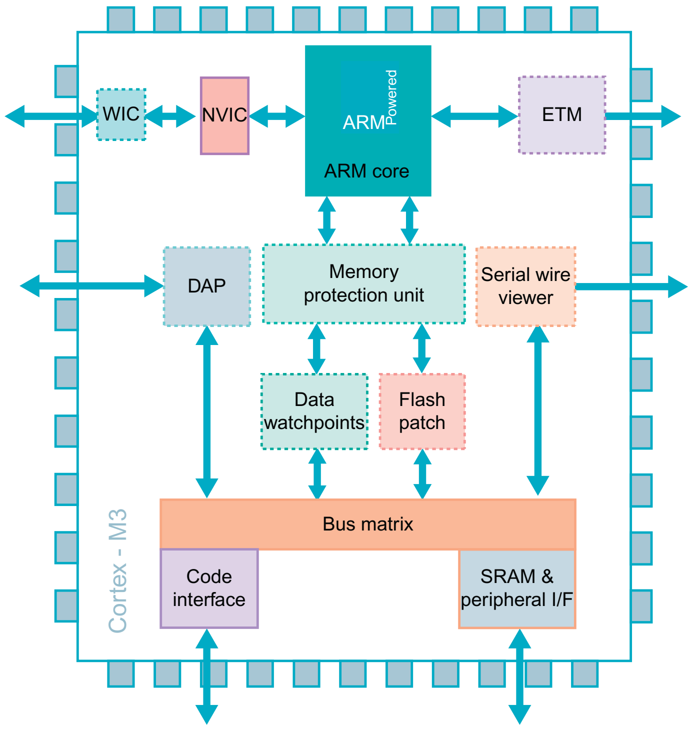


Figure 5.12
The MPU is available on the Cortex-M0+, Cortex-M3, and Cortex-M4. It allows you to place a protection template over the processor memory map.

In practice, the MPU allows you to define eight memory regions within the Cortex-M processor address space and grant privileged or unprivileged access to each region. These regions can then be further subdivided into eight equally sized subregions that can in turn be granted privileged or unprivileged access. There is also a default background region that covers the entire 4 GB address space. When this region is enabled, it makes access to all memory locations privileged. To further complicate things, memory regions may be overlapped with the highest numbered region taking precedent. Also any area of memory that is not covered by an MPU region may not have any kind of memory access.

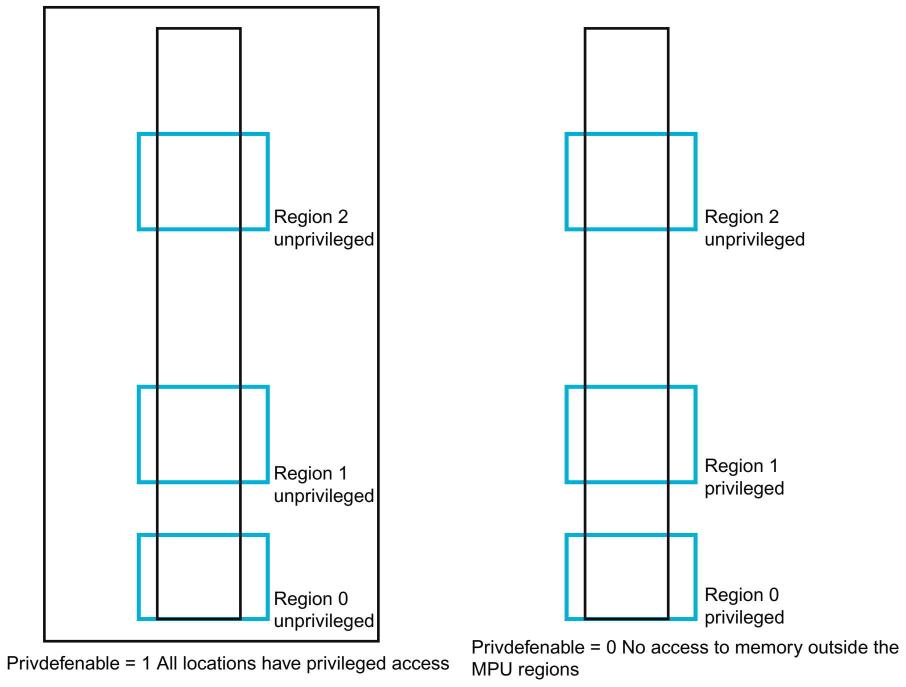


Figure 5.13
The MPU allows you to define eight regions, each with eight subregions over the processor memory map. Each region can grant different access privileges to its address range. It is also possible to set a default privileged access over the whole memory map and then create "holes" with different access privileges.

So it is possible to build up complex protection templates over the memory address space. This allows you to design a protection regime that helps build a robust operating environment but also gives you enough rope to hang yourself.

Configuring the MPU

The MPU is configured through a group of memory-mapped registers located in the Cortex-M processor system block. These registers may only be accessed when the Cortex processor is operating in privileged mode.

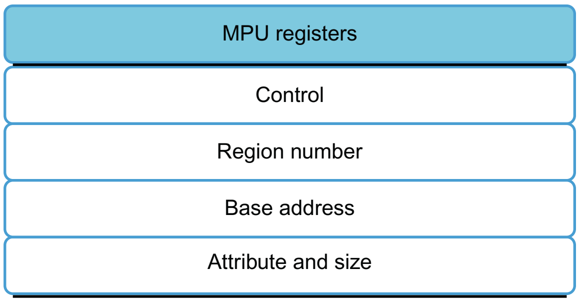


Figure 5.14
Each MPU region is configured through the region, base address, and attribute registers. Once each region is configured, the control register makes the MPU regions active.

The control register contains three active bits that effect the overall operation of the MPU. The first bit is the PRIVDEFENABLE bit that enables privileged access over the whole 4 GB memory map. The next bit is the HFNMIENA; when set, this bit enables the operation of the MPU during a hard fault, NMI, or FAULTMASK exception. The final bit is the MPU enable bit; when set, this enables the operation of the MPU. Typically, when configuring the MPU, the last operation performed is to set this bit. After reset, all of these bits are cleared to zero.

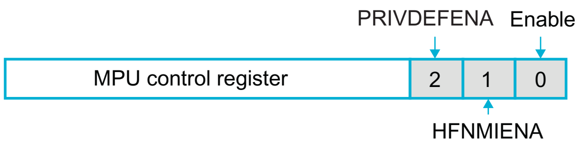


Figure 5.15
The control register allows you to enable the global privileged region with the Privilege default Enable (PRIVDEFENA). The enable bit is used to make the configured MPU regions active, and Hard Fault and Non Maskable Interrupt Enable (HFNMIEA) bit is used to enable the regions when the hard fault, NMI, or FAULTMASK exceptions are active.

The remaining registers are used to configure the eight MPU regions. To configure a given region (0–7), first select the region by writing its region number into the region number register. Once a region has been selected, it can then be configured by the base address and the attribute and size register. The base address register contains an address field which consists of the upper 27 address bits, the lower 5 address bits are not supported and are replaced by a valis bit and a repeat of the four bit MPU region number.

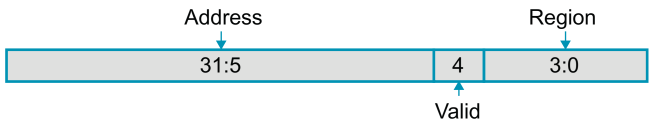


Figure 5.16
The address region of the base address register allows you to set the start address of an MPU region. The address values that can be written will depend on the size setting in the attribute and size register. If valid is set to 1, then the region number set in the region field is used; otherwise the region number in the region register is used.

As you might expect, the base address of the MPU region must be programmed into the address field. However, the available base addresses that may be used depend on the size of the defined for the region. The minimum size for a region is from 32 bytes upto 4 GB. The base address of an MPU region must be a multiple of the region size. Programming the address field sets the selected region's base address. You do not need to set the valid bit. If you write a new region number into the base address register region field, set the valid bit and write a new address; you can start to configure a new region without the need to update the region number register. Programming the attribute and size register finishes configuration of an MPU region.

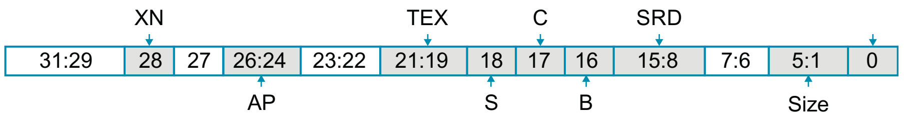


Figure 5.17
The attribute and size register allows you to define the MPU region size from 32 bytes to 4 GB. It also configures the memory attributes and access control options.

The size field defines the address size of the memory protection region in bytes. The region size is calculated using the formula

$$\text{MPU region memory size} = 2\ \text{POW}(\text{SIZE} + 1)$$

This gives us a minimum size starting at just 32 bytes. As noted above, the selected size also defines the range of possible base addresses. Next, it is possible to set the region attributes and access privileges. Like the Cortex-M processor, the MPU is designed to support multiprocessor systems. Consequently, it is possible to define regions as being shared between Cortex-M processors or as being exclusive to the given processor.

It is also possible to define the cache policy for the area of memory covered by the MPU region. Currently, the vast majority of microcontrollers only have a single Cortex processor, although asymmetrical multiprocessor devices have started to appear (Cortex-M4 and Cortex-M0), and currently no microcontroller has a cache. The attributes are defined by the TEX, C, B, and S bits, and suitable settings for most microcontrollers are shown below.

Table 5.2: Memory Region Attributes

Memory Region	TEX	C	B	S	Attributes
Flash	000	1	0	0	Normal memory, nonshareable, write through
Internal SRAM	000	1	0	1	Normal memory, shareable, write through
External SRAM	000	1	1	1	Normal memory, shareable, write back, write allocate
Peripherals	000	0	1	1	Device memory, shareable

When working with the MPU, we are more interested in defining the access permissions. These are defined for each region in the AP field.

Table 5.3: Memory Access Rights

AP	Privileged	Unprivileged	Description
000	No access	No access	All accesses generate a permission fault
001	RW	No access	Access from privileged code only
010	RW	RO	Unprivileged writes cause a permission fault
011	RW	RW	Full access
100	Unpredictable	Unpredictable	Reserved
101	RO	No access	
110	RO	RO	Reads by privileged code only
111	RO	RO	Read only for privileged and unprivileged code

Once the size, attributes, and access permissions are defined, the enable bit can be set to make the region active. When each of the required regions has been defined, the MPU can be activated by setting the global enable bit in the control register. When the MPU is active and the application code makes an access that violates the permissions of a region, an MPU exception will be raised. Once you enter an MPU exception, there are a couple of registers that provide information to help diagnose the problem. The first byte of the configurable fault status register located in the system control block is called the memory manager fault status register.

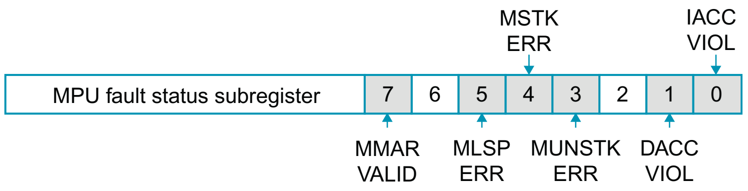


Figure 5.18
The MPU fault status register is a subsection of the configurable fault status register in the system control block. It contains error flags that are set when an MPU exception occurs. The purpose of each flag is shown.

Table 5.4: Fault Status Register Flag Descriptions

Flag	Description
IACCVIOL	Instruction access violation status flag
DACCVIOL	Data access violation status flag
MUNSTKERR	Memory manager fault on unstacking
MSTKERR	Memory manager fault on stacking
MLSPERR	Memory manager FPU lazy stacking error, Cortex-M4 only (see Chapter 7)
MMARVALID	Memory manager fault address valid

Depending on the status of the fault conditions, the address of the instruction that caused the memory fault may be written to a second register, that is, the memory manager fault address register. If this address is valid, it may be used to help diagnose the fault.

Exercise: MPU Configuration

In this exercise, we will configure the MPU to work with the blinky project.

Open the project in exercises\Blinky MPU.

This time, the microcontroller used is an NXP LPC1768, which has a Cortex-M3 processor fitted with the MPU. First, we have to configure the project so that there are distinct regions of code and data that will be used by the processor in thread and handler modes. When doing this, it is useful to sketch out the memory map of the application code and then define a matching MPU template.

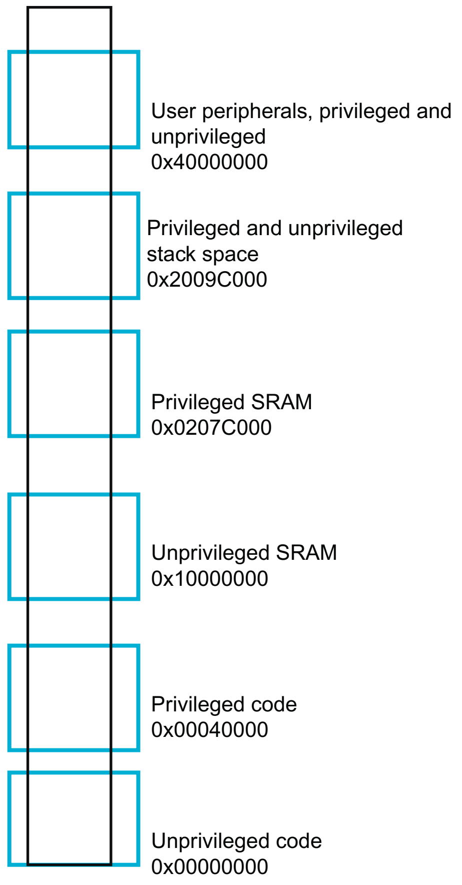
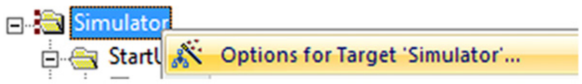
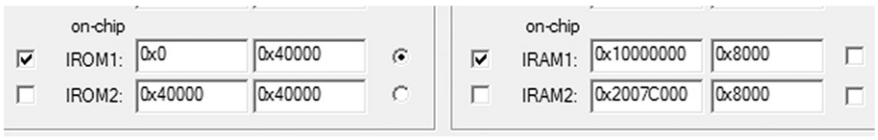


Figure 5.19
The memory map of the blinky project can be split into six regions.

```
Region 0: Unprivileged application code
Region 1: Privileged system code
Region 2: Unprivileged SRAM
Region 3: Privileged SRAM
Region 4: Privileged and unprivileged stack space
Region 5: Privileged and unprivileged user peripherals
```

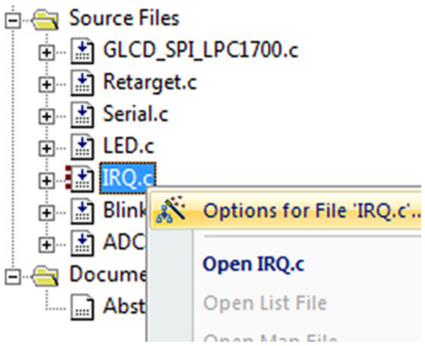
Now open the Options for Target\Target menu. Here, we can set up the memory regions to match the proposed MPU protection template.

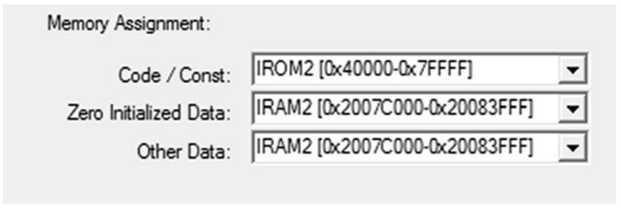




	on-chip				
✓	IROM1:	0x0	0x40000	●	
☐	IROM2:	0x40000	0x40000	○	
	on-chip				
✓	IRAM1:	0x10000000	0x8000	☐	
☐	IRAM2:	0x2007C000	0x8000	☐	

The target memory map defines two regions of code memory: 0-0x3FFF and 0x4000–0x7FFF. The lower region will be used as the default region to hold the application code and will be accessed by the processor in unprivileged mode. The upper region will be used to hold the interrupt and exception service routines and will be accessed by the processor in privileged mode. Similarly, there are two RAM regions—0x10000000 and 0x100008000—that will hold data used by the unprivileged code and the system stacks, while the upper region—0x0207C000—will be used to hold the data used by the privileged code. In this example, we are not going to set the background privileged region, so we must map MPU regions for the peripherals. All the peripherals except the GPIO are in one contiguous block from 0x40000000, while the GPIO registers sit at 0x000002C9. The peripherals will be accessed by the processor while it is in both privileged and unprivileged modes.





Memory Assignment:

Code / Const:	IROM2 [0x40000-0x7FFFF]
Zero Initialized Data:	IRAM2 [0x2007C000-0x20083FFF]
Other Data:	IRAM2 [0x2007C000-0x20083FFF]

To prepare the code, we need to force the interrupt handler code into the regions that will be granted privileged access. In this example, all the code that will run in handler mode has been placed in one module. In the local options for this module, we can select the code and data regions that will be given privileged access rights by the MPU. All of the other code and data will be placed in the default memory regions that will run in unprivileged mode.

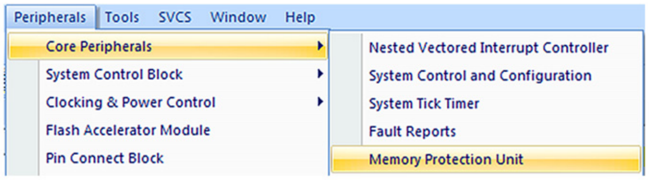
Once the project memory layout has been defined, we can add code to the project to set up the MPU protection template.

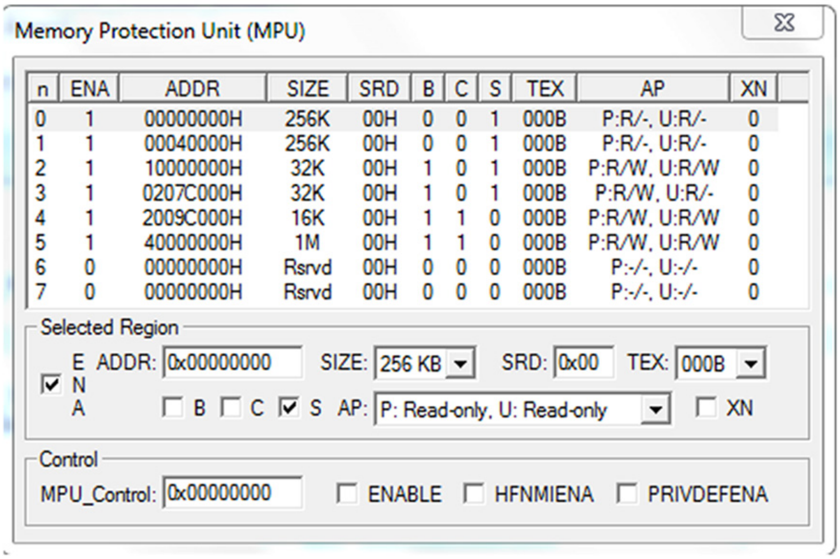
```
#define SIZE_FIELD                   1
#define ATTRIBUTE_FIELD              16
#define ACCESS_FIELD                 24
#define ENABLE                       1
#define ATTRIBUTE_FLASH              0x4
#define ATTRIBUTE_SRAM               0x5
#define ATTRIBUTE_PERIPHERAL         0x3
#define PRIV_RW_UPRIV_RW             3
#define PRIV_RO_UPRIV_NONE           5
#define PRIV_RO_UPRIV_RO             6
#define PRIV_RW_UPRIV_RO             2
#define USE_PSP_IN_THREAD_MODE       2
#define THREAD_MODE_IS_UNPRIVILIGED  1
#define PSP_STACK_SIZE               0x200
#define TOP_OF_THREAD_RAM            0x10007FF0
MPU->RNR   =  0x00000000;
MPU->RBAR  =  0x00000000;
MPU->RASR  =  (PRIV_RO_UPRIV_RO<<ACCESS_FIELD)
              |(ATTRIBUTE_FLASH<<ATTRIBUTE_FIELD)
              |(17<<SIZE_FIELD)|ENABLE;
```

The code shown above is used to set the MPU region for the unprivileged thread code at the start of memory. First, we need to set a region number followed by the base address of the region. Since this will be flash memory, we can use the standard attribute for this memory type. Next we can define its access type. In this case, we can grant read-only access for both privileged and unprivileged modes. Next, we can set the size of the region, which is 256 K, must equal 2POW(SIZE + 1), and equate to 17. The enable bit is set to activate this region when the MPU is fully enabled. Each of the other regions are programmed in a similar fashion. Finally, the memory management exception and the MPU are enabled.

```
NVIC_EnableIRQ(MemoryManagement_IRQn);
MPU->CTRL = ENABLE;
```

Start the debugger, set a breakpoint on line 82, and run the code. When the breakpoint is reached, we can view the MPU configuration via the Peripherals\Core Peripherals\Memory Protection Unit menu.




Memory Protection Unit (MPU)

n	ENA	ADDR	SIZE	SRD	B	C	S	TEX	AP	XN
0	1	00000000H	256K	00H	0	0	1	000B	P:R/-, U:R/-	0
1	1	00040000H	256K	00H	0	0	1	000B	P:R/-, U:R/-	0
2	1	10000000H	32K	00H	1	0	1	000B	P:R/W, U:R/W	0
3	1	0207C000H	32K	00H	1	0	1	000B	P:R/W, U:R/-	0
4	1	2009C000H	16K	00H	1	1	0	000B	P:R/W, U:R/W	0
5	1	40000000H	1M	00H	1	1	0	000B	P:R/W, U:R/W	0
6	0	00000000H	Rsrvd	00H	0	0	0	000B	P:-/-, U:-/-	0
7	0	00000000H	Rsrvd	00H	0	0	0	000B	P:-/-, U:-/-	0

Selected Region

[✓] ENA ADDR: 0x00000000 SIZE: 256 KB SRD: 0x00 TEX: 000B

[ ] B [ ] C [✓] S AP: P: Read-only, U: Read-only [ ] XN

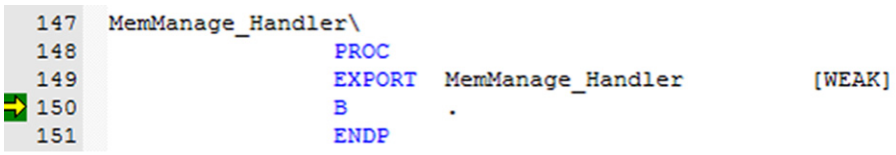
Control

MPU_Control: 0x00000000 [ ] ENABLE [ ] HFNMIENA [ ] PRIVDEFENA

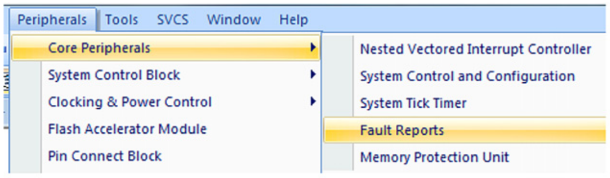
Here, we can easily see the regions defined and the access rights that have been granted.

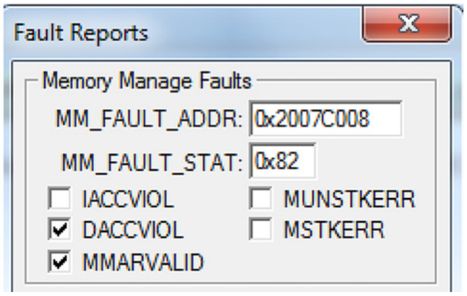
Now run the code for a few seconds and then halt the processor.

An MPU exception has been raised and execution has jumped to the memManager handler.


```
MemManage_Handler\
                    PROC
                    EXPORT  MemManage_Handler          [WEAK]
                    B       .
                    ENDP
```

The question now is what caused the MPU exception? We can find this out by looking at the memory manager fault and status register. This can be directly viewed in the debugger by opening the Peripherals\Core Peripherals\Fault Reports window.




Fault Reports

Memory Manage Faults

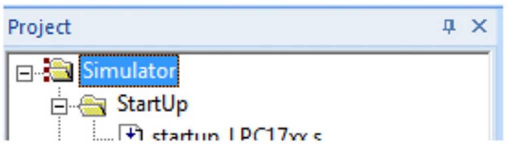
MM_FAULT_ADDR: 0x2007C008

MM_FAULT_STAT: 0x82

- [ ] IACCVIOL
- [ ] MUNSTKERR
- [x] DACCVIOL
- [ ] MSTKERR
- [x] MMARVALID

Here, we can see that the fault was caused by a data access violation to address 0x2007C008. If we now open the map file produced by the linker, we can search for this address and find what variable is placed at this location.

Highlight the project name in the project window and double click. This will open the map file. Now use the Edit\Find dialog to search the map file for the address 0x2007C008.



```
    text                    0x10000010   Data        10  blinky.o(.bss)
    __initial_sp            0x10000220   Data         0  startup_lpc17xx.o(STACK)
    clock_1s                0x2007c008   Data         1  irq.o(.data)
```

This shows that the variable clock_1s is at 0x2007C008 and that it is declared in irq.c.

Clock_1s is a global variable that is also accessed from the main loop running in unprivileged mode. However, this variable is located in the privileged RAM region, so accessing it while the processor is running in unprivileged mode will cause an MPU fault.

Find the declaration of clock_1s in blinky.c and remove the extern keyword.

Now find the declaration for clock_1s in irq.c and add the keyword extern.

Build the code and view the updated map file.

```
__stdin                   0x10000008   Data        4  retarget.o(.data)
clock_1s                  0x1000000c   Data        1  blinky.o(.data)
AD_done                   0x1000000e   Data        1  adc.o(.data)
```

Now clock_1s is declared in blinky.c and is located in the unprivileged RAM region so that it can be accessed by both privileged and unprivileged code.

Now restart the debugger and the code will run without raising any MPU exceptions.

MPU Subregions

As we have seen, the MPU has a maximum of eight regions that can be individually configured with location size and access type. Any region that is configured with a size of 256 bytes or more will contain eight equally spaced subregions. When the region is configured, each of the subregions is enabled and has the default region attributes and access settings. It is possible to disable a subregion by setting a matching subregion bit in the Subregion Disable (SRD) field of the MPU attribute and size register. When a subregion is disabled, a "hole" is created in the region; this "hole" inherits the attributes and access permission of any overlapped region. If there is no overlapped region, then the global privileged background region will be used. If the background region is not enabled, then no access rights will be granted and an MPU exception will be raised if an access is made to an address in the subregion "hole."

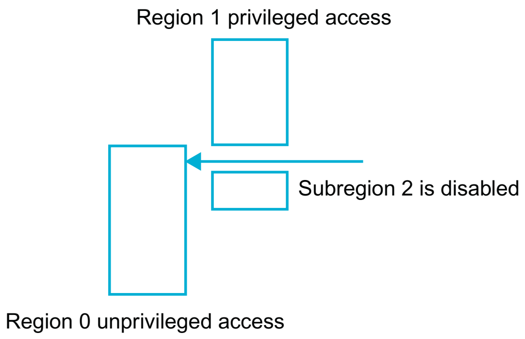


Figure 5.20
Each region has eight subregions. If a subregion is disabled, it inherits the access rights from an overlapped region or the global background region.

If we have two overlapped regions, the region with the highest region number will take precedence. In the case above, an unprivileged region is overlapped by a privileged region. The overlapped section will have privileged access. If a subregion is disabled in region 1, then the access rights in region 0 will be inherited and granted unprivileged access to the subregion range of addresses.

MPU Limitations

When designing your application software to use the MPU, it is necessary to realize that the MPU only monitors the activity of the Cortex-M processor. Many, if not most, Cortex-M-based microcontrollers have other peripherals, such as direct memory access (DMA) units, that are capable of autonomously accessing memory and peripheral registers. These units are additional "bus masters" that arbitrate with the Cortex-M processor to gain access to the microcontroller resources. If such a unit makes an access to a prohibited region of memory, it will not trigger an MPU exception. This is important to remember as the Cortex-M processor has a bus structure that is designed to support multiple independent "bus master" devices.

AHB Lite Bus Interface

The Cortex-M processor family has a final important architectural improvement over the earlier generation of ARM7- and ARM9-based microcontrollers. In these first-generation ARM-based microcontrollers, the CPU was interfaced to the microcontroller through two types of busses. These were the AHB and the advanced peripheral bus (APB).

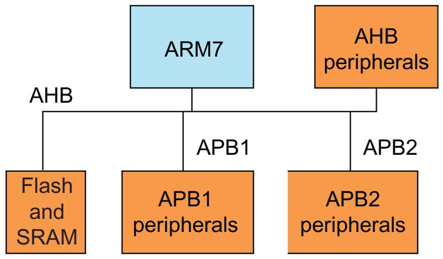


Figure 5.21
The first generation of ARM-based microcontrollers had an internal bus system based on the AHB and the APB. As multiple bus masters (CPU, DMA) were introduced, a bus arbitration phase had to be completed before a transfer could be made across the bus.

The high-speed bus connected the CPU to the flash and SRAM memory while the microcontroller peripherals were connected to one or more APB busses. The AHB bus also supported additional bus masters such as DMA units to sit alongside the ARM7 processor. While this system worked, the bus structure started to become a bottleneck, particularly as more complex peripherals such as Ethernet MAC and USB were added. These peripherals contained their own DMA units, which also needed to act as a bus master. This meant that there could be several devices (ARM7 CPU, general-purpose DMA, and Ethernet MAC) arbitrating for the AHB bus at any given point in time. As more and more complex peripherals are added, the overall throughput and deterministic performance became difficult to predict.

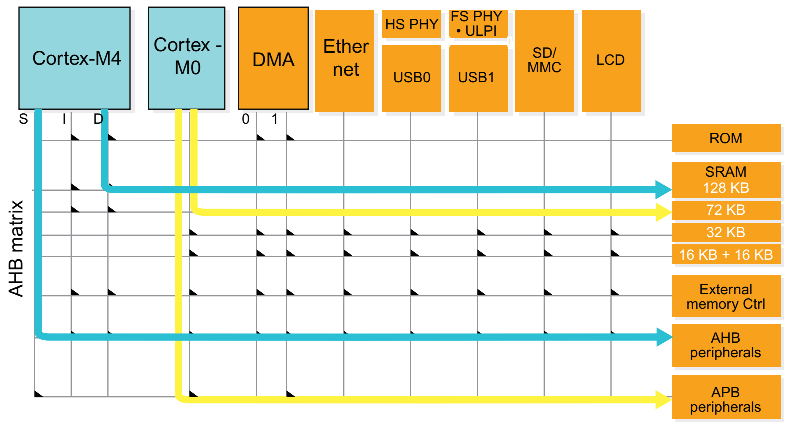


Figure 5.22
The Cortex-M processor family replaces the single AHB bus with a bus matrix that provides parallel paths for each bus master to block each of the slave devices.

The Cortex-M processor family overcomes this problem by using an AHB bus matrix. The AHB bus matrix consists of a number of parallel AHB busses that are connected to different regions of the chip. These regions are laid out by the manufacturer when the chip is designed. Each region is a slave device; this can be the flash memory, a block of SRAM, or a group of user peripherals on an APB bus. Each of these regions is then connected back to each of the bus masters through additional AHB busses to form the bus matrix. This allows manufacturers to design complex devices with multiple Cortex-M processors, DMA units, and advanced peripherals, each with parallel paths to the different device resources. The bus matrix is hardwired into the microcontroller and does not need any configuration by your application code. However, when you are designing the application code, you should pay attention to where different memory objects are located. For example, the memory used by the Ethernet controller should be placed in one block of SRAM while the USB memory is located in a separate SRAM block. This allows the Ethernet, USB and DMA units to work in parallel while the Cortex processor is accessing the flash and user peripherals. So by structuring the memory map of your application code, you can exploit this degree of parallelism and gain an extra boost in performance.