CHAPTER 3

Cortex-M Architecture

Introduction

In this chapter, we will take a closer look at the Cortex-M processor architecture. The bulk of this chapter will concentrate on the Cortex-M3 processor. Once we have a firm understanding of the Cortex-M3, we will look at the key differences in the Cortex-M0, M0+, and M4. There are a number of exercises throughout the chapter. These exercises will give you a deeper understanding of each topic and can be used as a reference when developing your own code.

Cortex-M Instruction Set

As we described in Chapter 1, the Cortex-M processors are Reduced Instruction Set Computer (RISC)-based processors and as such have a small instruction set. The Cortex-M0 has just 56 instructions, the Cortex-M3 has 74, and the Cortex-M4 has 137 with an option of additional 32 instructions for the FPU. The ARM CPUs, ARM7 and ARM9, which were originally used in microcontrollers, have two instruction sets: the ARM (32 bit) instruction set and the Thumb (16 bit) instruction set. The ARM instruction set was designed to get maximum performance from the CPU while the Thumb instruction set featured good code density to allow programs to fit into the limited memory resources of a small microcontroller. The developer had to decide which function was compiled with the ARM instruction set and which was compiled with the Thumb instruction set. Then the two groups of functions could be "interworked" together to build the final program. The Cortex-M instruction set is based on the earlier 16-bit Thumb instruction set found in the ARM processors but extends that set to create a combined instruction set with a blend of 16- and 32-bit instructions.

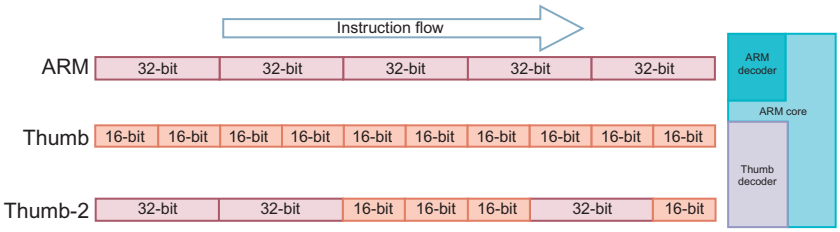


Figure 3.1
The ARM7 and ARM9 CPU had separate 32- and 16-bit instruction sets. The Cortex-M processor has a single instruction set that is a blend of 16- and 32-bit instructions.

The Cortex-M instruction set, called Thumb-2, is designed to make this much simpler and more efficient. The good news is that your whole Cortex-M project can be coded in a high level language such as C\C++ without the need for any hand-coded assembler. It is useful to be able to "read" Thumb-2 assembly code via a debugger disassembly window to check what the compiler is up to, but you will never need to write an assembly routine. There are some useful Thumb-2 instructions that are not reachable using the C language but most compiler toolchains provide intrinsic instructions which can be used to access these instructions from within your C code.

Programmer's Model and CPU Registers

The Cortex-M processors inherit the ARM RISC's load and store method of operation. This means that to do any kind of data processing, instruction such as ADD and SUBTRACT the data must first be loaded into the CPU registers; the data processing instruction is then executed and the result is stored back in the main memory. This means that code executing on a Cortex-M processor revolves around the central CPU registers.

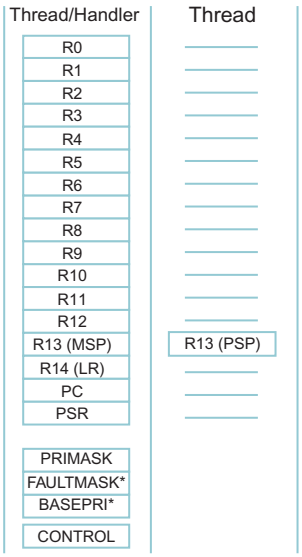


Figure 3.2
The Cortex-M CPU registers consist of 16 data registers, a program status register, and four special function registers. R13–R15 are used as the stack pointer, link register, and program counter. R13 is a banked register which allows the Cortex-M CPU to operate with dual stacks.

On all Cortex-M processors, the CPU register file consists of 16 data registers followed by the program status register and a group of configuration registers. All of the data registers (R0–R15) are 32 bits wide and can be accessed by all of the Thumb-2 load and store instructions. The remaining CPU registers may only be accessed by two dedicated

instructions, move general register to special register (MRS) and move special register to general register (MSR), where the "special registers" are PRIMASK, FAULTMASK, BASEPRI, and CONTROL. The registers R0−R12 are general user registers and are used by the compiler as it sees fit. The registers R13−R15 have special functions. R13 is used by the compiler as the stack pointer; this is actually a banked register with two R13 registers. When the Cortex-M processor comes out of reset, this second R13 register is not enabled and the processor runs in a "simple" mode with one stack pointer. It is possible to enable the second R13 register by writing to the Cortex control register. The processor will then be configured to run with two stacks. We will look at this in more detail in Chapter 5 "Advanced Architecture Features" but for now we will use the Cortex-M processor in its default mode. After the stack pointer we have R14, the link register. When a procedure is called the return address is automatically stored in R14. Since the Thumb-2 instruction set does not contain a RETURN instruction when the processor reaches the end of a procedure it uses the branch instruction on R14 to return. Finally, R15 is the program counter. You can operate on this register just like all the others but you will not need to do this during normal program execution. The CPU registers PRIMASK, FAULTMASK, and BASEPRI are used to temporarily disable interrupt handling and we will look at these later in this chapter.

Program Status Register

The PSR as its name implies contains all the CPU status flags.

Figure 3.3

The PSR contains several groups of CPU flags. These include the condition codes (NZCVQ), interrupt continuable instruction (ICI) status bits, If Then (IT) flag, and current exception number.

The PSR has a number of alias fields that are masked versions of the full register. The three alias registers are the application program status register (APSR), interrupt program status register (IPSR), and the execution program status register (EPSR). Each of these alias registers contains a subset of the full register flags and can be used as a shortcut if you need to access part of the PSR. The PSR is generally referred to as the xPSR to indicate the full register rather than any of the alias subsets.

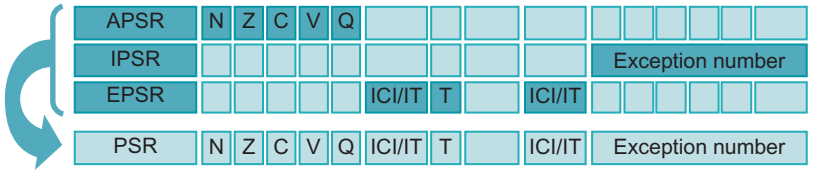


Figure 3.4

The PSR has three alias registers that provide access to specific subregions of the PSR. Hence, the generic name for the PSR is xPSR.

In a normal application program, your code will not make explicit access to the xPSR or any of its alias registers. Any use of the xPSR will be made by compiler generated code. As a programmer you need to have an awareness of the xPSR and the flags contained in it.

The most significant four bits of the xPSR are the condition code bits—**N**egative, **Z**ero, **C**arry, and o**V**erflow. These will be set and cleared depending on the results of a data processing instruction. The result of Thumb-2 data processing instructions can set or clear these flags. However, updating these flags is optional.

```
SUB R8, R6, #240   Perform a subtraction and do not update the condition code flags
SUBS R8, R6, #240  Perform a subtraction and update the condition code flags
```

This allows the compiler to perform an instruction that updates the condition code flags, then perform some additional instructions that do not modify the flags and then perform a conditional branch on the state of the xPSR condition codes. Following the four condition code flags is a further instruction flag, the Q bit.

Q Bit and Saturated Math Instructions

The Q bit is the saturation flag. The Cortex-M3 and Cortex-M4 processors have a special set of instructions called the saturated math instructions. If a normal variable reaches its maximum value and you increment it further, it will roll round to zero. Similarly, if a variable reaches its minimum value and is then decremented, it will roll round to the maximum value.

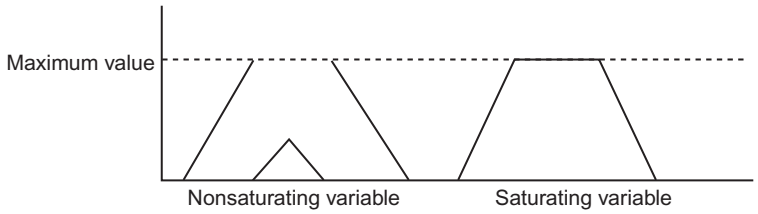


Figure 3.5
A normal variable will rollover to zero when it hits its maximum value. This is very dangerous in a control algorithm. The Cortex-M CPU supports saturated math instructions that stick at their maximum and minimum values.

While this is a problem for most applications, it is especially serious for applications such as motor control and safety critical applications. The Cortex-M3\M4 saturated math instructions prevent this kind of "roll round." When you use the saturated math instructions, if the variable reaches its maximum or minimum value it will stick (saturate) at that value. Once the variable is saturated the Q bit will be set. The Q bit is a "sticky" bit and must be cleared by the application code. The standard math instructions are not

used by the C compiler by default. If you want to make use of the saturated math instructions, you have to access them by using compiler intrinsic or CMSIS-core functions.

```
uint32_t __SSAT(uint32_t value, uint32_t sat)
uint32_t __USAT(uint32_t value, uint32_t sat)
```

Interrupts and Multicycle Instructions

The next field in the PSR is the "ICI" and "IT" instruction flags. Most of the Cortex-M processor instructions are executed in a single cycle. However, some instructions such as load store multiple, multiply, and divide take multiple cycles. If an interrupt occurs while these instructions are executing, they have to be suspended while the interrupt is served. Once the interrupt has been served, we have to resume the multicycle instructions. The ICI field is managed by the Cortex-M processor so you do not need to do anything special in your application code. It does mean that when an exception is raised reaching the start of your interrupt routine will always take the same amount of cycles regardless of the instruction currently being executed by the CPU.

Conditional Execution—IF THEN Blocks

As we have seen in Chapter 1, the Cortex-M processors have a three-stage pipeline. This allows the fetch, decode, and execute units to operate in parallel greatly improving the performance of the processor. However, there is a disadvantage that every time the processor makes a jump, the pipeline has to be flushed and refilled. This introduces a big hit on performance as the pipeline has to be refilled before the execution of instructions can resume. The Cortex-M3 and Cortex-M4 reduce the branch penalty by having an instruction fetch unit that can carry out speculative branch target fetches from the possible branch address so the execution of the branch targets can start earlier. However, for small conditional branches, the Cortex-M processor has another trick up its sleeve. For a small conditional branch, for example

```
If(Y==0x12C){
  I++;
  }else{
  I-;}
```

which compiles to less than four instructions, the Cortex-M processor can compile the code as an IF THEN condition block. The instructions inside the IF THEN block are extended with a condition code. This condition code is compared to the state of the condition code flags in the PSR. If the condition matches the state of the flags, then the instruction will be executed and if it does not then the instruction will still enter the pipeline but will be

executed as a no operation (NOP). This technique eliminates the branch and hence avoids the need to flush and refill the pipeline. So even though we are inserting NOP instructions, we still get better performance levels.

Table 3.1: Instruction Condition Codes

Condition Code	xPSR Flags Tested	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or HS	C = 1	Higher or same (unsigned)
CC or LO	C = 0	Lower (unsigned)
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 and Z = 0	Higher (unsigned)
LS	C = 0 or Z = 1	Lower or same (unsigned)
GE	N = V	Greater than or equal (signed)
LT	N! = V	Less than (signed)
GT	Z = 0 and N = V	Greater than (signed)
LE	Z = 1 and N! = V	Less than or equal (signed)
AL	None	Always execute

To trigger an IF THEN block, we use the data processing instructions to update the PSR condition codes. By default, most instructions do not update the condition codes unless they have an S suffix added to the assembler opcode. This gives the compiler a great deal of flexibility in applying the IF THEN condition.

```
ADDS R1, R2, R3  //perform and add and set the xPSR flags
ADD R2, R4, R5  //Do some other instructions but do not modify the xPSR
ADD R5, R6, R7
IT VS  //IF THEN block conditional on the first ADD instruction
SUBVS R3, R2, R4
```

Consistently our ‘C’ IF THEN ELSE statement can be compiled to four instructions.

```
CMP    r6,#0x12C
ITE    EQ
STREQ  r4,[r0,#0x08]
STRNE  r5,[r0,#0x04]
```

The CMP compare instruction is used to perform the test and will set or clear the zero Z flag in the PSR. The IF THEN block is created by the IF THEN (IT) instruction. The IT instruction is always followed by one conditionally executable instruction and

optionally up to four conditionally executable instructions. The format of the IT instruction is as follows:

```
IT x y z cond
```

The x, y, and z parameters enable the second, third, and fourth instructions to be a part of the conditional block. There can be a further THEN or ELSE instruction. The cond parameter is the condition applied to the first instruction. So,

```
ITTTE NE  A four-instruction IF THEN block with three THEN instructions which executes
          when Z = 1 followed by an ELSE instruction which executes when Z = 1
ITE GE    A two-instruction IF THEN block with one THEN instruction which executes when
          N = V and one ELSE instruction which executes when N! = V
```

The use of conditional executable IF THEN blocks is left up to the compiler. Generally, at low levels of optimization IF THEN blocks are not used; this gives a good debug view. However, at high levels of optimization, the compiler will make use of IF THEN blocks. So, normally there will not be any strange side effects introduced by the conditional execution technique but there are a few rules to bear in mind.

First, conditional code blocks cannot be nested; generally the compiler will take care of this rule. Secondly, you cannot use a GOTO statement to jump into a conditional code block. If you do make this mistake the compiler will warn you and not generate such illegal code.

Thirdly, the only time you will really notice execution of an IF THEN condition block is during debugging. If you single step the debugger through the conditional statement, the conditional code will appear to execute even if the condition is false. If you are not aware of the Cortex-M condition code blocks, this can be a great cause of confusion!

The next bit in the PSR is the T or Thumb bit. This is a legacy bit from the earlier ARM CPUs and is set to one when the processor is released from reset. If your code tries to clear this bit, it will be set to zero but a fault exception will also be raised. In previous CPUs, the T bit was used to indicate that the Thumb 16-bit instruction set was running. It is included in the Cortex-M PSR to maintain compatibility with earlier ARM CPUs and allow legacy 16-bit Thumb code to be executed on the Cortex-M processors. The final field in the PSR is the exception number field. The Cortex NVIC can support up to 256 exception sources. When an exception is being processed, the exception number is stored here. As we will see later, this field is not used by the application code when handling an interrupt, though it can be a useful reference when debugging.

Exercise: Saturated Math and Conditional Execution

In this exercise, we will use a simple program to examine the CPU registers and make use of the saturated math instructions. We will also rebuild the project to use conditional execution.

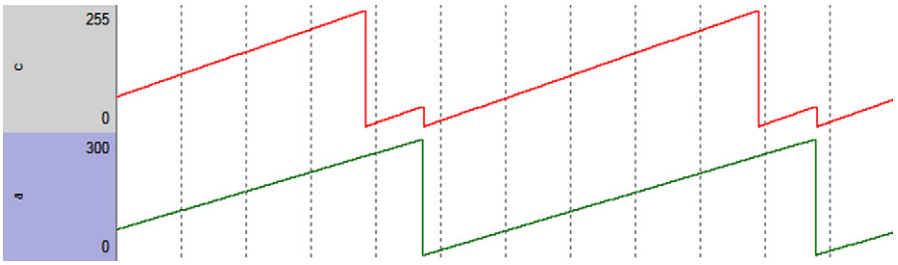
Open the project in C:\exercises\saturation.

```
int a,range=300;  char c;
int main(void){
while(1){
for(a=0;a<range;a++){
    c=a;
}}
```

This program increments an integer variable from 0 to 300 and copies it to a char variable.

Build the program and start the debugger.

Add the two variables to the logic analyzer and run the program.



In the logic analyzer window, we can see that while the integer variable performs as expected the char variable saturates when it reaches 255 and "rolls over" to zero and begins incrementing again.

Stop the debugger and modify the code as shown below to use saturated math.

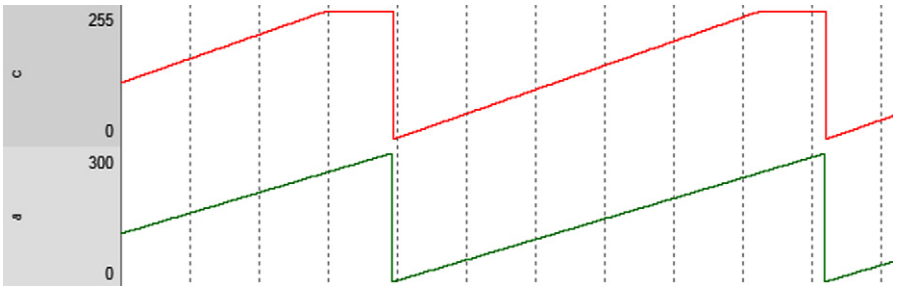
```
#define Q_FLAG 0x08000000
int a,range=300;
char c;
unsigned int xPSR;

int main(void){
while(1){
for(a=0;a<range;a++){
  c=__SSAT(a,9);
  }
}}
```

This code replaces the equate statement with a saturated intrinsic function that saturates on the ninth bit of the integer value. This allows values 0−255 to be written to the byte value; any other values will saturate at the maximum allowable value of 255.

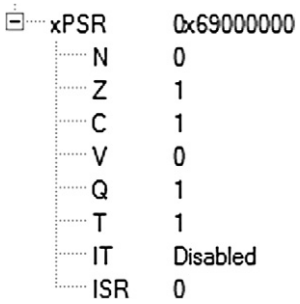
Build the project and start the debugger.

Run the code and view the contents of the variables in the logic analyzer.



Now the char variable saturates rather than "rolling over." This is still wrong but not as potentially catastrophically wrong as the rollover case.

In the registers window, click on the xPSR regiset to view the flags.



In addition to the normal NVCZ condition code flags, the saturation Q bit is set.

Stop the debugger and modify the code as shown below.

```
#define Q_FLAG 0x08000000
int a,range = 300;  char c;  unsigned int APSR;
register unsigned int apsr __asm("apsr");
int main (void){
while (1){
for(a = 0;a < range;a++){
    c = __SSAT(a, 9);
    }
APSR =  __get_APSR ();
if(APSR&Q_FLAG){
    range-;
```

```
}
apsr = apsr&~Q_FLAG;
}}
```

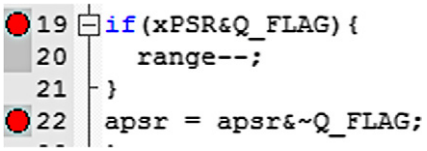
Once we have written to the char variable, it is possible to read the xPSR and check if the Q bit is set. If the variable has saturated, we can take some corrective action and then clear the Q bit for the next iteration.

Build the project and start the debugger.

Run the code and observe the variables in the watch window.

Now, when the data is over range, the char variable will saturate and gradually the code will adjust the range variable until the output data fits into the char variable.

Set breakpoints on lines 19 and 22 to enclose the Q bit test.

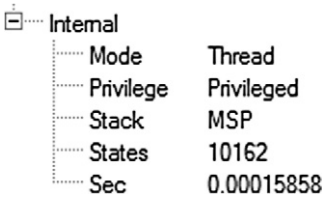


Reset the program and then run the code until the first breakpoint is reached.

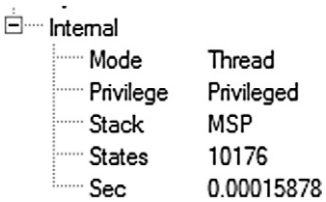
Open the disassembly window and examine the code generated for the Q bit test.

```
  19: if(xPSR&Q_FLAG){
MOV  r0,r1
LDR  r0,[r0,#0x00]
TST  r0,#0x8000000
BEQ  0x080003E0
  20:    range-;
  21: }else{
LDR  r0,[pc,#44]; @0x08000400
LDR  r0,[r0,#0x00]
SUB  r0,r0,#0x01
LDR  r1,[pc,#36]; @0x08000400
STR  r0,[r1,#0x00]
B  0x080003E8
  22: locked = 1;
  23: }
MOV  r0,#0x01
LDR  r1,[pc,#32]; @0x08000408
STR  r0,[r1,#0x00]
```

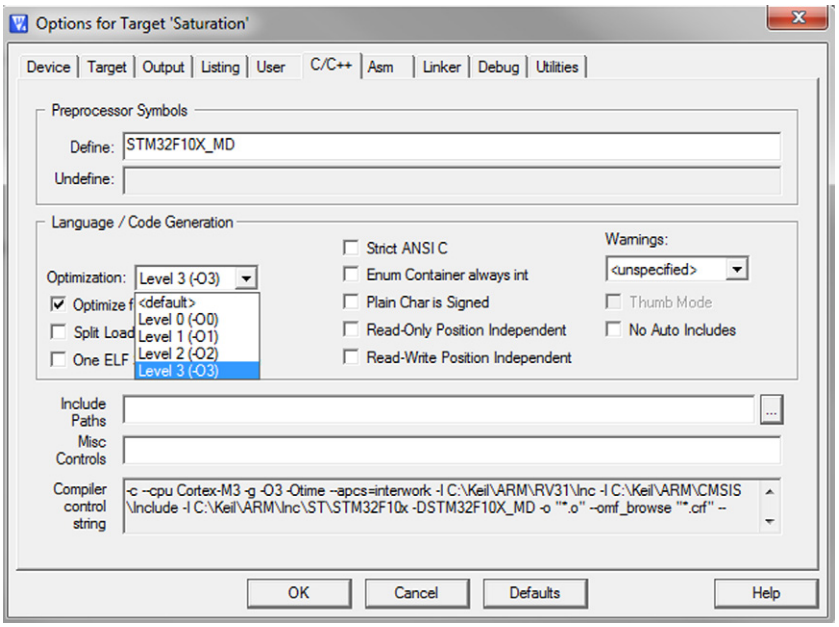
Also make a note of the value in the state counter. This is the number of cycles used since reset to reach this point.



Now run the code until it hits the next breakpoint and again make a note of the state counter.



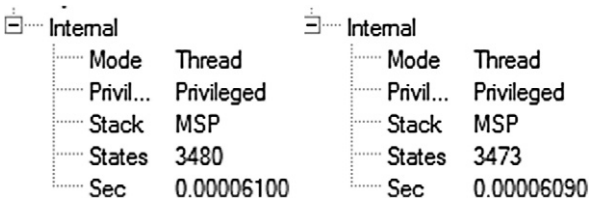
Stop the debugger and open the Options for Target\C tab.



Change the optimization level from Level 0 to Level 3.

Close the Options for Target window and rebuild the project.

Now repeat the cycle count measurement by running to the two breakpoints.



Now the Q bit test takes 7 cycles as opposed to the original 14.

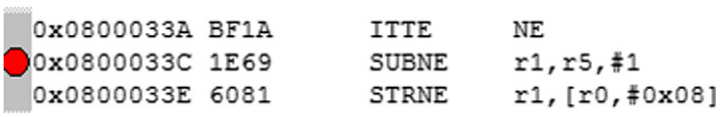
Examine the disassembly code for the Q bit test.

```
  19: if(xPSR&Q_FLAG){
0x08000336 F0116F00 TST  r1,#0x8000000
  20:    range-;
  21: }else{
ITTE  NE
STRNE  r1,[r0,#0x08]
  22: locked=1;
  23: }
STREQ  r4,[r0,#0x04]
```

At higher levels of optimization, the compiler has switched from test and branch instructions to conditional execution instructions. Here, the assembler is performing a bitwise AND test on R1, which is holding the current value of the xPSR. This will set or clear the Z flag in the xPSR. The ITT instruction sets up a two-instruction conditional block. The instructions in this block perform a subtract and store if the Z flag is zero; otherwise they pass through the pipeline as NOP instructions.

Remove the breakpoints. Run the code for a few seconds then halt it.

Set a breakpoint on one of the conditional instructions.



Start the code running again.

The code will hit the breakpoint even though the breakpoint is within an IF THEN statement that should no longer be executed. This is simply because the conditional instructions are always executed.

Cortex-M Memory Map and Busses

While each manufacturer's Cortex-M device has different peripherals and memory sizes, ARM has defined a basic memory template that all devices must adhere to. This provides a standard layout so all the vendor-provided memory and peripherals are located in the same blocks of memory.

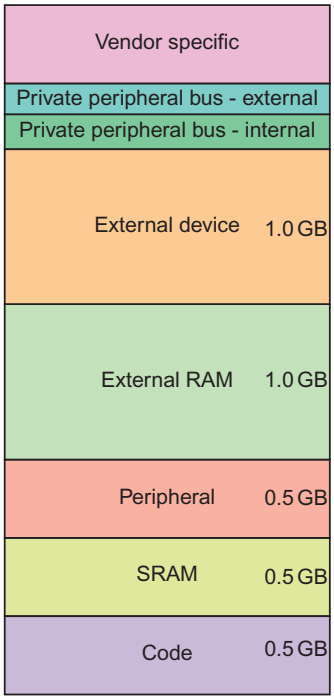


Figure 3.6
The Cortex-M memory map has a standard template which splits the 4 GB address range into specific memory regions. This memory template is common to all Cortex-M devices.

The Cortex-M memory template defines eight regions that cover the 4 GB address space of the Cortex-M processor. The first three regions are each 0.5 GB in size and are dedicated to the executable code space, internal SRAM, and internal peripherals. The next two regions are dedicated to external memory and memory mapped devices; both regions are 1 GB in size. The final three regions make up the Cortex-M processor memory space and contain the configuration registers for the Cortex-M processor and any vendor-specific registers.

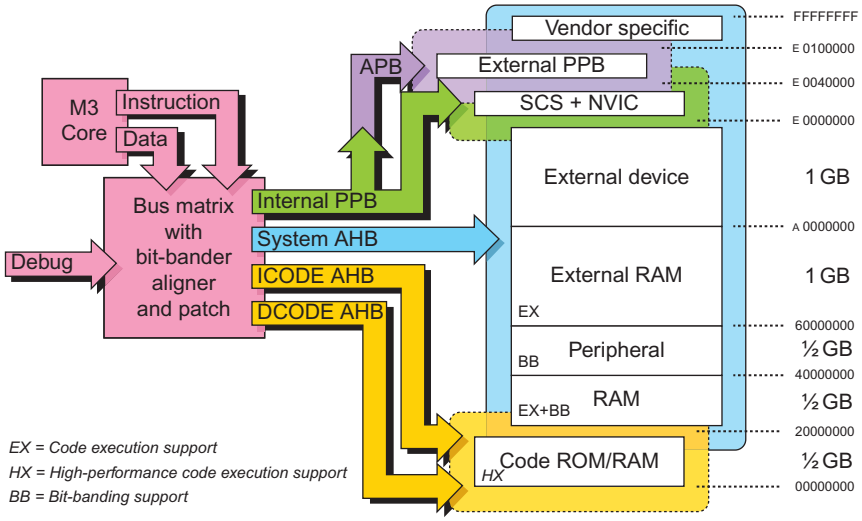


Figure 3.7
While the Cortex-M processor has a number of internal busses, these are essentially invisible to the software developer. The memory appears as a flat 4 GB address space.

While the Cortex-M memory map is a linear 4 GB address space with no paged regions or complex addressing modes, the microcontroller memory and peripherals are connected to the Cortex-M processor by a number of different busses. The first 0.5 GB of the address space is reserved for executable code and code constants. This region has two dedicated busses. The ICODE bus is used to fetch code instructions and the DCODE bus is used to fetch code constants. The remaining user memory spaces (internal SRAM and peripherals plus the external RAM and peripherals) are accessed by a separate system bus. The Cortex-M processor memory space has an additional private peripheral bus. While this may look complicated, as far as your application code is concerned you have one seamless memory space; the Cortex-M processor has separate internal busses to optimize its access to different memory regions.

As mentioned earlier, most of the instructions in the Thumb-2 instruction set are executed in a single cycle and the Cortex-M3 can run up to 200 MHz, and in fact some system on chip (SoC) designs manage to get the processor running even faster. However, current flash memory used to store the program has an access time of around 50 MHz. So there is a basic problem of pulling instructions out of the flash memory fast enough to feed the Cortex-M processor. When you are selecting a Cortex-M microcontroller it is important to study the data sheet to see how the silicon vendor has solved this problem. Typically, the flash memory will be arranged as 64- or 128-bit wide memory, so one read from the flash memory can load multiple instructions. These instructions are then held in a "memory accelerator" unit which then feeds the instructions to the Cortex-M processor as required.

The memory accelerator is a form of simple cache unit that is designed by the silicon vendor. Normally, this unit is disabled after reset, so you will need to enable it or the Cortex-M processor will be running directly from the flash memory. The overall performance of the Cortex-M processor will depend on how successfully this unit has been implemented by the designer of the microcontroller.

Write Buffer

The Cortex-M3 and Cortex-M4 contain a single entry data write buffer. This allows the CPU to make an entry into the write buffer and continue on to the next instruction while the write buffer completes the write to the real SRAM. If the write buffer is full, the CPU is forced to wait until it has finished its current write. While this is normally a transparent process to the application code, there are some cases where it is necessary to wait until the write has finished before continuing program execution. For example, if we are enabling an external bus on the microcontroller, it is necessary to wait until the write buffer has finished writing to the peripheral register and the bus is enabled before trying to access memory located on the external bus. The Cortex-M processor provides some memory barrier instructions to deal with these situations.

Memory Barrier Instructions

The memory barrier instructions halt execution of the application code until a memory write of an instruction has finished executing. They are used to ensure that a critical section of code has been completed before continuing execution of the application code.

Table 3.2: Memory Barrier Instructions

Instruction	Description
Data memory synchronization barrier (DMD)	Ensures all memory accesses are finished before a fresh memory access is made
Data synchronization barrier (DSB)	Ensures all memory accesses are finished before the next instruction is executed
Instruction synchronization barrier (ISB)	Ensures that all previous instructions are completed before the next instruction is executed. This also flushes the CPU pipeline

System Control Block

In addition to the CPU registers, the Cortex-M processors have a group of memory mapped configuration and status registers located near the top of the memory map starting at 0xE000 E008.

We will look at the key features supported by these registers in the rest of this book, but a summary is given below.

Table 3.3: The Cortex Processor Has Memory Mapped Configuration and Status Registers Located in the System Control Block (SCB)

Register	Size in Words	Description
Auxiliary control	1	Allows you to customize how some processor features are executed
CPU ID	1	Hardwired ID and revision numbers from ARM and the silicon manufacturer
Interrupt control and state	1	Provides pend bits for the systick and NMI Non Maskable interrupt along with extended interrupt pending\active information
Vector table offset	1	Programmable address offset to move the vector table to a new location in flash or SRAM memory
Application interrupt and reset control	1	Allows you to configure the PRIGROUP and generate CPU and microcontroller resets
System control	1	Controls configuration of the processor sleep modes
Configuration and control	1	Configures CPU operating mode and some fault exceptions
System handler priority	3	These registers hold the 8-bit priority fields for the configurable processor exceptions
System handler control and state	1	Shows the cause of a bus, memory management, or usage fault
Configurable fault status	1	Shows the cause of a bus, memory management, or usage fault
Hard fault status	1	Shows what event caused a hard fault
Memory manager fault address	1	Holds the address of the memory location that generated the memory fault
Bus fault address	1	Holds the address of the memory location that generated the memory fault

The Cortex-M instruction set has addressing instructions that allow you to load and store 8-, 16-, and 32-bit quantities. Unlike the ARM7 and ARM9, the 16- and 32-bit quantities do not need to be aligned on word or halfword boundaries. This gives the compiler and linker the maximum flexibility to fully pack the SRAM memory. However, there is a penalty to be paid for this flexibility because unaligned transfers take longer to carry out. The Cortex-M instruction set contains load and store multiple instructions that can transfer multiple registers to and from memory in one instruction. This takes multiple processor cycles but uses only one 2-byte or 4-byte instruction. This allows for very efficient stack manipulation and block memory copy. The load and store multiple instructions only work for word aligned data. So, if you use unaligned data, the compiler is forced to use multiple individual load and store instructions to achieve the same thing. While you are making full use of the valuable internal SRAM, you are potentially increasing the application instruction size. Unaligned data is for user data only; you must ensure that the stacks are word aligned. The main stack pointer's (MSP) initial value is

determined by the linker, but the second stack pointer, the process stack pointer (PSP), is enabled and initialized by the user. We will look at using the PSP in Chapter 5.

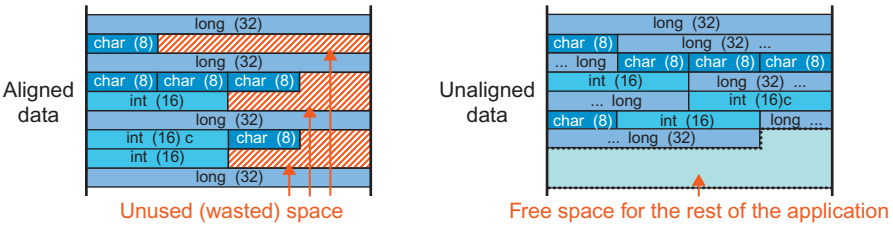


Figure 3.8

Unlike the earlier ARM7 and ARM9 CPUs, the Cortex processor can make unaligned memory accesses. This allows the compiler and linker to make the best use of the SRAM device.

Bit Manipulation

In a small embedded system, it is often necessary to set and clear individual bits within the SRAM and peripheral registers. By using the standard addressing instructions, we can set and clear individual bits by using the C language bitwise AND and OR commands. While this works fine, the Cortex-M processors provide a more efficient bit manipulation method.

The Cortex-M processor provides a method called "bit banding" which allows individual SRAM and peripheral register bits to be set and cleared in a very efficient manner.

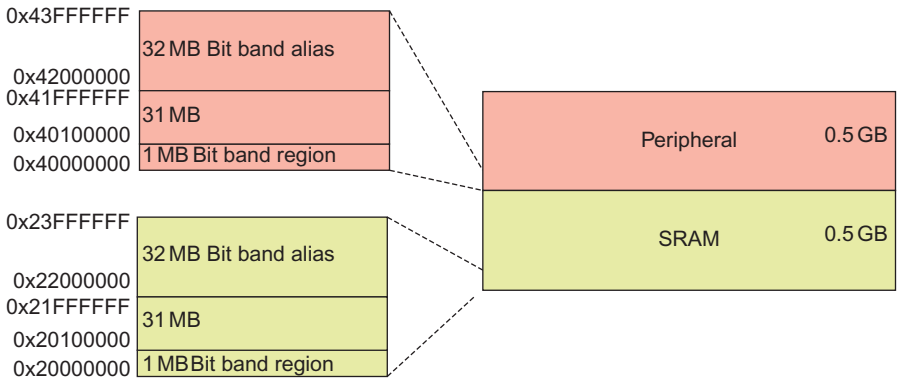


Figure 3.9

Bit banding is a technique that allows the first 1 MB of the SRAM region and the first 1 MB of the peripheral region to be bit addressed.

The first 1 MB of the SRAM region and the first 1 MB of the peripheral region are defined as bit band regions. This means that every memory location in these regions is bit addressable. So, in practice for today's microcontrollers, all of their internal SRAM and peripheral registers are bit addressable. Bit banding works by creating an alias word address

for each bit of real memory or peripheral register bit. So, the 1 MB of real SRAM is aliased to 32 MBytes of virtual word addresses.

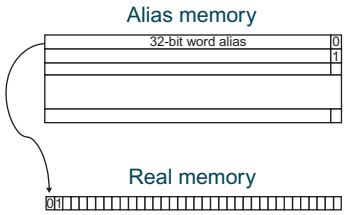


Figure 3.10
Each bit in the real memory is mapped to a word address in the alias memory.

This means that each bit of real RAM or peripheral register is mapped to a word address in the bit band alias region and by writing 1s and 0s to the alias word address we can set and clear the real memory bit location. Similarly, if we write a word to the real memory location, we can read the bit band alias address to check the current state of a bit in that word. To use bit banding, you simply need to calculate the word address in the bit band region that maps to the bit location in the real memory that you want to modify. Then create a pointer to the word address in the bit band region. Once this is done you can control the real bit memory location by reading and writing to the alias region via the pointer. The calculation for the word address in the bit band alias region is as follows:

$$\text{Bit band word address} = \text{bit band base} + (\text{byte offset} \times 32) + (\text{bit number} \times 4)$$

So, for example, if we want to read and write to bit 8 of the GPIO B port register on a typical Cortex microcontroller, we can calculate the bit band alias address as follows:

$$\text{GPIO B data register address} = 0x40010C0C$$

$$\begin{aligned}\text{Register byte offset from peripheral base address} &= 0x40010C0C - 0x40000000 \\ &= 0x00010C0C\end{aligned}$$

$$\begin{aligned}\text{Bit band word address} &= 0x42000000 + (0x000010C0C * 0x20) + (0x8 * 0x4) \\ &= 0x00010C0C\end{aligned}$$

Now we can define a pointer to this address.

```
#define GPIO_PORTB_BIT8 = (*((volatile unsigned long *)0x422181A0)
```

Now by reading and writing to this word address, we can directly control the individual port bit.

```
GPIO_PORTB_BIT8 = 1  //set the port pin
```

This will compile the following assembler instructions:

```
Opcode    Assembler
F04F0001    MOV  r0,#0x01
4927  LDR  r1,[pc,#156]; @0x080002A8
6008  STR  r0,[r1,#0x00]
```

This sequence uses one 32-bit instruction and two 16-bit instructions or a total of 8 bytes. If we compare this to setting the port pin by using a logical OR to write directly to the port register.

```
GPIOB->ODR|= 0x00000100;   //LED on
```

We then get the following code sequence:

```
Opcode    Assembler
481E  LDR  r0,[pc,#120]; @0x080002AC
6800  LDR  r0,[r0,#0x00]
F4407080  ORR  r0,r0,#0x100
491C  LDR  r1,[pc,#112]; @0x080002AC
6008  STR  r0,[r1,#0x00]
```

This uses four 16-bit instructions and one 32-bit instruction or 12 bytes.

The use of bit banding gives us a win-win situation, smaller code size, and faster operation. So, as a simple rule, if you are going to repetitively access a single bit location you should use bit banding to generate the most efficient code.

You may find some compiler tools or silicon vendor software libraries that provide macro functions to support bit banding. You should generally avoid using such macros as they do not yield the most efficient code.

Exercise: Bit Banding

In this exercise, we will look at defining a bit band variable to toggle a port pin and compare its use to bitwise AND and OR instructions.

Open the project in c:\exercises\bitband.

In this exercise, we want to toggle an individual port pin. We will use the port B bit 8 pin as we have already done the calculation for the alias word address.

So, now in the C code, we can define a pointer to the bit band address.

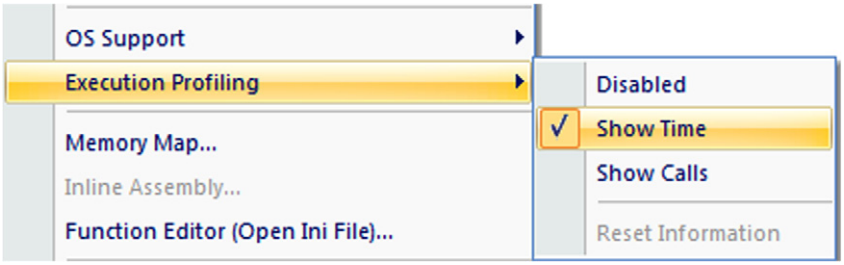
```
#define PortB_Bit8 (*((volatile unsigned long *)0x422181A0))
```

And in the application code, we can set and clear the port pin by writing to this pointer.

```
PortB_Bit8 = 1;
PortB_Bit8 = 0;
```

Build the project and start the debugger.

Enable the timing analysis with the Debug\Execution Profiling\Show Time menu.



This opens an additional column in the debugger that will display execution time for each line. Compare the execution time for the bit band instruction compared to the AND and OR instructions.

```
21          {
22    0.069 us  PortB_Bit8 = 1; //led on
23
```

```
30    0.153 us  GPIOB->ODR |= 0x00000100;     //LED on
```

Open the disassemble window and examine the code generated for each method of setting the port pin.

The bit banding instructions are the best way to set and clear individual bits. In some microcontrollers, the silicon vendor has added support for fast bit manipulation on certain registers, typically GPIO ports, which is as fast as bit banding. Whatever methods are available, you should use them in any part of your code that repetitively manipulates a bit.

Dedicated Bit Manipulation Instructions

In addition to bit band support, the Thumb-2 instruction set has some dedicated bit orientated instructions. Some of these instructions are not directly accessible from the C language and are supported by compiler "intrinsic" calls. However, this is a moving target as over time the compiler tools are becoming increasingly optimized to make full use of the Thumb-2 instruction set. For example, the sign extend a byte (SXTB), sign extend a halfword (SXTH), zero extend a byte (UXTB), and zero extend a halfword (UXTH) instructions are used in data type conversions.

Table 3.4: In Addition to Bit Banding the Cortex-M3 Processor Has Some Dedicated Bit Manipulation Instructions

BFC	Bit field clear
BFI	Bit field insert
SBFX	Signed bit field extract
SXTB	Sign extend a byte
SXTH	Sign extend a halfword
UBFX	Unsigned bit field extract
UXTB	Zero extend a byte
UXTH	Zero extend a halfword

Systick Timer

All of the Cortex-M processors also contain a standard timer. This is called the systick timer and is a 24-bit countdown timer with auto reload. Once started the systick timer will count down from its initial value. Once it reaches zero it will raise an interrupt and a new count value will be loaded from the reload register. The main purpose of this timer is to generate a periodic interrupt for an RTOS or other event-driven software. If you are not running an OS, you can also use it as a simple timer peripheral.

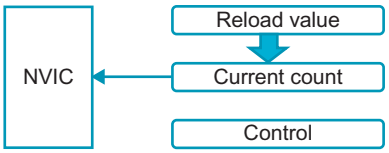


Figure 3.11
The systick timer is a 24-bit countdown timer with an auto reload. It is generally used to provide a periodic interrupt for an RTOS scheduler.

The default clock source for the systick timer is the Cortex-M CPU clock. It may be possible to switch to another clock source, but this will vary depending on the actual microcontroller you are using. While the systick timer is common to all the Cortex-M processors, its registers occupy the same memory locations within the Cortex-M3 and Cortex-M4. In the Cortex-M0 and Cortex-M0+, the systick registers are located in the SCB and have different symbolic names to avoid confusion. The systick timer interrupt line and all of the microcontroller peripheral lines are connected to the NVIC.

Nested Vector Interrupt Controller

Aside from the Cortex-M CPU, the next major unit within the Cortex-M processor is the NVIC. The usage of the NVIC is the same between all Cortex-M processors; once you have set up an interrupt on a Cortex-M3, the process is the same for the Cortex-M0, Cortex-M0+ ,

and Cortex-M4. The NVIC is designed for fast and efficient interrupt handling; on a Cortex-M3 you will reach the first line of C code in your interrupt routine after 12 cycles for a zero wait state memory system. This interrupt latency is fully deterministic so from any point in the background (noninterrupt) code you will enter the interrupt with the same latency. As we have seen, multicycle instructions can be halted with no overhead and then resumed once the interrupt has finished. On the Cortex-M3 and Cortex-M4, the NVIC supports up to 240 interrupt sources, and the Cortex-M0 can support up to 32 interrupt sources. The NVIC supports up to 256 interrupt priority levels on Cortex-M3 and Cortex-M4, and 4 priority levels on Cortex-M0.

Table 3.5: The NVIC Consists of Seven Register Groups That Allow You to Enable, Set Priority Levels for, and Monitor the User Interrupt Peripheral Channels

Register	Maximum Size in Words*	Description
Set enable	8	Provides an interrupt enable bit for each interrupt source
Clear enable	8	Provides an interrupt disable bit for each interrupt source
Set pending	8	Provides a set pending bit for each interrupt source
Clear pending	8	Provides a clear pending bit for each interrupt source
Active	8	Provides an interrupt active bit for each interrupt source
Priority	60	Provides an 8-bit priority field for each interrupt source
Software trigger	1	Write the interrupt channel number to generate a software interrupt

*The actual number of words used will depend on the number of interrupt channels implemented by the microcontroller manufacturer.

Operating Modes

While the Cortex CPU is executing background (noninterrupt code) code, the CPU is in an operating mode called thread mode. When an interrupt is raised, the NVIC will cause the processor to jump to the appropriate interrupt service routine (ISR). When this happens, the CPU changes to a new operating mode called handler mode. In simple applications without an OS, you can use the default configuration of the Cortex-M processor out of reset; there is no major functional difference in these operating modes and they can be ignored. The Cortex-M processors can be configured with a more complex operating model that introduces operating differences between thread and handler modes and we will look at this in Chapter 5.

Interrupt Handling—Entry

When a microcontroller peripheral raises an interrupt line, the NVIC will cause two things to happen in parallel. First, the exception vector is fetched over the ICODE bus. This is the

address of the entry point into the ISR. This address is pushed into R15, the program counter, forcing the CPU to jump to the start of the interrupt routine. In parallel with this, the CPU will automatically push key registers onto the stack. This stack frame consists of the following registers; xPSR, PC, LR, R12, R3, R2, R1, R0. This stack frame preserves the state of the processor and provides R0–R3 for use by the ISR. If the ISR needs to use more CPU registers, it must PUSH them onto the stack and POP them on exit.

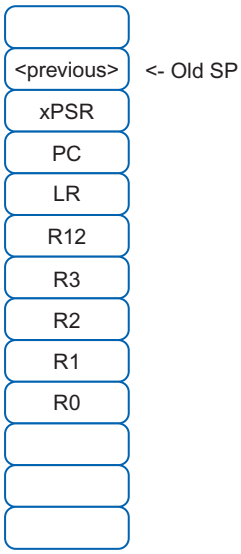


Figure 3.12
When an interrupt or exception occurs, the CPU will automatically push a stack frame. This consists of the xPSR, PC, LR, R12, and registers R0–R3. At the end of the interrupt or exception, the stack frame is automatically unstacked.

The interrupt entry process takes 12 cycles on the Cortex-M3\M4 and 16 cycles on the Cortex-M0. All of these actions are handled by microcode in the CPU and do not require any dedicated entry instructions like Long Jump (LJMP) or PUSH to be part of the application code.

The exception vectors are stored in an interrupt vector table. The interrupt vector table is located at the start of the address space; the first four bytes are used to hold the initial value of the stack pointer. The starting value of the stack pointer is calculated by the compiler and linker and automatically loaded into R13 when the Cortex processor is reset. The interrupt vector table then has address locations for every four bytes growing upward through the address space. The vector table holds the address of the ISR for each of the possible interrupt sources within the microcontroller. The vector table for each microcontroller comes predefined as part of the startup code. A label for each ISR is stored at each interrupt vector location. To create your ISR, you simply need to declare a void C function using the same name as the interrupt vector label.

```
  AREA  RESET, DATA, READONLY
  EXPORT __Vectors
__Vectors  DCD  __initial_sp    ; Top of Stack
  DCD  Reset_Handler    ; Reset Handler
; External Interrupts
  DCD  WWDG_IRQHandler    ; Window Watchdog
  DCD  PVD_IRQHandler    ; PVD through EXTI Line detect
  DCD  TAMPER_IRQHandler    ; Tamper
  DCD  RTC_IRQHandler    ; RTC
  DCD  FLASH_IRQHandler    ; Flash
  DCD  RCC_IRQHandler    ; RCC
```

So, to create the C routine to handle an interrupt from the real-time clock, we create a C function named as follows:

```
void RTC_IRQHandler(void) {
……
}
```

When the project is built, the linker will resolve the address of the C routine and locate it in the vector table in place of the label. If you are not using this particular interrupt in your project, the label still has to be declared to prevent an error occurring during the linking process. Following the interrupt vector table, there is a second table that declares all of the ISR addresses. These are declared as WEAK labels. This means that this declaration can be overwritten if the label is declared elsewhere in the project. In this case, they act as a "backstop" to prevent any linker errors if the interrupt routine is not formally declared in the project source code.

```
EXPORT  WWDG_IRQHandler [WEAK]
EXPORT PVD_IRQHandler  [WEAK]
EXPORT TAMPER_IRQHandler  [WEAK]  DCD  EXTI0_IRQHandler  ; EXTI Line 0
```

Interrupt Handling—Exit

Once the ISR has finished its task, it will force a return from the interrupt to the point in the background code from where it left off. However, the Thumb-2 instruction set does not have a return or return from interrupt instruction. The ISR will use the same return method as a noninterrupt routine, namely a branch on R14, the link register. During normal operation, the link register will contain the correct return address. However, when we entered the interrupt, the current contents of R14 were pushed onto the stack and in their place the CPU entered a special code. When the CPU tries to branch on this code instead of

doing a normal branch, it is forced to restore the stack frame and resume normal processing.

Table 3.6: At the Start of an Exception or Interrupt R14 (Link Register) Is Pushed Onto the Stack

Interrupt Return Value	Meaning
0xFFFFFFF9	Return to thread mode and use the MSP
0xFFFFFFFD	Return to thread mode and use the PSP
0xFFFFFFF1	Return to handler mode

The CPU then places a control word in R14. At the end of the interrupt, the code will branch on R14. The control word is not a valid return address and will cause the CPU to retrieve a stack frame and return to the correct operating mode.

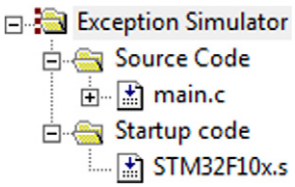
Interrupt Handling—Exit: Important!

The interrupt lines that connect the user peripheral interrupt sources to the NVIC interrupt channels can be level sensitive or edge sensitive. In many microcontrollers, the default is level sensitive. This means that once an interrupt has been raised, it will be asserted on the NVIC until it is cleared. This means that if you exit an ISR with the interrupt still asserted on the NVIC, a new interrupt will be raised. To cancel the interrupt, you must clear the interrupt status flags in the user peripheral before exiting the ISR. If the peripheral generates another interrupt while its interrupt line is asserted, a further interrupt will not be raised. If you clear the interrupt status flags at the beginning of the interrupt routine, then any further interrupts from the peripheral will be served. To further complicate things, some peripherals will automatically clear some of their status flags. For example, an ADC conversion complete flag may be automatically cleared when the ADC results register is read. Keep this in mind when you are reading the microcontroller user manual.

Exercise: Systick Interrupt

This project demonstrates setting up an interrupt using the systick timer.

Open the project in c:\exercises\exception.



This application consists of the minimum amount of code necessary to get the Cortex-M processor running and to generate a systick interrupt.

Open the main.c file.

```
#include "stm32f10x.h"
#define SYSTICK_COUNT_ENABLE  1
#define SYSTICK_INTERRUPT_ENABLE  2
int main (void)
{
GPIOB->CRH = 0x33333333;
SysTick->VAL = 0x9000;
SysTick->LOAD = 0x9000;
SysTick->CTRL = SYSTICK_INTERRUPT_ENABLE | SYSTICK_COUNT_ENABLE;
while(1);
}
```

The main function configures a bank of port pins as outputs. Next, we load the systick timer and reload the register and then enable the timer and its interrupt line to the NVIC. Once this is done the background code sits in a while loop doing nothing.

When the timer counts down to zero, it will generate an interrupt that will run the systick ISR.

```
void SysTick_Handler (void)
{
static unsigned char count = 0;
if(count++ > 0x60){
GPIOB->ODR ^ = 0xFFFFFFFF;
count = 0;
}}
```

The interrupt routine is then used to periodically toggle the GPIO lines.

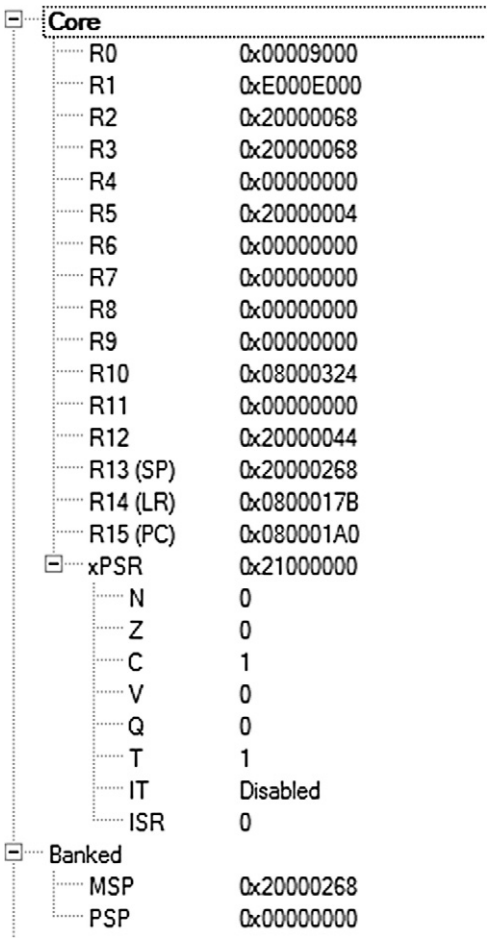
Open the STM32F10x.s file and locate the vector table.

```
SysTick_Handler PROC
  EXPORT SysTick_Handler  [WEAK]
  B  .
  ENDP
```

The vector table provides standard labels for each interrupt source created as "weak" declarations. To create a C ISR, we simply need to use the label name as the name for a void function. The C function will then override the assembled stub and be called when the interrupt is raised.

Build the project and start the debugger.

Without running the code, open the register window and examine the state of the registers.

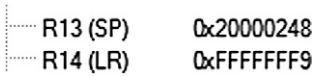


In particular, note the value of the stack pointer (R13), the link register (R14), and the PSR.

Set a breakpoint in the interrupt routine and start running the code.

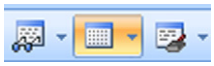
```
26  void SysTick_Handler ( void)
27  {
28  static unsigned char count = 0;
29    if(count++>0x60)
30  {
31  GPIOB->ODR ^=0xFFFFFFFF;
32  count = 0;
33  }
34  }
```

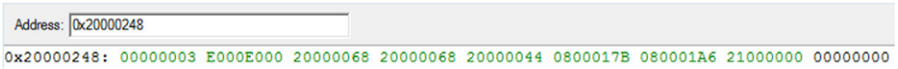
When the code hits the breakpoint again examine the register window.



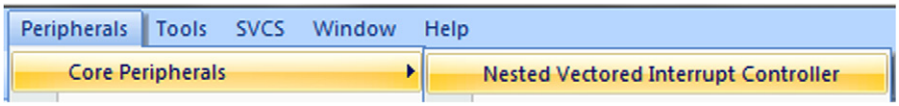
Now, R14 has the interrupt return code in place of a normal return address and the stack pointer has been decremented by 32 bytes.

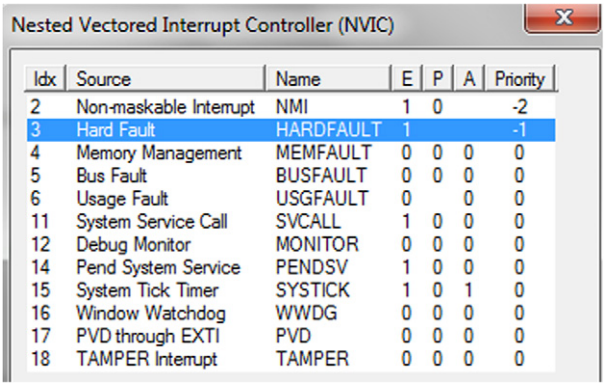
Open a memory window at 0x20000248 and decode the stack frame.





Now open the Peripherals\Core Peripherals\Nested Vectored Interrupt Controller menu.





Idx	Source	Name	E	P	A	Priority
2	Non-maskable Interrupt	NMI	1	0		-2
3	Hard Fault	HARDFAULT	1			-1
4	Memory Management	MEMFAULT	0	0	0	0
5	Bus Fault	BUSFAULT	0	0	0	0
6	Usage Fault	USGFAULT	0		0	0
11	System Service Call	SVCALL	1	0	0	0
12	Debug Monitor	MONITOR	0	0	0	0
14	Pend System Service	PENDSV	1	0	0	0
15	System Tick Timer	SYSTICK	1	0	1	0
16	Window Watchdog	WWDG	0	0	0	0
17	PVD through EXTI	PVD	0	0	0	0
18	TAMPER Interrupt	TAMPER	0	0	0	0

xPSR	0x2100000F
N	0
Z	0
C	1
V	0
Q	0
T	1
IT	Disabled
ISR	15

The NVIC peripheral window shows the state of each interrupt line. Line 15 is the systick timer and it is enabled and active (P = pending). This also ties up with the ISR channel number in the PSR.

Now set a breakpoint on the closing brace of the interrupt function and run the code.

```
26  void SysTick_Handler ( void)
27  {
28  static unsigned char count = 0;
29    if(count++>0x60)
30  {
31  GPIOB->ODR ^=0xFFFFFFFF;
32  count = 0;
33  }
34  }
```

Now open the disassembly window and view the return instruction.

```
   32: count = 0;
   33: }
x080001C6 2000      MOVS     r0,#0x00
x080001C8 4611      MOV      r1,r2
x080001CA 7008      STRB     r0,[r1,#0x00]
   34: }
x080001CC 4770      BX       lr
```

The return instruction is a branch instruction, same as if you were returning from a subroutine. However, the code in the link register (R14) will force the CPU to unstack and

return from the interrupt. Single step this instruction (F11) and observe the return to the background code and restoration of the stacked values to the CPU registers.

Cortex-M Processor Exceptions

In addition to the peripheral interrupt lines, the Cortex-M processor has some internal exceptions and these occupy the first 15 locations of the vector table.

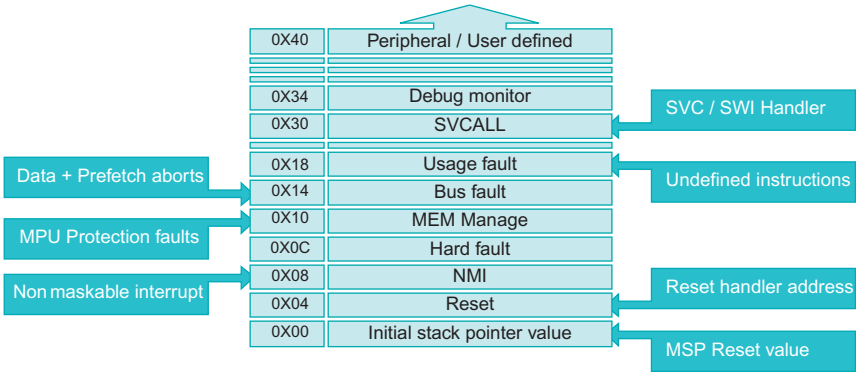


Figure 3.13
The first four bytes of memory hold the initial stack value. The vector table starts from 0x00000004. The first 10 vectors are for the Cortex processor while the remainder are for user peripherals.

The first location in the vector table is the reset handler. When the Cortex-M processor is reset, the address stored here will be loaded into the Cortex-M program counter forcing a jump to the start of your application code. The next location in the vector table is for a nonmaskable interrupt. How this is implemented will depend on the specific microcontroller you are using. It may, for example, be connected to an external pin on the microcontroller or to a peripheral such as a watchdog within the microcontroller. The next four exceptions are for handling faults that may occur during execution of the application code. All of these exceptions are present on the Cortex-M3 and Cortex-M4 but only the hard fault handler is implemented on the Cortex-M0. The types of faults that can be detected by the processor are usage fault, bus fault, memory manager fault, and hard fault.

Usage Fault

A usage fault occurs when the application code has incorrectly used the Cortex-M processor. The typical cause is when the processor has been given an invalid opcode to execute. Most ARM compilers can generate code for a range of ARM processor cores.

So, it is possible to incorrectly configure the compiler and to produce code that will not run on a Cortex-M processor. Other causes of a usage fault are shown below.

Table 3.7: Possible Causes of the Usage Fault Exception

Undefined instruction
Invalid interrupt return address
Unaligned memory access using load and store multiple instructions
Divide by zero*
Unaligned memory access*

*This feature must be enabled in the SCB configurable fault usage register.

Bus Fault

A bus fault is raised when an error is detected on the AHB bus matrix (see more about the bus matrix in Chapter 5). The potential reasons for this fault are as follows.

Table 3.8: Possible Causes of the Bus Fault Exception

Invalid memory region
Wrong size of data transfer, that is, a byte write to a word-only peripheral register
Wrong processor privilege level (we will look at privilege levels in Chapter 5)

Memory Manager Fault

The MPU is an optional Cortex-M processor peripheral that can be added when the microcontroller is designed. It is available on all variants except the Cortex-M0. The MPU is used to control access to different regions of the Cortex-M address space depending on the operating mode of the processor. This will be looked at in more detail in Chapter 5. The MPU will raise an exception in the following cases.

Table 3.9: Possible Causes of the Memory Manager Fault Exception

Accessing an MPU region with the wrong privilege level
Writing to a read-only region
Accessing a memory location outside of the defined MPU regions
Program execution from memory region that is defined as nonexecutable

Hard Fault

A hard fault can be raised in two ways. First, if a bus error occurs when the vector table is being read. Secondly, the hard fault exception is also reached through fault escalation. This means that if the usage, memory manager, or bus fault exceptions are disabled, or if the exception service does not have sufficient priority level, then the fault will escalate to a hard fault.

Enabling Fault Exceptions

The hard fault handler is always enabled and can only be disabled by setting the CPU FAULTMASK register. The other fault exceptions must be enabled in the SCB, system handler control (SHC), and state register (SR) (SCB- > SHCSR). The SCB- > SHCSR register also contains pend and active bits for each fault exception.

We will look at the fault exceptions and tracking faults in Chapter 8.

Priority and Preemption

The NVIC contains a group of priority registers with an 8-bit field for each interrupt source. In its default configuration, the top 7 bits of the priority register allow you to define the preemption level. The lower the preemption level, the more important the interrupt. So, if an interrupt is being served and a second interrupt is raised with a lower preemption level, then the state of the current interrupt will be saved and the processor will serve the new interrupt. When it is finished, the processor will resume serving the original interrupt provided a higher-priority interrupt is not pending.

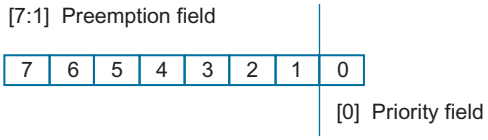


Figure 3.14
Each peripheral priority register consists of a configurable preemption field and a subpriority field.

The least significant bit is the subpriority bit. If two interrupts are raised with the same preemption level, the interrupt with the lowest subpriority level will be served first. This means we have 128 preemption levels each with two subpriority levels.

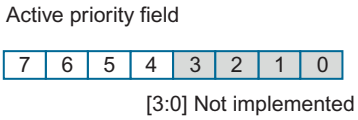


Figure 3.15
Each priority register is 8-bits wide. However, the silicon manufacturer may not implement all of the priority bits. The implemented bits always extend from the most significant bits (MSB) toward the least significant bits (LSB).

When the microcontroller is designed, the manufacturer can define the number of active bits in the priority register. For the Cortex-M3 and Cortex-M4, this can be a minimum of three and up to a maximum of eight. For the Cortex-M0, Cortex-M0+, and Cortex-M1, it

is always 2 bits. Reducing the number of active priority bits reduces the NVIC gate count and hence its power consumption. If the manufacturer does not implement the full 8 bits of the priority register, the LSB will be disabled. This makes it safer to port code between microcontrollers with different numbers of active priority bits. You will need to check the manufacturer's datasheet to see how many bits of the priority register are active.

Groups and Subgroups

By default, in NVIC the first 7 bits of the priority register define the preemption level and the LSB defines the subpriority level. This split between preemption group and priority subgroup can be modified by writing to the NVIC priority group field in the application interrupt and reset control (AIRC) register. This register allows us to change the size of the preemption group field and priority subgroup. On reset, this register defaults to priority group zero.

Table 3.10: Priority Group and Subgroup Values

Priority Group	Preempt Group Bits	Subpriority Group Bits
0	7—1	0
1	7—2	1—0
2	7—3	2—0
3	7—4	3—0
4	7—5	4—0
5	7—6	5—0
6	7	6—0
7	None	7—0

So, for example, if our microcontroller has four active priority bits we could select priority group 5, which would give us four levels of preemption each with four levels of subpriority.

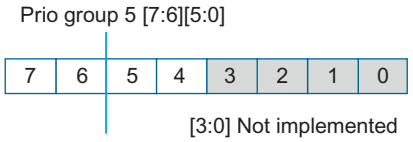


Figure 3.16
A priority register with four active bits and priority group 5. This yields four preempt levels and four priority levels.

The highest preemption level for a user exception is zero, however, some of the Cortex-M processor exceptions have negative priority levels so they will always preempt a user interrupt.

	Exception	Name	Priority	Descriptions
Fault Mode and Startup Handlers	1	Reset	–3 (Highest)	Reset
	2	NMI	–2	Nonmaskable Interrupt
	3	Hard fault	–1	Default fault if other hander not implemented
	4	Memory manage fault	Programmable	MPU violation or access to illegal locations
	5	Bus fault	Programmable	Fault if AHB interface receives error
	6	Usage fault	Programmable	Exceptions due to program errors
System Handlers	11	SVCall	Programmable	System service call
	12	Debug monitor	Programmable	Breakpoints, watch points, external debug
	14	PendSV	Programmable	Pendable service request for System Device
	15	Systick	Programmable	System Tick Timer
Custom Handlers	16	Interrupt #0	Programmable	External interrupt #0
	...	...	...	...
	...	...	...	...
	...	...	...	...
	255	Interrupt #239	Programmable	External interrupt #239

Figure 3.17
Cortex-M processor exceptions and possible priority levels.

Run Time Priority Control

There are three CPU registers that may be used to dynamically disable interrupt sources within the NVIC. These are the PRIMASK, FAULTMASK, and BASEPRI registers.

Table 3.11: The CPU PRIMASK, FAULTMASK, and BASEPRI Registers Are Used to Dynamically Disable Interrupts and Exceptions

CPU Mask Register	Description
PRIMASK	Disables all exceptions except hard fault and NMI
FAULTMASK	Disables all exceptions except NMI
BASEPRI	Disables all exceptions at the selected preemption level and lower preempt level

These registers are not memory mapped; they are CPU registers and may only be accessed with the MRS and MSR instructions. When programming in C, they may be accessed by dedicated compiler intrinsic instructions; we will look at these intrinsics more closely in Chapter 4.

Exception Model

When the NVIC serves a single interrupt, there is a delay of 12 cycles until we reach the ISR and a further 10 cycles at the end of the ISR until the Cortex-M processor resumes

execution of the background code. This gives us fast deterministic handling of interrupts in a system that may have only one or two active interrupt sources.

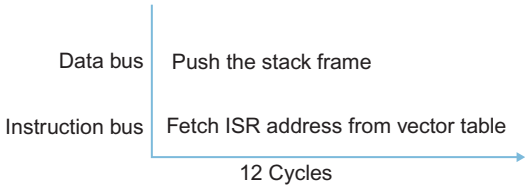


Figure 3.18
When an exception is raised, a stack frame is pushed in parallel with the ISR address being fetched from the vector table. On the Cortex-M3 and Cortex-M4 this is always 12 cycles. On the Cortex-M0, it takes 16 cycles. The Cortex-M0+ takes 15 cycles.

In more complex systems, there may be many active interrupt sources all demanding to be served as efficiently as possible. The NVIC has been designed with a number of optimizations to ensure optimal interrupt handling in such a heavily loaded system. All of the interrupt handling optimizations described below are an integral part of the NVIC and as such are performed automatically by the NVIC and do not require any configuration by the application code.

NVIC Tail Chaining

In a very interrupt-driven design, we can often find that while the CPU is serving a high-priority interrupt a lower-priority interrupt is also pending. In the earlier ARM CPUs and many other processors, it was necessary to return from the interrupt by POPing the CPU context from the stack back into the CPU registers and then performing a fresh stack PUSH before running the pending ISR. This is quite wasteful in terms of CPU cycles as it performs two redundant stack operations.

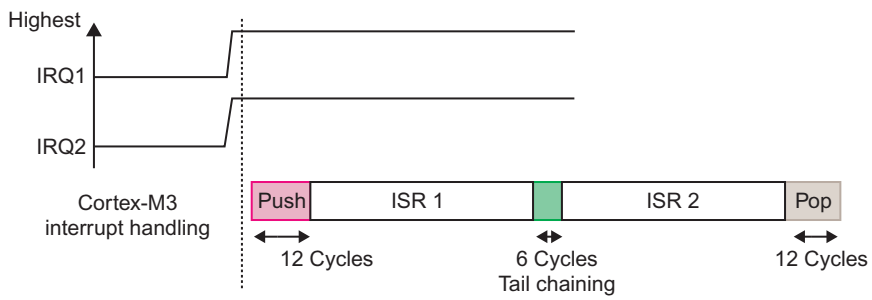


Figure 3.19
If an interrupt ISR is running and a lower-priority interrupt is raised, it will be automatically "tail chained" to run six cycles after the initial interrupt has terminated.

When this situation occurs on a Cortex-M processor, the NVIC uses a technique called tail chaining to eliminate the unnecessary stack operations. When the Cortex processor reaches the end of the active ISR and there is a pending interrupt, the NVIC simply forces the processor to vector to the pending ISR. This takes a fixed six cycles to fetch the start address of the pending interrupt routine and then execution of the next ISR can begin. Any further pending interrupts are dealt with in the same way. When there are no further interrupts pending, the stack frame will be POPed back to the processor registers and the CPU will resume execution of the background code. As you can see from Figure 3.19, tail chaining can significantly improve the latency between interrupt routines.

NVIC Late Arriving

Another situation that can occur is a "late arriving" high-priority interrupt. In this situation, a low-priority interrupt is raised followed almost immediately by a high-priority interrupt. Most microcontrollers will handle this by preempting the initial interrupt. This is undesirable because it will cause two stack frames to be pushed and delay the high-priority interrupt.

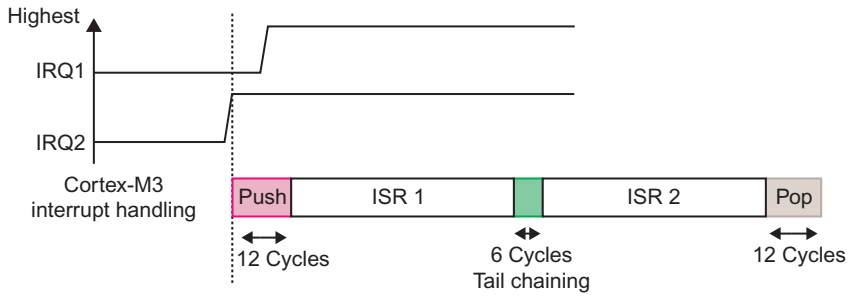


Figure 3.20
If the Cortex-M processor is entering the ISR and a higher-priority interrupt is raised, the NVIC will automatically switch to serve the high-priority interrupt. This will only happen if the initial interrupt is in its first 12 cycles.

If this situation occurs on a Cortex-M processor and the high-priority interrupt arrives within the initial 12-cycle PUSH of the low-priority stack frame, then the NVIC will switch to serving the high-priority interrupt and the low-priority interrupt will be tail chained to execute once the high-priority interrupt is finished. For the "late arriving" switch to happen, the high-priority interrupt must occur in the initial 12-cycle period of the low-priority interrupt. If it occurs any later than this, then it will preempt the low-priority interrupt, which requires the normal stack PUSH and POP.

NVIC POP Preemption

The final optimization technique used by the NVIC is called POP preemption. This is kind of a reversal of the late arriving technique discussed above.

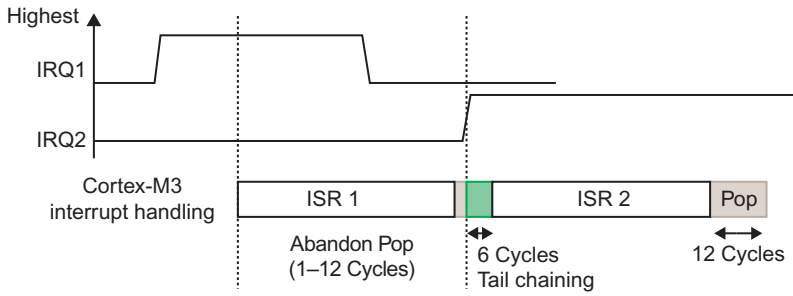


Figure 3.21
If an interrupt is raised while it is in its exiting 12 cycles, the processor will "rewind" the stack and serve the new interrupt with a minimum delay of six cycles.

When a typical microcontroller reaches the end of an ISR, it always has to restore the stack frame regardless of any pending interrupts. As we have seen above, the NVIC will use tail chaining to efficiently deal with any currently pending interrupts. However, if there are no pending interrupts, the stack frame will be restored to the CPU registers in the standard 12 cycles. If during this 12-cycle period a new interrupt is raised, the POPing of the stack frame will be halted and the stack pointer will be wound back to the beginning of the stack frame, the new interrupt vector will be fetched, and the new ISR will be executed. At the end of the new interrupt routine, we return to the background code through the usual 12-cycle POP process. This technique is called POP preemption.

It is important to remember that these three techniques, tail chaining, late arriving, and POP preemption, are all handled by the NVIC without any instructions being added to your application code.

Exercise: Working with Multiple Interrupts

This exercise extends our original systick exception exercise to enable a second ADC interrupt. We can use these two interrupts to examine the behavior of the NVIC when it has multiple interrupt sources.

Open the project in c:\exercises\multiple exceptions.

```
volatile unsigned char BACKGROUND = 0;unsigned char ADC = 0;unsigned char SYSTICK = 0;
int main (void){
int i;
```

```
GPIOB->CRH = 0x33333333;    //Configure the Port B LED pins
SysTick->VAL = 0x9000;    //Start value for the systick counter
SysTick->LOAD = 0x9000;    //Reload value
SysTick->CTRL = SYSTICK_INTERRUPT_ENABLE|SYSTICK_COUNT_ENABLE;
init_ADC();    //setup the ADC peripheral
ADC1->CR1|= (1UL << 5);    // enable for EOC Interrupt
NVIC->ISER[0]=(1UL << 18);    // enable ADC Interrupt
ADC1->CR2|= (1UL << 0);    // ADC enable
while(1){
  BACKGROUND=1;
}}
```

We initialize the systick timer in the same manner as before. In addition, the ADC peripheral is also configured. To enable a peripheral interrupt, it is necessary to enable the interrupt source in the peripheral and also enable its interrupt channel in the NVIC by setting the correct bit in the NVIC Interrupt Set Enable Register ISER registers.

We have also added three variables: BACKGROUND, ADC, and SYSTICK. These will be set to logic one when the matching region of code is executing and zero at other times. This allows us to track execution of each region of code using the debugger logic analyzer.

```
void ADC_IRQHandler (void){
int i;
BACKGROUND    = 0;
SYSTICK    = 0;
for (i=0;i<0x1000;i++){
  ADC =1;
}
ADC1->SR &= ~(1 << 1);      /* clear EOC interrupt      */
ADC =0;
}
```

The ADC interrupt handler sets the execution region variables then sits in a delay loop. Before exiting it also writes to the ADC status register to clear the end of conversion flag. This deasserts the ADC interrupt request to the NVIC.

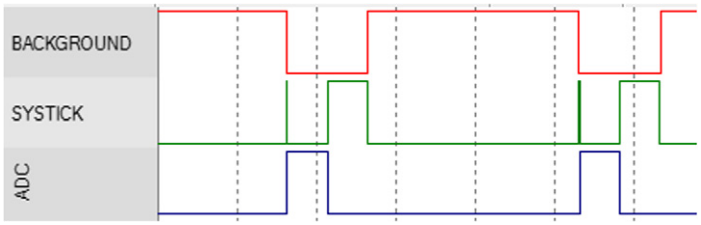
```
void SysTick_Handler (void){
int i;
BACKGROUND=0;
ADC=0;
ADC1->CR2|= (1UL << 22);
for (i=0;i<0x1000;i++){
```

```
    SYSTICK = 1;
  }
  SYSTICK = 0;
  }
```

The systick interrupt handler is similar to the ADC handler. It sets the region execution variables and sits in a delay loop before exiting. It also writes to the ADC control register to trigger a single ADC conversion.

Build the project and start the simulator.

Add each of the execution variables to the logic analyzer and start running the code.

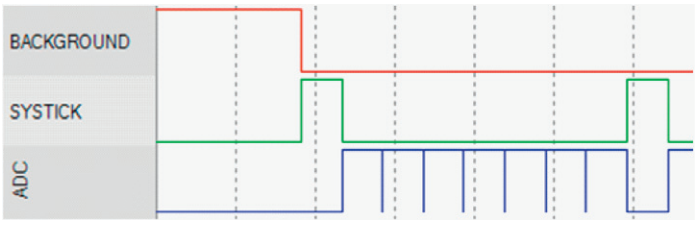


The systick interrupt is raised, which starts the ADC conversion. The ADC finishes conversion and raises its interrupt before the systick interrupt completes so it enters a PEND state. When the systick interrupt completes, the ADC interrupt is tail chained and begins execution without returning to the background code.

Exit the debugger and comment out the line of code that clears the ADC at the end of conversion flag.

```
//ADC1->SR &= ~(1 << 1);
```

Build the code and restart the debugger and observe the execution of the interrupts in the logic analyzer window.



After the first ADC interrupt has been raised, the ADC 'End Of Conversion' interrupt status flag will be set and the ADC interrupt line to the NVIC stays asserted. If you quit the interrupt service routine without clearing all the pending interrupt status flags continuous

ADC interrupts to be raised by the NVIC, blocking the activity of the background code. The systick interrupt has the same priority as the ADC so it will be tail chained to run after the current ADC interrupt has finished. Neglecting to clear interrupt status flags is the most common mistake made when first starting to work with the Cortex-M processors.

Exit the debugger and uncomment the end of conversion code.

```
ADC1->SR &= ~(1 << 1);
```

Add the following lines to the background initializing code.

```
NVIC->IP[18]=(2 << 4);
SCB->SHP[11]=(3 << 4);
```

This programs the user peripheral NVIC interrupt priority registers to set the ADC priority level and the system handler priority registers to set the systick priority level. These are both byte arrays that cover the 8-bit priority field for each exception source. However, on this microcontroller the manufacturer has implemented four priority bits out of the possible eight. The priority bits are located in the upper nibble of each byte. On reset, the PRIGROUP is set to zero, which creates a 7-bit preemption field and 1-bit priority field.

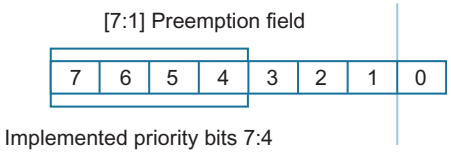
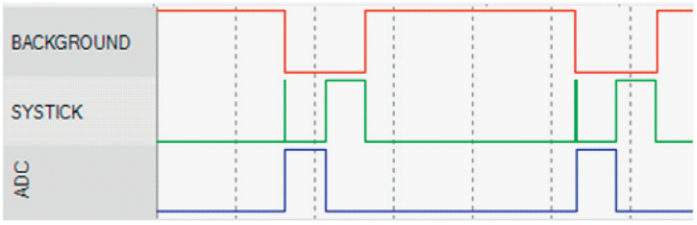


Figure 3.22
After reset, a microcontroller with four implemented priority bits will have 16 levels of preemption.

On our device all of the available priority bits are located in the preemption field giving us 16 levels of priority preemption.

Build the code, restart the debugger, and observe the execution of the interrupts in the logic analyzer window.



The ADC now has the highest preemption value so as soon as its interrupt is raised, it will

preempt the systick interrupt. When it completes, the systick interrupt will resume and complete before returning to the background code.

Exit the debugger and uncomment the following lines in the background initialization code.

The AIRC register cannot be written too freely. It is protected by a key field that must be programmed with the value 0x5FA before a write is successful.

```
temp = SCB->AIRCR;
temp &= ~(SCB_AIRCR_VECTKEY_Msk | SCB_AIRCR_PRIGROUP_Msk);
temp = (temp|((uint32_t)0x5FA << 16) | (0x05 << 8));
SCB->AIRCR = temp;
```

This programs the PRIGROUP field in the AIRC register to a value of 5, which means a 2-bit preemption field and a 6-bit priority field. This maps onto the available 4-bit priority field giving four levels of preemption each with four levels of priority.

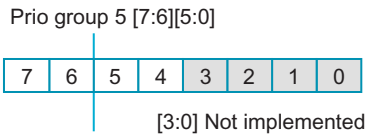
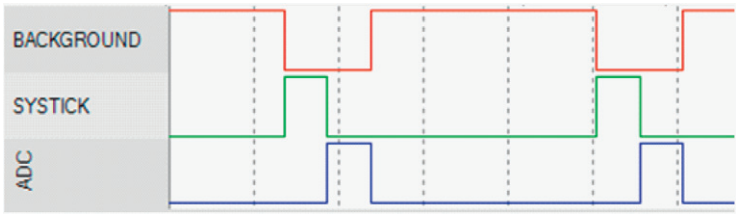


Figure 3.23
When the PRIGROUP field in the AIRC register is set to 5 each priority register has a 2-bit preemption field and a 6-bit priority field.

Build the code, restart the debugger, and observe the execution of the interrupts in the logic analyzer window.



The ADC interrupt is no longer preempting the systick timer despite them having different values in their priority registers. This is because they now have different values in the priority field but with the same preempt value.

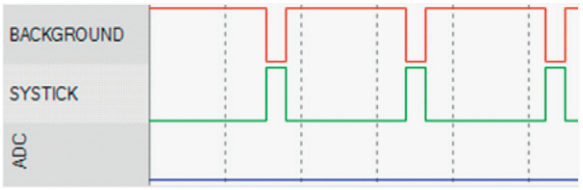
Exit the debugger and change the interrupt priorities as shown below.

```
NVIC->IP[18] = (2<<6 | 2<<4);
SCB->SHP[11] = (1<<6 | 3<<4);
```

Set the base priority register to block the ADC preempt group.

```
__set_BASEPRI  (  2<<6  );
```

Build the code, restart the debugger, and observe the execution of the interrupts in the logic analyzer window.



Setting the BASEPRI register has disabled the ADC interrupt and any other interrupts that are on the same level of preempt group or lower.

Bootloader Support

While the interrupt vector table is located at the start of memory when the Cortex-M processor is reset, it is possible to relocate the vector table to a different location in memory. As software embedded in small microcontrollers becomes more sophisticated, there is an increasing need to develop systems with a permanent bootloader program that can check the integrity of the main application code before it runs and checks for a program update that can be delivered by various serial interfaces (e.g., Ethernet, USB, UART) or an SD/multimedia card.

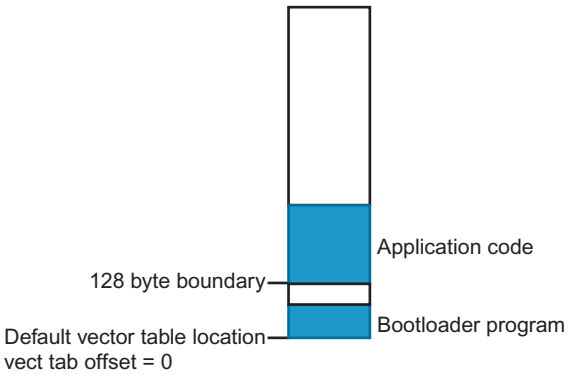


Figure 3.24
A bootloader program can be placed into the first sector in the flash memory. It will check if there is an update to the application code before starting the main application program.

Once the bootloader has performed its checks and, if necessary, updated the application program, it will jump the program counter to the start of the application code, which will start running. To operate correctly, the application code requires the vector table to be mapped to the start address of the application code.

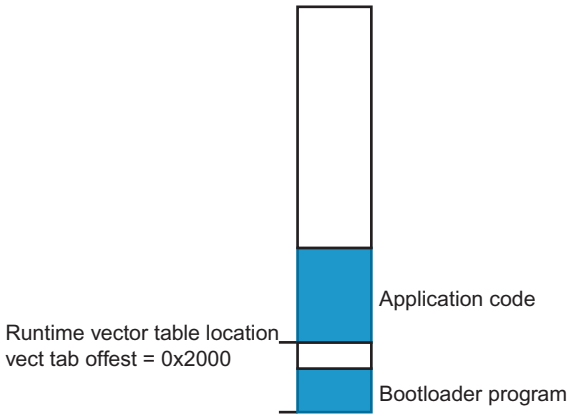


Figure 3.25
When the application code starts to run, it must relocate the vector table to the start of the application code by programming the NVIC vector table offset register.

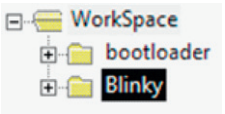
The vector table can be relocated by writing to a register in the SCB called the vector table offset register. This register allows you to relocate the vector table to any 128-byte boundary in the Cortex processor memory map.

Exercise: Bootloader

This exercise demonstrates how a bootloader and application program can both be resident on the same Cortex-M microcontroller and how to debug such a system.

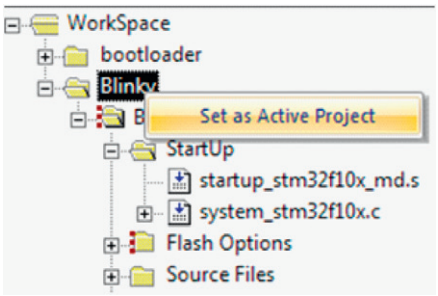
Open the multiworkspace project in C:\exercises\bootloader.

This is a more advanced feature of the μVision IDE that allows you to view two or more projects at the same time.

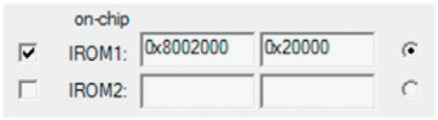


The workspace consists of two projects, the bootloader project, which is built to run on the Cortex processor reset vector as normal, and the blinky project, which is our application. First, we need to build the blinky project to run from an application address that is not in the same flash sector as the bootloader. In this example, the application address is chosen to be 0x2000.

Expand the blinky project, right click on the workspace folder, and set it as the active project.



Now click on the blinky project folder and open the Options for Target\Target tab.



The normal start address for this chip is 0x8000000 and we have increased this to 0x8002000.

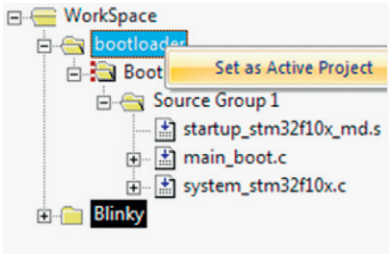
Open the system_stm32F10x.c file and locate line 128.

```
#define VECT_TAB_OFFSET 0x02000
```

This contains a define for the vector table offset register. Normally, this is zero but if we set this to 0x2000 the vector table will be remapped to match our application code when it starts running.

Build the blinky project.

Expand the bootloader project and set it as the active project.



Open main_boot.c.

The bootloader program demonstrates how to jump from one program to the start of another. We need to define the start address of our second program. This must be a multiple of 128 bytes (0x200). In addition, a void function pointer is also defined.

```
#define APPLICATION_ADDRESS 0x2000
typedef void (*pFunction)(void);
pFunction Jump_To_Application;
uint32_t JumpAddress;
```

When the bootloader code enters main, it performs a custom check on the application code flash, such as a checksum, and could also test other critical aspects of the hardware. The bootloader then checks to see if there is a new application ready to be programmed into the application area. This could be in response to a command from an upgrade utility via a serial interface, for example. If the application program checks fail or a new update is available, we enter into the main bootloader code.

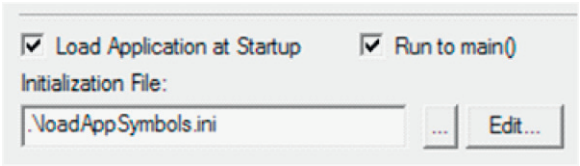
```
int main(void) {
uint32_t bootFlags;
  /* check the integrity of the application code */
  /* check if an update is available */
  /* if either case is true set a bit in the bootflags register */
  bootFlags = 0;
  if (bootFlags != 0) {
  //enter the flash update code here
  }
```

If the application code and the hardware is OK, then the bootloader will handover to the application code. The reset vector of the application code is now located at the application address + 4. This can be loaded into the function pointer, which can then be executed resulting in a jump to the start of the application code. Before we jump to the application code, it is also necessary to load the stack pointer with the start address expected by the application code.

```
else {
    JumpAddress = *(__IO uint32_t*) (APPLICATION_ADDRESS + 4);
    Jump_To_Application = (pFunction) JumpAddress;
    // read the first four bytes of the application code and program this value into the
       stack pointer;
    //This sets the stack ready for the application code
    __set_MSP(*(__IO uint32_t*) APPLICATION_ADDRESS);
    Jump_To_Application();
  }}
```

Build the project.

Open the bootloader Options for Target\Debug tab and open the loadApp.ini file.



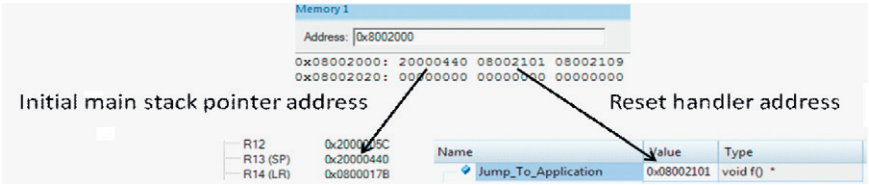

```
Load "C:\\Cortex Microcontroller Tutorial Exercises\\Bootloader\\Blinky\\Flash\
\Blinky.AXF" incremental
```

This script file can be used with the simulator or the hardware debugger. It is used to load the blinky application code as well as the bootloader code. This allows us to debug seamlessly between the two separate programs.

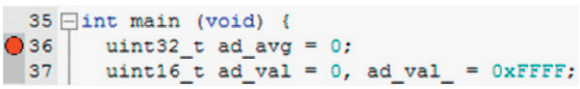
Start the debugger.

Single step the code through the bootloader checking that the correct stack pointer address is loaded into the main stack pointer and that the blinky start address is loaded into the function pointer.

Use the memory window to view the blinky application, which starts at 0×800200, and check that the stack pointer value is loaded into R13 and the reset address is loaded into the function pointer.



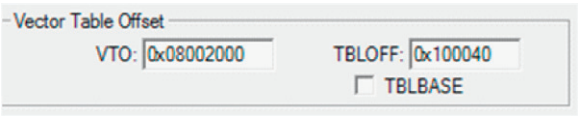
Open the blinky.c file in the blinky project and set a breakpoint on main.



Run the code.

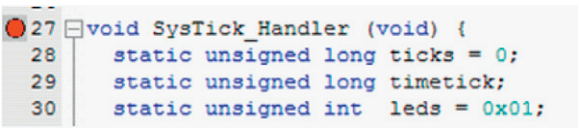
Now the Cortex processor has left the bootloader program and entered the blinky application program. The startup system code has programmed the vector table offset register so now the hardware vector table matches the blinky software.

Open the Peripherals\Core Peripherals\Nested Vector Interrupt Table.



Now the vector table is at 0x8002000 to match the blinky code.

Open the IRQ.C file and set a breakpoint on the systick interrupt handler.



Run the code.

When the systick timer raises its interrupt, the handler address will be fetched from the blinky vector table and not from the default address at the start of the memory and the correct systick handler will be executed. Now the blinky program is running happily at its offset address. If you need the application code to call the bootloader, then set a pattern in memory and force a reset within the application code by writing to the SCB application interrupt and reset the control register.

```
Shared_memory = USER_UPDATE_COMMAND
NVIC_SystemReset();
```

When the bootloader restarts part of its startup checks will be to test the shared_memory location and if it is set run the required code.

Power Management

While the Cortex-M0 and Cortex-M0+ are specifically designed for low-power operation the Cortex-M3 and Cortex-M4 still have remarkably low power consumption. While the actual power consumption will depend on the manufacturing process used by the silicon vendor, the figures below give an indication of expected power consumption.

Table 3.12: Power Consumption Figures by Processor Variant in 90 nm LP (Low-Power) Process

Processor	Dynamic Power Consumption (μW/MHz)	Details
Cortex-M0+	11	Excludes debug units
Cortex-M0	16	Excludes debug units
Cortex-M3	32	Excludes MPU and debug units
Cortex-M4	33	Excludes FPU, MPU, and debug units

The Cortex-M processors are capable of entering low-power modes called SLEEP and DEEPSLEEP. When the processor in placed in SLEEP mode, the main CPU clock signal is stopped, which halts the Cortex-M processor. The rest of the microcontroller clocks and peripherals will still be running and can be used to wake up the CPU. The DEEPSLEEP mode is an extension of the SLEEP mode and its action will depend on the specific implementation made on the microcontroller. Typically, when the DEEPSLEEP mode is entered, the processor peripheral clocks will also be halted along with the Cortex-M processor clock. Other areas of the microcontroller such as the on-chip SRAM and power to the flash memory may also be switched off depending on the microcontroller configuration.

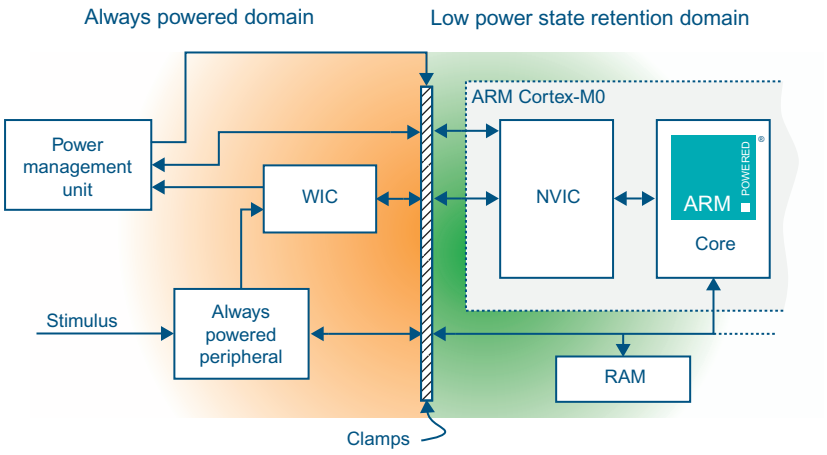


Figure 3.26

The WIC is a small area of gates that do not require a clock source. The WIC can be located on a different power domain to the Cortex-M processor. This allows all the processor clocks to be halted. The range of available power modes is defined by the silicon manufacturer.

When a Cortex-M processor has entered a low-power sleep mode, it can be woken up by a microcontroller peripheral raising an interrupt to the NVIC. However, the NVIC need clock to operate, so if all the clocks are stopped we need another hardware unit to tell the power management unit (PMU) to restore the clocks before the NVIC can respond. The Cortex-M processors can be fitted with an optional unit called the WIC. The WIC handles interrupt

detection when all clocks are stopped and allows all of the Cortex processors to enter low-power modes. The WIC consists of a minimal number of gates and does not need a system clock. The WIC can be placed on a different power domain to the main Cortex-M processor. This allows the microcontroller manufacturers to design a device that can have low-power modes where most of the chip is switched off while keeping key peripherals alive to wake up the processor.

Entering Low-Power Modes

The Thumb-2 instruction set contains two dedicated instructions that will place the Cortex processor into SLEEP or DEEPSLEEP mode.

Table 3.13: Low-Power Entry Instructions

Instruction	Description	CMSIS-Core Intrinsic
WFI	Wait for interrupt	__WFI()
WFE	Wait for event	__WFE

As its name implies the WFI instruction will place the Cortex-M processor in the selected low-power mode. When an interrupt is received from one of the microcontroller peripherals, the processor will exit low-power mode and resume processing the interrupt as normal. The wait for event instruction is also used to enter the low-power modes but has some configuration options as we shall see next.

Configuring the Low-Power Modes

The system control register (SCR) is used to configure the Cortex-M processor's low-power options.

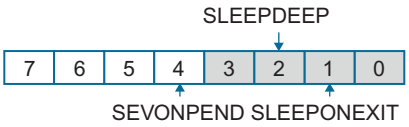


Figure 3.27
The SCR contains the Cortex-M processor's low-power configuration bits.

The Cortex-M processor has two external sleep signals which are connected to the power management system designed by the manufacturer. By default, the SLEEPING signal is activated when the WFI or WFE instructions are executed. If the SLEEPDEEP bit is set, the second sleep signal SLEEPDEEP is activated when the Cortex-M processor enters a sleeping mode. These two sleep signals are used by the microcontroller manufacturer to provide a broader power management scheme for the microcontroller. Setting the sleep on exit bit will force the microcontroller to enter its SLEEP mode when it reaches the end of an interrupt. This allows you to design a system that wakes up in response to an interrupt,

runs the required code, and then automatically returns to sleep. In such a system no stack management is required (except in the case of preempted interrupts) during an interrupt entry/exit sequence and no background code will be executed. The WFE instruction places the Cortex-M processor into its sleeping mode and the processor may be woken by an interrupt in the same manner as the WFI instruction. However, the WFE instruction has an internal event latch. If the event latch is set to one of the processor, it will clear the latch but do not enter low-power mode. If the latch is zero, it will enter low-power mode. On a typical microcontroller, the events are the peripheral interrupt signals. So pending interrupts will prevent the processor from sleeping. The SEVONPEND bit is used to change the behavior of the WFE instruction. If this bit is set, the peripheral interrupt lines can be used to wake the processor even if the interrupt is disabled in the NVIC. This allows you to place the processor into its sleep mode, and when a peripheral interrupt occurs the processor will wake and resume execution of the instruction following the WFE instruction rather than jumping to an interrupt routine.

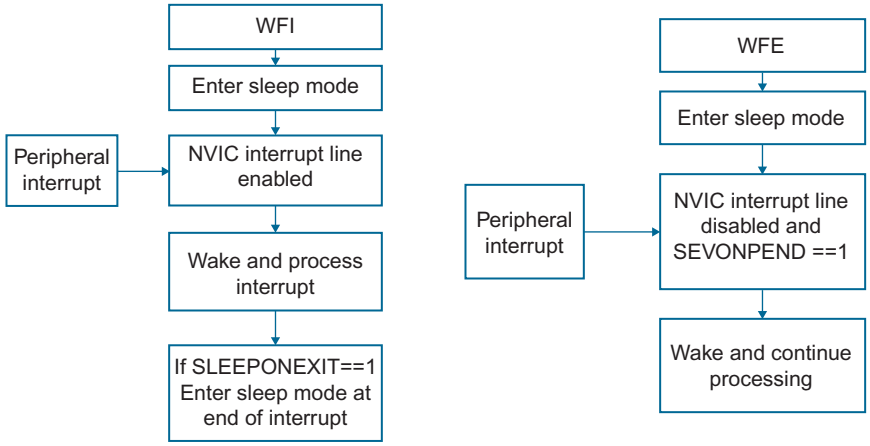


Figure 3.28
The two low-power entry instructions place the Cortex processor into its low-power mode. Both modes use a peripheral interrupt to wake the processor but their wakeup behavior is different.

An interrupt disabled in the NVIC cannot be used to exit a sleep mode entered by the WFI instruction. However, the WFE instruction will respond to an activity on any interrupt line even if it is disabled or temporarily disabled by the processor mask registers (i.e., BASEPRI, PRIMASK, and FAULTMASK).

Exercise: Low-Power Modes

In this exercise, we will use the exception project to experiment with the Cortex-M low-power modes.

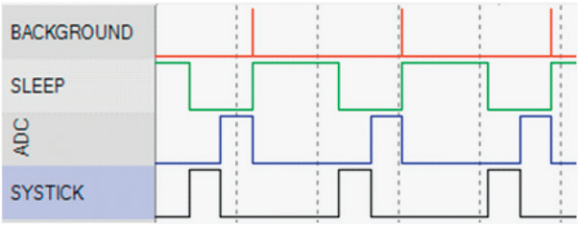
Open the project in c:\execises\low power modes.

```
while(1)
{
  SLEEP = 1;
  BACKGROUND = 0;
  __wfe();
  BACKGROUND = 1;
  SLEEP = 0;
}
```

This project uses the systick and ADC interrupts that we saw in the last example. This time we have added an additional SLEEP variable to monitor the processor operating state. The wait for interrupt instruction has been added in the main while loop.

Build the project and start the debugger.

Open the logic analyzer window and start running the code.



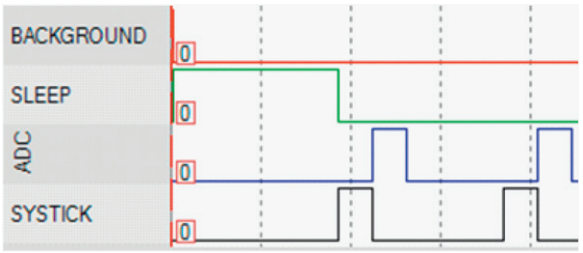
Here, we can see that the background code executes the __WFI() instruction and then goes to sleep until an interrupt is raised. When the interrupts have completed we return to the background code, which will immediately place the processor in its low-power mode.

Exit the debugger and change the code to match the lines below.

```
SCB->SCR = 0x2;
SLEEP = 1;
BACKGROUND = 0;
__wfi();
while(1){
```

Add the code to set bit two of the SCR. This sets the sleep on exit flag, which forces the processor into a low-power mode when it completes an interrupt. Cut the remaining three lines from inside the while loop and paste them into the initializing section of the main() function.

Build the code, restart the debugger, and observe the execution of the interrupts in the logic analyzer window.



Here, we can see that the interrupts are running but after the initializing code has run the background loop never executes so the background and sleep variables are never updated. In the debugger, you will also be able to see from the coverage monitor that the main() while loop is never executed. This feature allows the processor to wake up, run some critical code, and then sleep with the absolute minimum overhead.

Moving from the Cortex-M3

In this chapter, we have concentrated on learning the Cortex-M3 processor. Now that you are familiar with how the Cortex-M3 works, we can examine the differences between the Cortex-M3 and the other Cortex-M variants. As we will see these are mainly architectural differences, and if you can use the Cortex-M3 you can easily move up to a Cortex-M4 or down to a Cortex-M0(+)-based microcontroller. Increasingly silicon manufacturers are producing families of microcontrollers which have variants using different Cortex-M processors. This allows you to seamlessly switch devices trading off performance against cost.

Cortex-M4

The Cortex-M4 is most easily described as a Cortex-M3 with an additional FPU and DSP instructions. We will look at these features in Chapter 7, but here we will take a tour of the main differences between the Cortex-M3 and Cortex-M4. The Cortex-M4 offers the same processing power of 1.25 DMIPS/MHz as the Cortex-M3 but has a much greater math capability. This is delivered in three ways. The hardware FPU can perform calculations in a little as one cycle compared to the hundreds of cycles the same calculation would take on the Cortex-M3. For integer calculations, the Cortex-M4 has a higher performance MAC that improves on the Cortex-M3 MAC to allow single cycle calculations to be performed on 32-bit wide quantities, which yield a 64-bit result. Finally, the Cortex-M4 adds a group of SIMD instructions that can perform multiple integer calculations in a single cycle.

While the Cortex-M4 has a larger gate count than the Cortex-M3, the FPU contains more gates than the entire Cortex-M0 processor.

Table 3.14: Additional Features in the Cortex-M4

Feature	Comments
FPU	See Chapter 7
DSP, SIMD instructions	
GE field in xPSR	
Extended integer MAC unit	The Cortex-M4 extends the integer MAC to support single cycle execution of 32-bit multiplies which yield a 64-bit result

Cortex-M0

The Cortex-M0 is a reduced version of the Cortex-M3; it is intended for low-cost and low-power microcontrollers. The Cortex-M0 has a processing power of 0.84 DMIPS/MHz. While the Cortex-M0 can run at high clock frequencies it is often designed into low-cost devices with simple memory systems. Hence, a typical Cortex-M0 microcontroller runs with a CPU frequency of 50 DMIPS/MHz, however, its low-power consumption makes it ideal for low-power applications. While it essentially has the same programmer's model as the Cortex-M3, there are some limitations and these are summarized below.

Table 3.15: Features Not Included in the Cortex-M0

Feature	Comments
Three-stage pipeline	Same pipeline stages as the Cortex-M3, but no speculative branch target fetch
von Neumann bus interface	Instruction and data use the same bus port which can create slower performance compared to the Harvard architecture of the Cortex-M3/M4
No conditional if then blocks	Conditional branches are always used which cause a pipeline flush
No saturated math instructions	
SYSTICK timer is optional	However, so far every Cortex-M0 microcontroller had it fitted
No MPU	The MPU is covered in Chapter 5
32 NVIC channels	A limited number of interrupt channels compared to the Cortex-M3/M4, however, in practice this is not a real limitation and the Cortex-M0 is intended for small devices with a limited number of peripherals
Four programmable priority levels	Priority level registers only implemented 2 bits (four levels), and there is no priority group setting
Hard fault exception only	No usage fault, memory management fault, or bus fault exception vectors
16 cycle interrupt latency	Same deterministic interrupt handling as the Cortex-M3 but with four more cycles of overhead

(*Continued*)

Table 3.15: (Continued)

Feature	Comments
No priority group No BASEPRI register No Faultmask register	Limited to a fixed four levels of preemption
Reduced debug features	The Cortex-M0 has fewer debug features compared to the Cortex-M3\M4, see Chapter 8 for more details
No exclusive access instructions	The exclusive access instructions are covered in Chapter 5
No reverse bit order of count leading zero instructions	
Reduced number of registers in the SCB	See below
Vector table cannot be relocated	The NVIC does not include the vector table offset register
All code executes at privileged level	The Cortex-M0 does not support the unprivileged operating mode

The SCB contains a reduced number of features compared to the Cortex-M3\M4. Also the systick timer registers have been moved from the NVIC to the SCB.

Table 3.16: Registers in the Cortex-M0 SCB

Register	Size in Words	Description
Systick control and status	1	Enables the timer and its interrupt
Systick reload	1	Holds the 24-bit reload value
Systick current value	1	Holds the current 24-bit timer value
Systick calibration	1	Allows trimming the input clock frequency
CPU ID	1	Hardwired ID and revision numbers from ARM and the silicon manufacturer
Interrupt control and state	1	Provides pend bits for the systick and NMI interrupts and extended interrupt pending\active information
AIRC	1	Contains the same fields as the Cortex-M3 minus the PRIGROUP field
Configuration and control	1	
System handler priority	2	These registers hold the 8-bit priority fields for the configurable processor exceptions

Cortex-M0+

The Cortex-M0+ is the latest Cortex-M processor variant to be released from ARM. It is an enhanced version of the Cortex-M0. As such it boasts lower power consumption figures combined with greater processing power. The Cortex-M0+ also brings the MPU and real-time debug capability to very low-end devices. The Cortex-M0+ also introduces

a fast IO port that speeds up access to peripheral registers, typically GPIO ports, to allow fast switching of port pins. As we will see in Chapter 8, the debug system is fitted with a new trace unit called the MTB which allows you to capture a history of executed code with a low-cost development tool.

Table 3.17: Cortex-M0+ Features

Feature	Comments
Code compatible with the Cortex-M0	The Cortex-M0+ is code compatible with the Cortex-M0 and provides higher performance with lower-power consumption
Two-stage pipeline	This reduces the number of flash accesses and hence power consumption
I/O port	The I/O port provides single cycle access to GPIO and peripheral registers
Vector table can be relocated	This supports more sophisticated software designs. Typically a bootloader and separate application
Supports 16-bit flash memory accesses	This allows devices featuring the Cortex-M0+ to use low-cost memory
Code can execute at privileged and unprivileged levels	The Cortex-M0+ has the same operating modes as the Cortex-M3/M4
MPU	The Cortex-M0+ has a similar MPU to the Cortex-M3/M4
MTB	This is a "snapshot" trace unit which can be accessed by low cost debug units