CHAPTER 4

Cortex Microcontroller Software Interface Standard

Like desktop computing the software complexity of embedded applications is increasing exponentially. Now more than ever developers are using third-party code to meet project deadlines. ARM has defined the Cortex Microcontroller Software Interface Standard (CMSIS), which allows easy integration of source code from multiple sources. CMSIS is gaining increasing support through the industry and should be adopted for new projects.

Introduction

The widespread adoption of the Cortex-M processor into general purpose microcontrollers has led to two rising trends within the electronics industry. First of all the same processor is available from a wide range of vendors each with their own family of microcontrollers. In most cases, each vendor creates a range of microcontrollers that span a range of requirements for embedded systems developers. This proliferation of devices means that as a developer you can select a suitable microcontroller from many hundreds of devices while still using the same tools and skills regardless of the silicon vendor. This explosive growth in Cortex-M-based microcontrollers has made the Cortex-M processor the de facto industry standard for 32-bit microcontrollers and there are currently no real challengers.

Figure 4.1
CMSIS compliant software development tools and middleware stacks are allowed to carry the CMSIS logo.

The flip side of the coin is differentiation. It would be possible for a microcontroller vendor to design their own proprietary 32-bit processor. However, this is expensive to do and also requires an ecosystem of affordable tools and software to achieve mass adoption. It is more cost effective to license the Cortex-M processor from ARM and then use their own expertise to create a microcontroller with innovative peripherals. There are now more than 10 silicon vendors shipping Cortex-M-based microcontrollers. While in each device the Cortex-M processor is the same, each silicon manufacturer seeks to offer a unique set of user peripherals for a given range of applications. This can be a microcontroller designed for low-power applications, motor control, communications, or graphics. This way a silicon vendor can offer a microcontroller with a state-of-the-art processor that has wide development tools support while at the same time using their skill and knowledge to develop a microcontroller featuring an innovative set of peripherals.

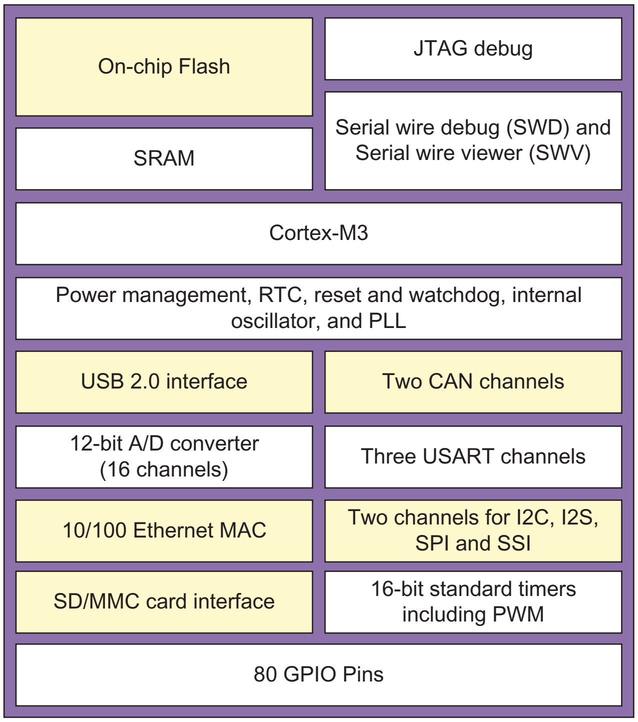

On-chip Flash
JTAG debug
SRAM
Serial wire debug (SWD) and Serial wire viewer (SWV)
Cortex-M3
Power management, RTC, reset and watchdog, internal oscillator, and PLL
USB 2.0 interface
Two CAN channels
12-bit A/D converter (16 channels)
Three USART channels
10/100 Ethernet MAC
Two channels for I2C, I2S, SPI and SSI
SD/MMC card interface
16-bit standard timers including PWM
80 GPIO Pins

Figure 4.2
Cortex-based microcontrollers can have a number of complex peripherals on a single chip. To make these work you will need to use some form of third-party code. CMSIS is intended to allow stacks from different sources to integrate together easily.

These twin factors have led to a vast "cloud" of standard microcontrollers with increasingly complex peripherals as well as typical microcontroller peripherals such as USART, I2C, ADC, and DAC. A modern high-end microcontroller could well have a Host\Device USB controller, Ethernet MAC, SDIO controller, and LCD interface. The software to drive any of these peripherals is effectively a project in itself, so gone are the days of a developer using an 8\16-bit microcontroller and writing all of the application code from the reset vector. To release any kind of sophisticated product it is almost certain that you will be using some form of third-party code in order to meet project deadlines. The third-party code may take the form of example code, an open source or commercial stack or a library provided by the silicon vendor. Both of these trends have created a need to make C-level code more portable between different development tools and different microcontrollers. There is also a need to be able to easily integrate code taken from a variety of sources into a project.

In order to address these issues, a consortium of silicon vendors and tools vendors has developed the CMSIS (seeMsys) for short.

CMSIS Specifications

The main aim of CMSIS is to improve software portability and reusability across different microcontrollers and toolchains. This allows software from different sources to integrate seamlessly together. Once learned, CMSIS helps to speed up software development through the use of standardized software functions.

At this point it is worth being clear about exactly what CMSIS is. CMSIS consists of five interlocking specifications that support code development across all Cortex-M-based microcontrollers. The four specifications are as follows: CMSIS core, CMSIS RTOS, CMSIS DSP, CMSIS SVD, and CMSIS DAP.

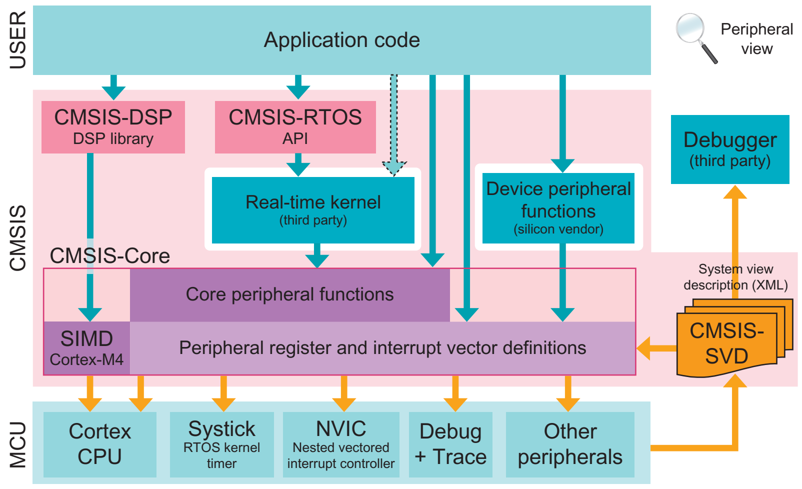
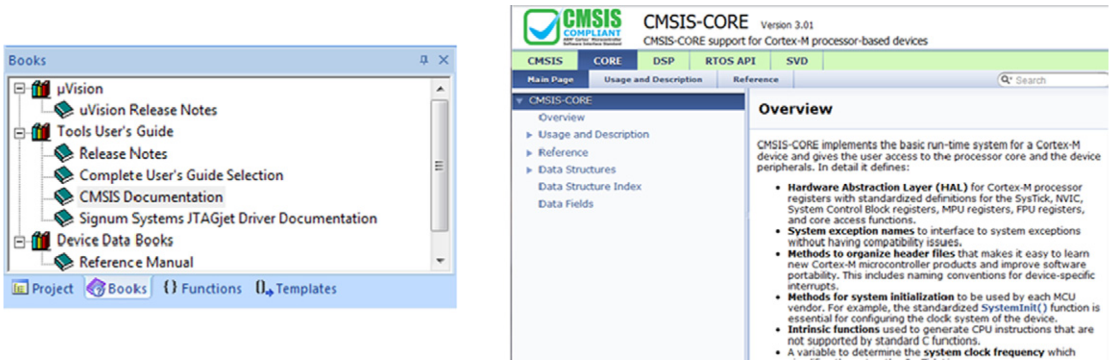


Figure 4.3
CMSIS consists of a several separate specifications (CORE, DSP, RTOS, SVD, and DAP) that make source code more portable between tools and devices.

It is also worth being clear what CMSIS is not. CMSIS is not a complex abstraction layer that forces you to use a complex and bulky library. CMSIS does not attempt to "dumb down" peripherals by providing standard profiles that make different manufacturers' peripherals work the same way. Rather, the CMSIS core specification takes a very small amount of resources (about 1 k of code and just 4 bytes of RAM) and just standardizes the way you access the Cortex-M processor and microcontroller registers. Furthermore, CMSIS does not really affect the way you develop code or force you to adopt a particular methodology. It simply provides a framework that helps you to integrate third-party code and reuse the code on future projects. Each of the CMSIS specifications are not that complicated and can be learned easily.

The full CMSIS specifications can be downloaded from the URL www.onarm.com. Each of the CMSIS specifications are integrated into the MDK-ARM toolchain and the CMSIS documentation is available from the online help.



Overview

CMSIS-CORE implements the basic run-time system for a Cortex-M device and gives the user access to the processor core and the device peripherals. In detail it defines:

- **Hardware Abstraction Layer (HAL)** for Cortex-M processor registers with standardized definitions for the SysTick, NVIC, System Control Block registers, MPU registers, FPU registers, and core access functions.
- **System exception names** to interface to system exceptions without having compatibility issues.
- **Methods to organize header files** that makes it easy to learn new Cortex-M microcontroller products and improve software portability. This includes naming conventions for device-specific interrupts.
- **Methods for system initialization** to be used by each MCU vendor. For example, the standardized SystemInit() function is essential for configuring the clock system of the device.
- **Intrinsic functions** used to generate CPU instructions that are not supported by standard C functions.
- A variable to determine the **system clock frequency** which

Figure 4.4
The CMSIS documentation can be found in the µVision books tab.

CMSIS Core

The core specification provides a minimal set of functions and macros to access the key Cortex-M processor registers. The core specification also defines a function to configure the microcontroller oscillators and clock tree in the startup code so the device is ready for use when you reach main(). The core specification also standardizes the naming of the device peripheral registers. The CMSIS core specification also standardizes support for the instrumentation trace during debug sessions.

CMSIS RTOS

The CMSIS RTOS specification provides a standard API for an RTOS. This is in effect a set of wrapper functions that translate the CMSIS RTOS API to the API of the specific RTOS that you are using. We will look at the use of an RTOS in general and the CMSIS RTOS API in Chapter 6. The Keil RTX RTOS was the first RTOS to support the CMSIS RTOS API and it has been released as an open source reference implementation. RTX can be compiled with both the GCC and IAR compilers. It is licensed with a three-clause Berkeley Software Distribution (BSD) license that allows its unrestricted use in commercial and noncommercial applications.

CMSIS DSP

As we have seen in Chapter 3, the Cortex-M4 is a "digital signal controller" with a number of enhancements to support DSP algorithms. Developing a real-time DSP system is best described as a "nontrivial pastime" and can be quite daunting for all but the simplest systems. To help mere mortals include DSP algorithms in Cortex-M4 and Cortex-M3 projects. CMSIS includes a DSP library that provides over 60 of the most commonly used DSP mathematical functions. These functions are optimized to run on the Cortex-M4 but can also be compiled to run on the Cortex-M3. We will take a look at using this library in Chapter 7.

CMSIS SVD and DAP

One of the key problems for tools vendors is to provide debug support for new devices as soon as they are released. The debugger must provide peripheral view windows that show the developer the current state of the microcontroller peripheral registers. With the growth in both the numbers of Cortex-M vendors and also the rising number and complexity of on-chip peripherals it is becoming all but impossible for any given tools vendor to maintain support for all possible microcontrollers. To overcome this hurdle, the CMSIS debug specification defines a "system viewer description"(SVD) file. This file is provided and maintained by the silicon vendor and contains a complete description of the microcontroller peripheral registers in an XML format. This file is then imported by the development tool, which uses it to automatically construct the peripheral debug windows for the microcontroller. This approach allows full debugger support to be available as new microcontrollers are released.

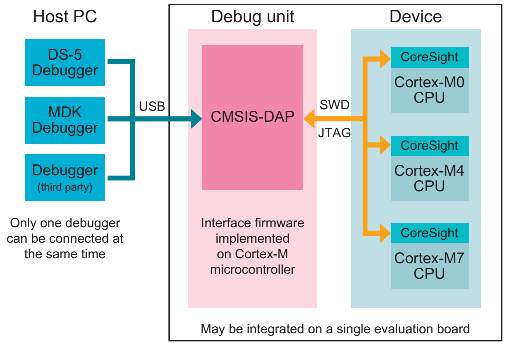

Host PC
DS-5 Debugger
MDK Debugger
Debugger (third party)
Only one debugger can be connected at the same time
USB
Debug unit
CMSIS-DAP
Interface firmware implemented on Cortex-M microcontroller
SWD
JTAG
Device
CoreSight
Cortex-M0 CPU
CoreSight
Cortex-M4 CPU
CoreSight
Cortex-M7 CPU
May be integrated on a single evaluation board

Figure 4.5
CMSIS DAP allows for interoperability between different vendors, software, and hardware debuggers.

The CMSIS DAP specification defines the interface protocol for a hardware debug unit that sits between the host PC and the debug access port (DAP) of the microcontroller. This allows any software toolchain that supports CMSIS DAP to connect to any hardware debug unit that also supports CMSIS DAP. There are an increasing number of very-low-cost evaluation boards that contain an integral debugger. Often this debugger supports a selected toolchain; with CMSIS DAP such a board could be used with any compliant development tool.

Foundations of CMSIS

The CMSIS core specification provides a standard set of low-level functions, macros, and peripheral register definitions that allow your application code to easily access the Cortex-M processor and microcontroller peripheral registers. This framework needs to be added to your code at the start of a project. This is actually very easy to do as the CMSIS core functions are very much a part of the compiler toolchain.

Coding Rules

While CMSIS is important for providing a standardized software interface for all Cortex-M microcontrollers it is also interesting for embedded developers because it defines a consistent set of C coding rules. When applied, these coding rules generate clear unambiguous C code. This approach is worth studying as it embodies many of the best practices that should be adopted when writing the C source code for your own application software.

MISRA C

The main backbone of the CMSIS coding rules is a set of coding guidelines called MISRA C published by MIRA. MIRA stands for "Motor Industry Research Agency" and is located near Rugby in England. It is responsible for many of the industry standards used by the UK motor industry. In 1998, its software division the Motor Industry Software Research Agency released the first version of its coding rules formally called "MISRA guidelines for the use of C in vehicle electronics."

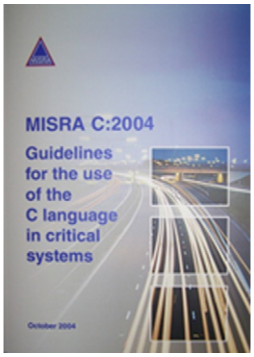

MISRA C:2004
Guidelines for the use of the C language in critical systems
October 2004

Figure 4.6
The CMSIS source code has been developed using MISRA C as a coding standard.

The original MISRA C specification contained 127 rules which attempted to prevent common coding mistakes and resolve gray areas of the ANSI C specification when applied to embedded systems. Although originally intended for the automotive industry MISRA C has found acceptance in the wider embedded systems community. In 2004, a revised edition of MISRA C was released with the title "MISRA C coding guidelines for the use of C in safety systems." This change in the title reflects the growing adoption of MISRA C as a coding standard for general embedded systems. One of the other key attractions of MISRA C is that it was written by engineers and not by computer scientists. This has resulted in a clear, compact, and easy to understand set of rules. Each rule is clearly explained with examples of good coding practice. This means that the entire coding standard is contained in a book of just 106 pages which can easily be read in an evening. A typical example of a MISRA C rule is shown below.

Rule 13.6(required) Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.

Loop counters shall not be modified in the body of the loop. However, other loop control variables representing logical values may be modified in the loop. For example, a flag to indicate that something has been completed which is then tested in the for statement.

```
Flag = 1;
For ((I = 0;(i < 5) && (flag == 1);i++)
{
/*.......*/
Flag = 0;  /* Compliant — allows early termination of the loop */
i = i + 3;    /*Not Compliant — altering the loop counter */
}
```

Where possible, the MISRA C rules have been designed so that they can be statically checked either manually or by a dedicated tool. The MISRA C standard is not an open standard and is published in paper and electronic form on the MIRA Web site. Full details of how to obtain the MISRA standard are available in Appendix A.

In addition to the MISRA C guidelines, CMSIS enforces some additional coding rules. To prevent any ambiguity in the compiler implementation of standard C types CMSIS uses the data types defined in the ANSI C header file stdint.h.

The typedefs ensure that the expected data size is mapped to the correct ANSI type for a given compiler. Using typedefs like this is a good practice, as it avoids any ambiguity about the underlying variable size, which may vary between compilers particularly if you are migrating code between different processor architectures and compiler tools.

CMSIS also specifies IO type qualifiers for accessing peripheral variables. These are typedefs that make clear the type of access each peripheral register has.

Table 4.1: CMSIS Variable Types

Standard ANSI C Type	MISRA C Type
Signed char	int8_t
Signed short	int16_t
Signed int	int32_t
Signed __int64	int64_t
Unsigned char	uint8_t
Unsigned short	uint16_t
Unsigned int	uint32_t
Unsigned __int64	uint64_t

While this does not provide any extra functionality for your code it provides a common mechanism that can be used by static checking tools to ensure that the correct access is made to each peripheral register.

Much of the CMSIS documentation is autogenerated using a tool called Doxegen. This is a free download released under a GNU Public (GPL) license. While Doxegen cannot

Table 4.2: CMSIS IO Qualifiers

MISRA C IO Qualifier	ANSI C Type	Description
#define __I	Volatile const	Read only
#define __O	Volatile	Write only
#define __IO	Volatile	Read and write

actually write the documentation for you it does do much of the dull boring stuff for you (leaving you to do the exiting documentation work). Doxegen works by analyzing your source code and extracting declarations and specific source code comments to build up a comprehensive "object dictionary" for your project. The default output format for Doxegen is a browsable HTML but this can be converted to other formats if desired. The CMSIS source code comments contain specific tags prefixed by the @ symbol, for example, @brief. These tags are used by Doxegen to annotate descriptions of the CMSIS functions.

```
/**
* @brief Enable Interrupt in NVIC Interrupt Controller
* @param IRQn interrupt number that specifies the interrupt
* @return none.
* Enable the specified interrupt in the NVIC Interrupt Controller.
* Other settings of the interrupt such as priority are not affected.
*/
```

When the Doxegen tool is run it analyzes your source code and generates a report containing a dictionary of your functions and variables based on the comments and source code declarations.

CMSIS Core Structure

The CMSIS core functions can be included in your project through the addition of three files. These include the default startup code with the CMSIS standard vector table. The second file is the system_<device>.c file, which contains the necessary code to initialize the microcontroller system peripherals. The final file is the device include file, which imports the CMSIS header files that contain the CMSIS core functions and macros.

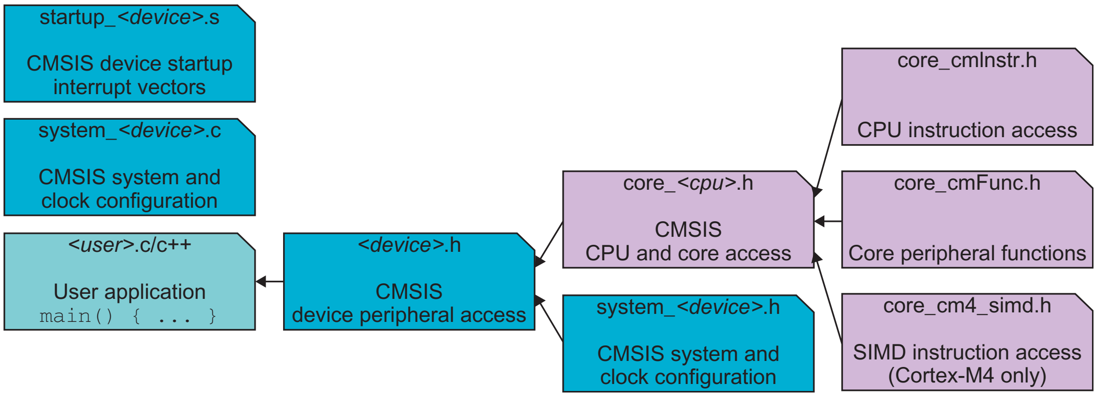

startup_<device>.s
CMSIS device startup interrupt vectors
system_<device>.c
CMSIS system and clock configuration
<user>.c/c++
User application
main() { ... }
<device>.h
CMSIS device peripheral access
core_<cpu>.h
CMSIS CPU and core access
system_<device>.h
CMSIS system and clock configuration
core_cmInstr.h
CPU instruction access
core_cmFunc.h
Core peripheral functions
core_cm4_simd.h
SIMD instruction access (Cortex-M4 only)

Figure 4.7
The CMSIS core standard consists of the device startup, system C code, and a device header. The device header defines the device peripheral registers and pulls in the CMSIS header files. The CMSIS header files contain all of the CMSIS core functions.

Startup Code

The startup code provides the reset vector, initial stack pointer value, and a symbol for each of the interrupt vectors.

```
__Vectors  DCD  __initial_sp      ; Top of Stack
  DCD Reset_Handler               ; Reset Handler
  DCD NMI_Handler                 ; NMI Handler
  DCD HardFault_Handler           ; Hard Fault Handler
  DCD MemManage_Handler           ; MPU Fault Handler
```

When the processor starts, it will initialize the MSP by loading the value stored in the first 4 bytes of the vector table. Then it will jump to the reset handler.

```
Reset_Handler  PROC
  EXPORT  Reset_Handler    [WEAK]
```

```
    IMPORT __main
    IMPORT SystemInit
     LDR  R0, =SystemInit
     BLX  R0
     LDR  R0, =__main
     BX  R0
     ENDP
```

System Code

The reset handler calls the SystemInit() function, which is located in the CMSIS system_<device>.c file. This code is delivered by the silicon manufacturer and it provides all the necessary code to configure the microcontroller out of reset. Typically this includes setting up the internal phase-locked loops, configuring the microcontroller clock tree and internal bus structure, enabling the external bus if required, and switching on any peripherals held in low-power mode. The configuration of the initializing functions is controlled by a set of #defines located at the start of the module. This allows you to customize the basic configuration of the microcontroller system peripherals. Since the SystemInit() function is run when the microcontroller leaves reset the microcontroller and the Cortex-M processor will be in a running state when the program reaches main. In the past, this initializing code was something you would have had to write for yourself or crib from example code. The SystemInit() function does save you a lot of time and effort. The SystemInit() function also sets the CMSIS global variable SystemCoreClock to the CPU frequency. This variable can then be used by the application code as a reference value when configuring the microcontroller peripherals. In addition to the SystemInit() function the CMSIS system file contains an additional function to update the SystemCoreClock variable if the CPU clock frequency is changed on the fly. The function SystemCoreClockUpdate() is a void function that must be called if the CPU clock frequency is changed. This function is tailored to each microcontroller and will evaluate the clock tree registers to calculate the new CPU operating frequency and change the SystemCoreClock variable accordingly.

Once the SystemInit() function has run and we reach the application code we will need to access the CMSIS core functions. This framework is added to the application modules through the microcontroller-specific header file.

Device Header File

The header file first defines all of the microcontroller special function registers in a CMSIS standard format.

A typedef structure is defined for each group of special function registers on the supported microcontroller. In the code below, a general GPIO typedef is declared for the group of

GPIO reregisters. This is a standard typedef but we are using the IO qualifiers to designate the type of access granted to a given register.

```
typedef struct
{
__IO uint32_t MODER;   /*!< GPIO port mode register,        Address offset: 0x00 */
__IO uint32_t OTYPER;   /*!< GPIO port output type register,      Address offset: 0x04 */
__IO uint32_t OSPEEDR;   /*!< GPIO port output speed register,      Address offset: 0x08 */
__IO uint32_t PUPDR;   /*!< GPIO port pull-up/pull-down register,   Address offset: 0x0C */
__IO uint32_t IDR;   /*!< GPIO port input data register,      Address offset: 0x10 */
__IO uint32_t ODR;   /*!< GPIO port output data register,      Address offset: 0x14 */
__IO uint16_t BSRRL;   /*!< GPIO port bit set/reset low register,   Address offset: 0x18 */
__IO uint16_t BSRRH;   /*!< GPIO port bit set/reset high register,   Address offset: 0x1A */
__IO uint32_t LCKR;   /*!< GPIO port configuration lock register,   Address offset: 0x1C */
__IO uint32_t AFR[2];   /*!< GPIO alternate function registers,   Address offset:
                                                                0x24-0x28 */
} GPIO_TypeDef;
```

Next #defines are used to lay out the microcontroller memory map. First, the base address of the peripheral special function registers is declared and then offset addresses to each of the peripheral busses and finally an offset to the base address of each GPIO port.

```
#define PERIPH_BASE    ((uint32_t)0x40000000)
#define APB1PERIPH_BASE     PERIPH_BASE
#define GPIOA_BASE  (AHB1PERIPH_BASE + 0x0000)
#define GPIOB_BASE  (AHB1PERIPH_BASE + 0x0400)
#define GPIOC_BASE  (AHB1PERIPH_BASE + 0x0800)
#define GPIOD_BASE  (AHB1PERIPH_BASE + 0x0C00)
```

Then the register symbols for each GPIO port can be declared.

```
#define GPIOA   ((GPIO_TypeDef *)GPIOA_BASE)
#define GPIOB  ((GPIO_TypeDef *) GPIOB_BASE)
#define GPIOC  ((GPIO_TypeDef *) GPIOC_BASE)
#define GPIOD  ((GPIO_TypeDef *) GPIOD_BASE)
```

Then in the application code we can program the peripheral special function registers by accessing the structure elements.

```
void LED_Init (void) {
RCC->AHB1ENR |= ((1UL << 3) );   /* Enable GPIOD clock    */
GPIOD->MODER &= ~((3UL << 2*12) |
    (3UL << 2*13) |
    (3UL << 2*14) |
        (3UL << 2*15) ); /* PD.12..15 is output    */
    GPIOD->MODER |= ((1UL << 2*12)|
        (1UL << 2*13)|
        (1UL << 2*14)|
        (1UL << 2*15) );
```

The microcontroller include file provides similar definitions for all of the on-chip peripheral special function registers. These definitions are created and maintained by the silicon manufacturer and as they do not use any non-ANSI keywords the include file may be used with any C compiler. This means that any peripheral driver code written to the CMSIS specification is fully portable between CMSIS compliant tools.

The microcontroller include file also provides definitions of the interrupt channel number for each peripheral interrupt source.

```
    WWDG_IRQn         = 0,  /*!< Window WatchDog interrupt    */
    PVD_IRQn          = 1,  /*!< PVD through EXTI line detection interrupt    */
    TAMP_STAMP_IRQn   = 2,  /*!< Tamper and TimeStamp interrupts through the EXTI line    */
    RTC_WKUP_IRQn     = 3,  /*!< RTC Wakeup interrupt through the EXTI line    */
    FLASH_IRQn        = 4,  /*!< FLASH global interrupt    */
    RCC_IRQn          = 5,  /*!< RCC global interrupt    */
    EXTI0_IRQn        = 6,  /*!< EXTI Line0 interrupt    */
    EXTI1_IRQn        = 7,  /*!< EXTI Line1 interrupt    */
    EXTI2_IRQn        = 8,  /*!< EXTI Line2 interrupt    */
    EXTI3_IRQn        = 9,  /*!< EXTI Line3 interrupt    */
    EXTI4_IRQn        = 10,  /*!< EXTI Line4 interrupt
```

In addition to the register and interrupt definitions, the silicon manufacturer may also provide a library of peripheral driver functions. Again as this code is written to the CMSIS standard it will compile with any suitable development tool. Often these libraries are very useful for getting a project to work quickly and minimizing the amount of time you have to spend writing low-level code. However, they are often very general libraries that do not yield the most optimized code. So, if you need to get the maximum performance/minimal code size, you will need to rewrite the driver functions to suit your specific application. The microcontroller include file also imports up to five further include files. These include stdint.h, a CMSIS core file for the Cortex processor you are using. A header file System_<device>.h is also included to give access to the functions in the system file. The CMSIS instruction intrinsic and helper functions are contained in two further files, core_cminstr.h and core_cmfunc.h. If you are using the Cortex-M4 an additional file, core_CM4_simd.h, is added to provide support for the Cortex-M4 SIMD instructions.

As discussed earlier, the stdint.h file provides the MISRA C types that are used in the CMSIS definitions and should be used through your application code.

CMSIS Core Header Files

The device file imports the Cortex-M processor which are held in their own include files. There are a small number of defines that are set up for a given device. These can be found in the main processor include file.

The processor include file also imports the CMSIS header files, which contain the CMSIS core helper functions. The helper functions are split into the groups shown below.

The NVIC group provides all the functions necessary to configure the Cortex-M interrupts and exceptions. A similar function is provided to configure the systick timer and interrupt. The CPU register group allows you to easily read and write to the CPU registers using the MRS and MSR instructions. Any instructions that are not reachable by the C language are provided by dedicated intrinsic functions and are contained in the CPU instructions group. An extended set of intrinsics are also provided for the Cortex-M4 to access the SIMD instructions. Finally, some standard functions are provided to access the debug instrumentation trace.

Table 4.3: CMSIS Configuration Values

__CMx_REV	Core revision number
__NVIC_PRIO_BITS	Number of priority bits implemented in the NVIC priority registers
__MPU_PRESENT	Defines if an MPU is present (see Chapter 5)
__FPU_PRESENT	Defines if an FPU is present (see Chapter 7)
__Vendor_SysTickConfig	Defines if there is a vendor-specific systick configuration

Interrupts and Exceptions

Management of the NVIC registers may be done by the functions provided in the interrupt and exception group. These functions allow you to setup an NVIC interrupt channel and manage its priority as well as interrogate the NVIC registers during runtime.

A configuration function is also provided for the systick timer.

So, for example, to configure an external interrupt line we first need to find the name for the external interrupt vector used in the startup code of the vector table.

Table 4.4: CMSIS Function Groups

CMSIS Core Function Groups
NVIC access functions
Systick configuration
CPU register access
CPU instruction intrinsics
Cortex-M4 SIMD intrinsics
ITM debug functions

Table 4.5: CMSIS Interrupt and Exception Group

CMSIS Function	Description
NVIC_SetPriorityGrouping	Set the priority grouping (Not for Cortex-M0)
NVIC_GetPriorityGrouping	Read the priority grouping (Not for Cortex M0)
NVIC_EnableIRQ	Enable a peripheral interrupt channel
NVIC_DisableIRQ	Disable a peripheral interrupt channel
NVIC_GetPendingIRQ	Read the pending status of an interrupt channel
NVIC_SetPendingIRQ	Set the pending status of an interrupt channel
NVIC_ClearPendingIRQ	Clear the pending status of an interrupt channel
NVIC_GetActive	Get the active status of an interrupt channel (Not for Cortex-M0)
NVIC_SetPriority	Set the active status of an interrupt channel
NVIC_GetPriority	Get the priority of an interrupt channel
NVIC_EncodePriority	Encode the priority group (Not for Cortex-M0)
NVIC_DecodePriority	Decode the priority group (Not for Cortex-M0)
NVIC_SystemReset	Force a system reset

Table 4.6: CMSIS Systick Function

CMSIS Function	Description
SysTick_Config	Configures the timer and enables the interrupt

```
  DCD FLASH_IRQHandler          ; FLASH
DCD  RCC_IRQHandler             ; RCC
DCD  EXTI0_IRQHandler           ; EXTI Line0
DCD  EXTI1_IRQHandler           ; EXTI Line1
DCD  EXTI2_IRQHandler           ; EXTI Line2
DCD  EXTI3_IRQHandler           ; EXTI Line3
DCD  EXTI4_IRQHandler           ; EXTI Line4
DCD  DMA1_Stream0_IRQHandler    ; DMA1 Stream 0
```

So for external interrupt line 0 we simply need to create a void function duplicating the name used in the vector table.

```
void EXTI0_IRQHandler (void);
```

This now becomes our interrupt service routine. In addition, we must configure the microcontroller peripheral and NVIC to enable the interrupt channel. In the case of an external interrupt line, the following code will setup Port A pin 0 to generate an interrupt to the NVIC on a falling edge.

```
AFIO->EXTICR[0]  &= ~AFIO_EXTICR1_EXTI0;    /* clear used pin  */
AFIO->EXTICR[0]  |= AFIO_EXTICR1_EXTI0_PA;    /* set PA.0 to use  */
EXTI->IMR        |= EXTI_IMR_MR0;    /* unmask interrupt  */
EXTI->EMR        &= ~EXTI_EMR_MR0;    /* no event    */
EXTI->RTSR       &= ~EXTI_RTSR_TR0;    /* no rising edge trigger  */
EXTI->FTSR       |= EXTI_FTSR_TR0;    /* set falling edge trigger  */
```

Next we can use the CMSIS functions to enable the interrupt channel.

```
NVIC_EnableIRQ(EXTI0_IRQn);
```

Here we are using the defined enumerated type for the interrupt channel number. This is declared in the microcontroller header file. Once you get a bit familiar with the CMSIS core functions, it becomes easy to intuitively work out the name rather than having to look it up or look up the NVIC channel number.

We can also add a second interrupt source by using the systick configuration function, which is the only function in the systick group.

```
uint32_t SysTick_Config(uint32_t ticks)
```

This function configures the countdown value of the systick timer and enables its interrupt so an exception will be raised when its count reaches zero. Since the SystemInit() function sets the global variable SystemCoreClock with the CPU frequency that is also used by the systick timer we can easily setup the systick timer to generate a desired periodic interrupt. So a 1 ms interrupt can be generated as follows:

```
Systick_Config(SystemCoreClock/1000);
```

Again we can look up the exception handler from the vector table

```
  DCD  0                ; Reserved
DCD  PendSV_Handler    ; PendSV Handler
DCD  SysTick_Handler   ; SysTick Handler
```

and create a matching C function:

```
void SysTick_Handler (void);
```

Now that we have two interrupt sources we can use other CMSIS interrupt and exception functions to manage the priority levels. The number of priority levels will depend on how many priority bits have been implemented by the silicon manufacturer. For all of the Cortex-M processors we can use a simple “flat” priority scheme where zero is the highest priority. The priority level is set by

```
NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority);
```

The set priority function is a bit more intelligent than a simple macro. It uses the IRQn NVIC channel number to differentiate between user peripherals and the Cortex processor exceptions. This allows it to program either the system handler priority registers in the SCB or the interrupt priority registers in the NVIC itself. The set priority function also uses the NVIC_PRIO_BITS definition to shift the priority value into the active priority bits that have been implemented by the device manufacturer.

```
_STATIC_INLINE void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)
{
  if(IRQn < 0) {
    SCB->SHP[((uint32_t)(IRQn) & 0xF)-4] = ((priority << (8 - __NVIC_PRIO_BITS)) &
0xff); } /* set Priority for Cortex M System Interrupts */
  else {
    NVIC->IP[(uint32_t)(IRQn)] = ((priority << (8 - __NVIC_PRIO_BITS)) & 0xff);  }
                                             /* set Priority for device specific Interrupts */
}
```

However, for the Cortex-M3 and Cortex-M4 we have the option to set priority groups and subgroups as discussed in Chapter 3. Depending on the number of priority bits defined by the manufacturer, we can configure priority groups and subgroups.

```
NVIC_SetPriorityGrouping();
```

To set the NVIC priority grouping you must write to the AIRC register. As discussed in Chapter 3, this register is protected by its VECTKEY field. In order to update this register, you must write 0x5FA to the VECTKEY field. The SetPriorityGrouping function provides all the necessary code to do this.

```
__STATIC_INLINE void NVIC_SetPriorityGrouping(uint32_t PriorityGroup)
{
uint32_t reg_value;
uint32_t PriorityGroupTmp = (PriorityGroup & (uint32_t)0x07);     /* only values 0..7 are
                                                                    used    */
reg_value = SCB->AIRCR;      /* read old register configuration  */
reg_value &=  ~(SCB_AIRCR_VECTKEY_Msk | SCB_AIRCR_PRIGROUP_Msk);  /* clear bits to
                                                                        change     */
reg_value = (reg_value | ((uint32_t)0x5FA << SCB_AIRCR_VECTKEY_Pos) |  /* insert write
                                                                   key and priorty group */
  (PriorityGroupTmp << 8));
SCB->AIRCR = reg_value;
}
```

The interrupt and exception group also provides a system reset function that will generate a hard reset of the whole microcontroller.

```
NVIC_SystemReset(void);
```

This function writes to bit 2 of the “Application Interrupt Reset Control” register. This strobes a logic line out of the Cortex-M core to the microcontroller reset circuitry which resets the microcontroller peripherals and the Cortex-M core. However, you should be a little careful here as the implementation of this feature is down to the microcontroller manufacturer and may not be fully implemented. So if you are going to use this feature you need to test it first. Bit zero of the same register will do a warm reset of the Cortex-M core, that is, force a reset of the Cortex-M processor but leave the microcontroller registers configured. This feature is normally used by debug tools.

Exercise: CMSIS and User Code Comparison

In this exercise, we will revisit the multiple interrupts example and examine a rewrite of the code using the CMSIS core functions.

Open the project in c:\exercises\CMSIS core multiple interrupt and c:\exercises\multiple interrupt.

Open main.c in both projects and compare the initializing code.

The systick timer and ADC interrupts can be initialized with the following CMSIS functions.

```
SysTick_Config(SystemCoreClock / 100);
NVIC_EnableIRQ  (ADC1_2_IRQn);
NVIC_SetPriorityGrouping  (5);
NVIC_SetPriority  (SysTick_IRQn,4);
NVIC_SetPriority  (ADC1_2_IRQn,4);
```

Or you can use the equivalent non-CMSIS code.

```
SysTick->VAL = 0x9000;              //Start value for the sysTick counter
SysTick->LOAD = 0x9000;             //Reload value
SysTick->CTRL = SYSTICK_INTERRUPT_ENABLE
```

```
      |SYSTICK_COUNT_ENABLE;          //Start and enable interrupt
    NVIC->ISER[0] = (1UL << 18);      /* enable ADC Interrupt  */
    NVIC->IP[18] = (2<<6 | 2<<4);
    SCB->SHP[11] = (1<<6 | 3<<4);
    Temp = SCB->AIRC;
    Temp &= ~0x
    Temp = Temp|(0xAF0<< )|(0x05<< );
```

Although both blocks of code achieve the same thing, the CMSIS version is much more readable and far less prone to coding mistakes.

Build both projects and compare the size of the code produced.

The CMSIS functions introduce a small overhead but this is an acceptable trade-off against ease of use and maintainability.

CMSIS Core Register Access

The next group of CMSIS functions gives you direct access to the processor core registers.

These functions provide you with the ability to globally control the NVIC interrupts and set the configuration of the Cortex-M processor into its more advanced operating mode. First, we can globally enable and disable the microcontroller interrupts with the following functions.

```
    __set_PRIMASK(void);
    __set_FAULTMASK (void);
    __enable-IRQ
    __enable_Fault-irq
    __set_BASEPRI()
```

Table 4.7: CMSIS CPU Register Functions

Core Function	Description
__get_Control	Read the control register
__set_Control	Write to the control register
__get_IPSR	Read the IPSR register
__get_APSR	Read the APSR register
__get_xPSR	Read the xPSR register
__get_PSP	Read the process stack pointer
__set_PSP	Write to the process stack pointer
__get_MSP	Read the main stack pointer
__set_MSP	Write to the main stack pointer
__get_PRIMASK	Read the PRIMASK

(*Continued*)

Table 4.7: (Continued)

Core Function	Description
__set_PRIMASK	Write to the PRIMASK
__get_BASEPRI	Read the BASEPRI register
__set_BASEPRI	Write to the BASEPRI register
__get_FAULTMASK	Read the FAULTMASK
__set_FAULTMASK	Write to the FAULTMASK
__get_FPSCR	Read the FPSCR
__set_FPSCR	Write to the FPSCR
__enable_irq	Enable interrupts and configurable fault exceptions
__disable_irq	Disable interrupts and configurable fault exceptions
__enable_fault_irq	Enable interrupts and all fault handlers
__disable_fault_irq	Disable interrupts and all fault handlers

While all of these functions are enabling and disabling interrupt sources they all have slightly different effects. The __set_PRIMASK() function and the enable_IRQ\Disable_IRQ functions have the same effect in that they set and clear the PRIMASK bit, which enables and disables all interrupt sources except the hard fault handler and the nonmaskable interrupt. The __set_FAULTMASK() function can be used to disable all interrupts except the nonmaskable interrupt. We will see later how this can be useful when we want to bypass the MPU. Finally, the __set_BASEPRI() function sets the minimum active priority level for user peripheral interrupts. When the base priority register is set to a nonzero level the interrupt at the same priority level or lower will be disabled.

These functions allow you to read the PSR and its aliases. You can also access the control register to enable the advanced operating modes of the Cortex-M processor as well as explicitly setting the stack pointer values. A dedicated function is also provided to access the floating point status control (FPSC) register, if you are using the Cortex-M4. We will take a closer look at the more advanced operating modes of the Cortex-M processor in Chapter 5.

CMSIS Core CPU Intrinsic Instructions

The CMSIS core header also provides two groups of standardized intrinsic functions. The first group is common to all Cortex-M processors and the second group provides standard intrinsic for the Cortex-M4 SIMD instructions.

The CPU intrinsics provide direct access to Cortex-M processor instructions that are not directly reachable from the C language. While many of their functions can be achieved by using multiple instructions generated by high-level C code you can optimize your code by the judicious use of these instructions.

With the CPU intrinsics we can enter the low-power modes using the __WFI() and _WFE() instructions. The CPU intrinsics also provide access to the saturated math instructions that we met in Chapter 3. The intrinsic functions also give access to the execution of barrier instructions that ensure completion of a data write or instruction for execution before continuing with the next instruction. The next group of instruction intrinsics is used to guarantee exclusive access to a memory region by one region of code. We will take a look at these in Chapter 5. The remainder of the CPU intrinsics

Table 4.8: CMSIS Instruction Intrinsics

CMSIS Function	Description	More Information
__NOP	No operation	
__WFI	Wait for interrupt	See Chapter 3
__WFE	Wait for event	See Chapter 3
__SEV	Send event	
__ISB	Instruction synchronization barrier	See Chapter 3
__DSB	Data synchronization barrier	See Chapter 3
__DMD	Data memory synchronization barrier	See Chapter 3
__REV	Reverse byte order (32 bit)	See Chapter 4 for rotation instructions
__REV16	Reverse byte order (16 bit)	
__REVSH	Reverse byte order, signed short	
__RBIT	Reverse bit order (not for Cortex-M0)	
__ROR	Rotate right by n bits	
__LDREXB	Load exclusive (8 bits)	See Chapter 5 for exclusive access instructions
__LDREXH	Load exclusive (16 bits)	
__LDREXW	Load exclusive (32 bits)	
__STREXB	Store exclusive (8 bits)	
__STREXH	Store exclusive (16 bits)	
__STREXW	Store exclusive (32 bits)	
__CLREX	Remove exclusive lock	
__SSAT	Signed saturate	See Chapter 3
__USAT	Unsigned saturate	
__CLZ	Count leading zeros	See Chapter 4

support single-cycle data manipulation functions such as the rotate and RBIT instructions.

Exercise: Intrinsic Bit Manipulation

In this exercise we will look at the data manipulation intrinsic supported in CMSIS.

Open the exercise in c:\exercises\CMSIS core intrinsic.

The exercise declares an input variable and a group of output variables and then uses each of the intrinsic data manipulation functions.

```
outputREV    = __REV(input);
outputREV16  =__REV16(input);
outputREVSH  =__REVSH(input);
outputRBIT   =__RBIT(input);
outputROR    =__ROR(input,8);
outputCLZ    = __CLZ(input);
```

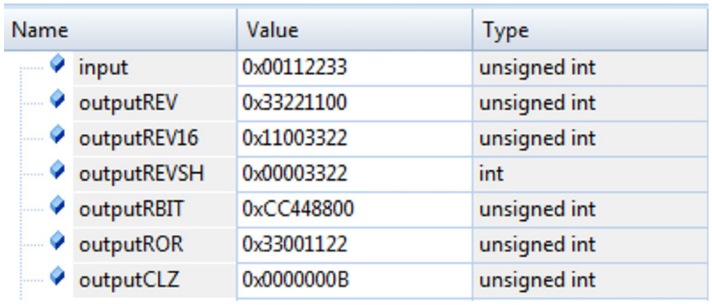
Build the project and start the debugger.

Add the input and each of the output variables to the watch window.

Step through the code and count the cycles taken for each function.

While each intrinsic instruction takes a single cycle some surrounding instructions are required so the intrinsic functions take between 9 and 18 cycles.

Examine the values in the output variables to familiarize yourself with the action of each intrinsic.



Name	Value	Type
input	0x00112233	unsigned int
outputREV	0x33221100	unsigned int
outputREV16	0x11003322	unsigned int
outputREVSH	0x00003322	int
outputRBIT	0xCC448800	unsigned int
outputROR	0x33001122	unsigned int
outputCLZ	0x0000000B	unsigned int

Consider how you would code each intrinsic in pure C.

CMSIS SIMD Intrinsics

The final group of CMSIS intrinsics provide direct access to the Cortex-M4 SIMD instructions.

The SIMD instructions provide simultaneous calculations for two 16-bit operations or four 8-bit operations. This greatly enhances any form of repetitive calculation over a data set, as in a digital filter. We will take a close look at these instructions in Chapter 7.

CMSIS Core Debug Functions

The final group of CMSIS core functions provides enhanced debug support through the CoreSight instrumentation trace. The CMSIS standard has two dedicated debug specifications, CMSIS SVD and CMSIS DAP, which we will look at in Chapter 8.

However, the CMSIS core specification contains some useful debug support. As part of their hardware debug system, the Cortex-M3 and Cortex-M4 provide an instrumentation trace unit (ITM). This can be thought of as a debug UART that is connected to a console window in the debugger. By adding debug hooks (instrumenting) into your code it is possible to read and write data to the debugger while the code is running. We will look at using the instrumentation trace for additional debug and software testing in Chapter 8. For now there are a couple of CMSIS functions that standardize communication with the ITM.

Exercise: Simple ITM

In this exercise we will look at the use of the instrumentation trace in a Cortex-M3 microcontroller. This is an exercise involving hardware debug so you will need the STM32 discovery board or other evaluation board.

Connect the discovery board to the PC via its USB debug port.

Open the exercise in c:\exercises\CMSIS core debug.

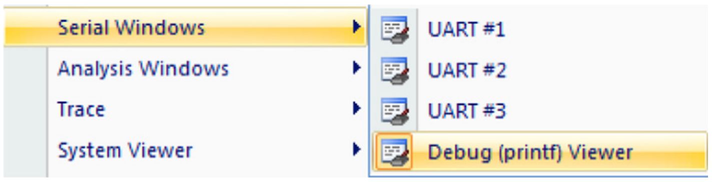
In this exercise the code configures the instrumentation trace and then reads and writes characters from the instrumentation port.

Build the project and start the debugger.

Table 4.9: CMSIS Debug Functions

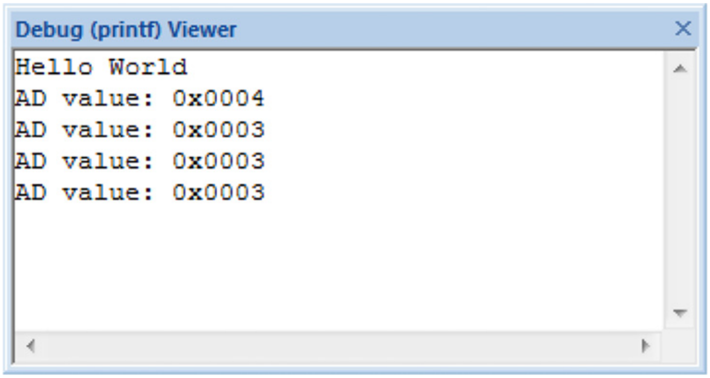
CMSIS Debug Function	Description
volatile int ITM_RxBuffer = ITM_RXBUFFER_EMPTY;	Declare one word of storage for receive flag
ITM_SendChar(c);	Send one character to the ITM
ITM_CheckChar()	Check if any data has been received
ITM_ReceiveChar()	Read one character from the ITM

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



Start running the code.

As the code runs it will send characters to the debug viewer.



```
Hello World
AD value: 0x0004
AD value: 0x0003
AD value: 0x0003
AD value: 0x0003
```

Highlight the debug viewer, enter menu options, and check they are received in the code.

The ITM allows you to use two-way communication with the target software without the need to use any of the microcontroller peripherals. The ITM can be very useful for teasing out more complex debug problems and can also be used for functional software testing. We will look at configuring the ITM and also how to use it for software testing in Chapter 8.