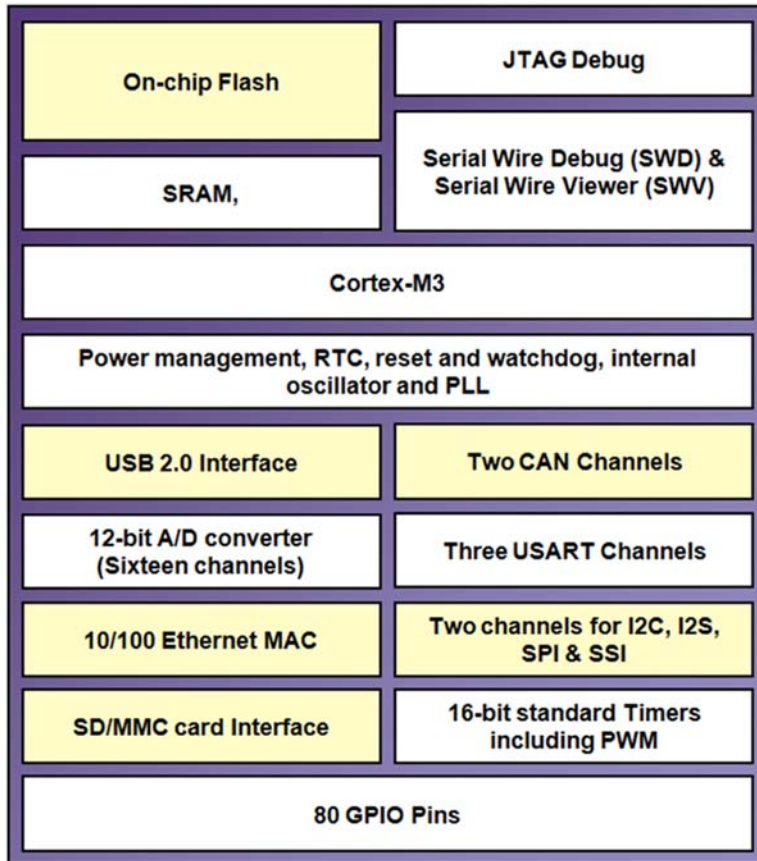


Common Microcontroller Software Interface Standard

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 family 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 several thousand 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.

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 enable 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 17 (up from 10 when I first wrote this book 3 years ago) Silicon Vendors shipping Cortex-M-based microcontrollers. While the Cortex-M processor is the same in all devices, each manufacturer of the final silicon seeks to offer a unique set of user peripherals (Fig. 4.1) 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 which has a wide ecosystem of development tools and software while at the same time using their skill and knowledge to develop a microcontroller featuring an innovative set of peripherals.

**Figure 4.1**

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, 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 single project.

**Figure 4.2**

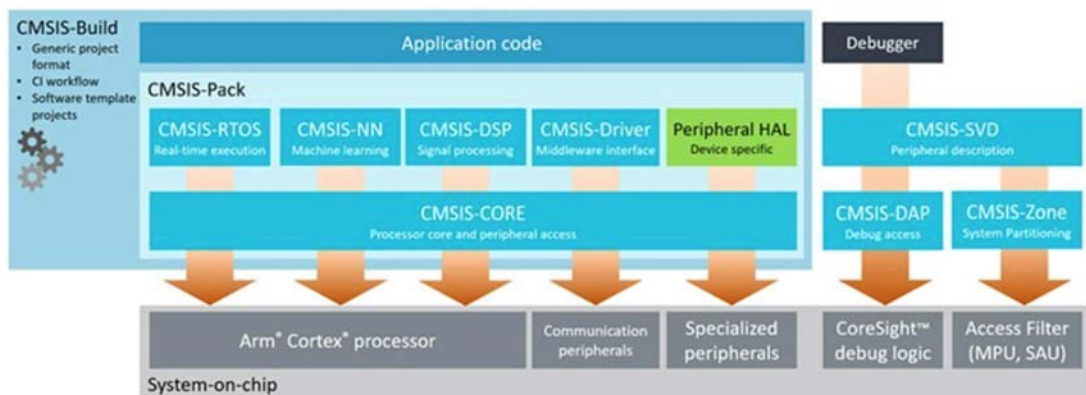
CMSIS-compliant software development tools and middleware stacks are allowed to carry the CMSIS logo.

In order to address these issues, a consortium of silicon vendors and tools vendors has developed a set of standards called CMSIS (seeMsys, [Fig. 4.2](#)). This originally stood for “Cortex Microcontroller Software Interface Standard” but more recently has been rebranded as “Common Microcontroller Software Interface Standard.”

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 learnt, 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 10 interlocking specifications that support code development across all Cortex-M-based microcontrollers. The 10 specifications are as follows: CMSIS-Core, CMSIS-RTOS, CMSIS-DSP, CMSIS-NN, CMSIS-Driver, CMSIS-Zone, CMSIS-Pack, CMSIS-SVD, CMSIS-DAP, and CMSIS-Build ([Fig. 4.3](#)). As we will see, some of these standards will directly affect how you write “C” code, while others work more “under the hood” to help align the development ecosystem to prevent work from being endlessly duplicated.

**Figure 4.3**

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

It is also worth being clear about what CMSIS is not. CMSIS is not a complex abstraction layer that forces you to use a complex and bulky library. Rather the CMSIS-Core specification takes a very small amount of resources about 1k of code and just 4 bytes of RAM. It is used to standardize 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 develop your project, integrate third-party code, and reuse code on future projects. Once you are familiar with the key specifications, we will look at a software architecture that uses CMSIS to boost productivity and code reuse. Each of the CMSIS specifications is not that complicated and can be learnt easily through the course of this book.

The full documentation for each of the CMSIS specifications can be downloaded from the URL <http://www.keil.com/cmsis>. Each of the CMSIS specifications is integrated into the MDK-Arm toolchain and the CMSIS documentation is available by opening the Run-Time Environment and clicking on the CMSIS link in the description column (Fig. 4.4).

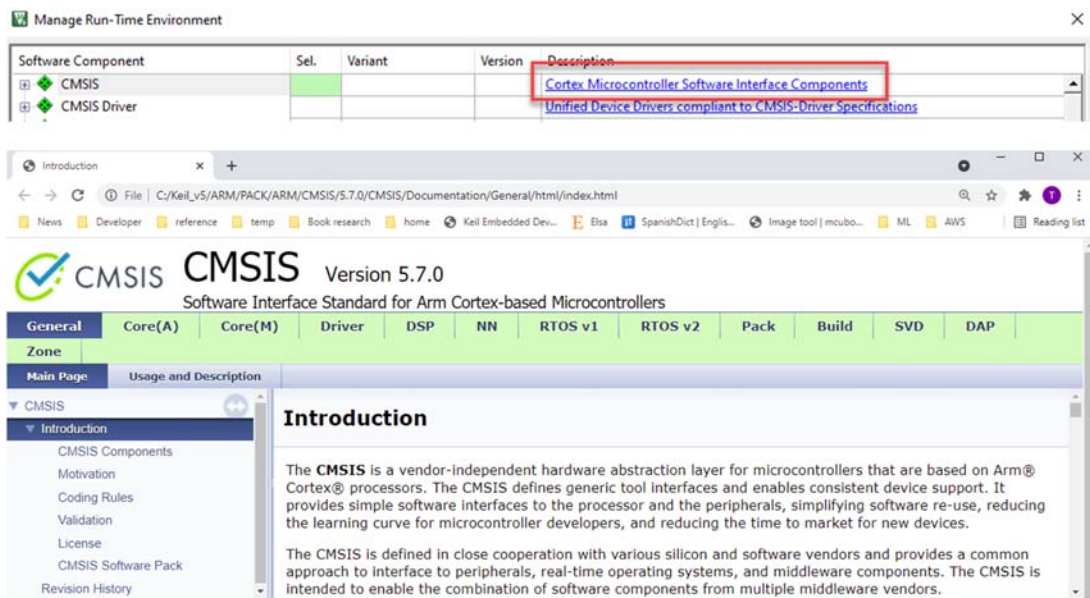


Figure 4.4

The CMSIS documentation is accessed through the Description link in the Run Time Environment Manager.

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 convention for the device peripheral registers. The CMSIS-Core specification includes support for the Instrumentation Trace (ITM) during debug sessions.

CMSIS-RTOS

The CMSIS-RTOS2 specification provides a standard API for a Real-Time Operating System RTOS. This is, in effect, a set of wrapper functions that translate the CMSIS-RTOS2 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-RTOS2 API in [Chapter 10](#), Using a Real Time Operating System. The Keil RTX RTOS was the first RTOS to support the CMSIS-RTOS2 API, and it has been released as an open-source reference implementation. RTX can be compiled with the Keil/Arm, GCC, and IAR compilers. It is licensed with a three-clause BSD license that allows its unrestricted use in commercial and noncommercial applications.

CMSIS-DSP

As we have seen in [Chapter 3](#), Cortex-M Architecture, 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 pass time” and can be quite daunting for all but the simplest systems. To help mere mortals include DSP algorithms in Cortex-M4/M7 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 and Cortex-M7 but can also be compiled to run on the Cortex-M3. We will have a look at using this library in [Chapter 9](#), Practical DSP for Cortex-M Microcontrollers.

CMSIS-Driver

The CMSIS-Driver specification defines a standard API for a range of peripherals common to most microcontroller families. This includes peripherals such as USART, SPI, and I2C as

well as more complex peripherals such as Ethernet MAC and USB. The CMSIS-Drivers are intended to provide a standard target for middleware libraries. For example, this would allow a third-party developer to create a USB library that uses the CMSIS USB driver. Such a library could then be deployed on any device that has a CMSIS USB driver. This greatly speeds up support for new devices and allows library developers to concentrate on adding features to their products rather than continually having to spend time developing support for new devices. It is important to note here that the development of CMSIS-Drivers is really the responsibility of the Silicon Vendor. They should provide a set of drivers when the microcontroller is released.

CMSIS-SVD and DAP

A key problem for software toolchain vendors is to provide debug support for new devices as soon as they are released. One of the main areas that must be customized in the debugger is the “Peripheral View” windows that show the developer the current state of the microcontroller peripherals. With the growth in both the number of Cortex-M vendors and 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 available microcontrollers. To overcome this hurdle, the CMSIS-SVD 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 soon as new microcontrollers are released. Like the CMSIS drivers, the development of the SVD files is the responsibility of the Silicon Vendor.

The CMSIS-DAP specification defines the interface protocol for a hardware debug adapter that sits between the host PC and the Debug Access Port DAP of the microcontroller (Fig. 4.5). This allows any software debugger that supports CMSIS-DAP to connect to any hardware debug adapter that also supports the CMSIS-DAP protocol. There are an increasing number of very low-cost evaluation boards that contain an integrated debugger that connects to a PC using USB. In many cases, this hardware debugger supports the CMSIS-DAP protocol so that it can be connected to any compliant toolchain.

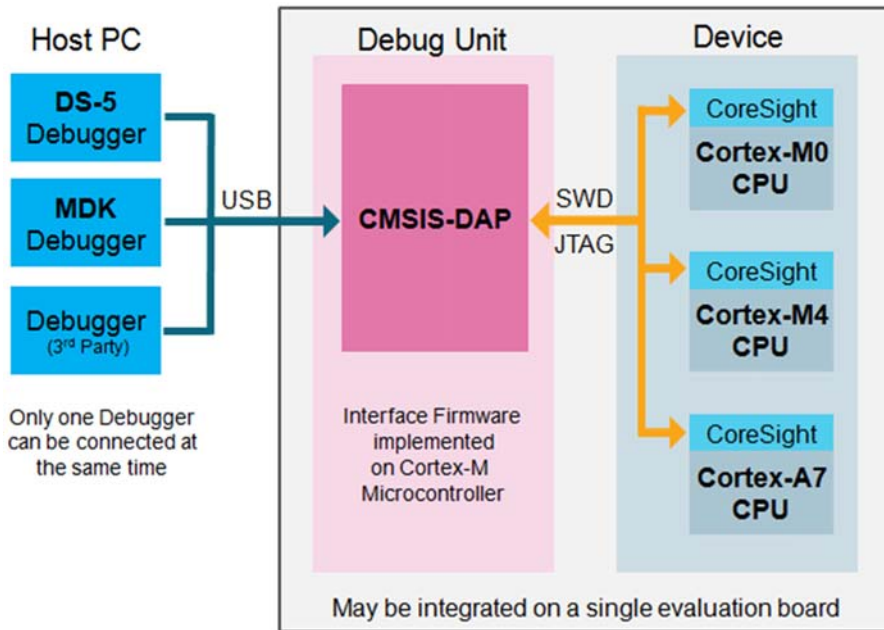


Figure 4.5

The CMSIS-DAP debug adapter provides a standard serial protocol that supports multiple IDEs and debuggers.

CMSIS-Pack

The CMSIS-Core and Driver specifications can be used to develop reusable software components that are portable across device families. The CMSIS-Pack specification defines a method of bundling all of the component elements (software files, examples, help files, templates) into a software Pack. Such a Pack can be downloaded and installed into your toolchain. The Pack also contains information on the software component dependencies, that is, the other files that need to be present when the component is used. This allows you to quickly integrate software from many sources to build a platform on which to develop your application code. As the complexity of Cortex-M microcontrollers is ever-increasing, this is a very important technology to increase a developer's productivity and code reliability.

CMSIS-NN

CMSIS Neural Net is a collection of building block functions that support the efficient design of Neural Net algorithms on Cortex-M microcontrollers. They aim to minimize the overall memory footprint whilst providing around a five-times performance increase over

other typical implementations. CMSIS-NN supports (but is not limited to) the design of the Machine learning algorithms shown in [Table 4.1](#).

Table 4.1: CMSIS NN algorithms

Algorithm	Description
Artificial Neural Net	Used for pattern recognition
Recurrent Neural Net	Pattern recognition in a time series
Convolutional Neural Net	DSP + ANN for image recognition

CMSIS-NN has been integrated with the Tensor Flow Lite framework, and the Tensor Flow Lite interpreter is available as a software pack for use on Cortex-M microcontrollers.

CMSIS-ZONE

The CMSIS-Zone specification is used to manage complex memory maps where each execution region is described as a zone. A markup language and an external utility are used to generate linker script files along with source code configuration files. CMSIS-Zone can be used to manage complex memory maps found in multiprocessor projects, defining execution regions for safety-critical software and also partitioning memory for security applications.

CMSIS-Build

CMSIS-Build defines a generic project file format that allows projects to be shared between different IDEs and build systems. The CMSIS-Build specification also defines workflows for continuous integration servers based on software components supplied in the CMSIS-Pack format. CMSIS-Build also includes the concept of software layers that can be retargeted to different hardware platforms.

Overview of CMSIS-Core

In the remainder of this chapter, we will look at the CMSIS-Core specification and then cover the remaining CMSIS specifications throughout the rest of this book. The CMSIS-Core specification provides a standard set of low-level functions, macros, and peripheral register definitions that allows 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 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 is based on a

consistent set of “C” coding rules called MISRA-C. 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 MISRA-C coding standard is maintained and published by MIRA. MIRA stands for “Motor Industry Research Agency.” MIRA is located near Rugby in England and is responsible for many of the industry standards used by the UK motor industry. In 1998 its software division released the first version of its coding rules formally called “MISRA guidelines for the use of C in-vehicle electronics” (Fig. 4.6).



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 initially 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 Guidelines for the use of C in critical systems.” This change in the title reflects the growing adoption of MISRA-C as a coding standard for general embedded systems. There have been two further updates to the MISRA-C standard in 2008 and 2012 which have expanded the number of rules to 143 with 20 directives. One of the other key attractions of MISRA-C is that it was written by engineers and not 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 slim volume that 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 website. 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` as shown in [Table 4.2](#).

Table 4.2: 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

The typedefs ensure that the expected data size is mapped to the correct C language data type for a given compiler. Using typedefs like this is 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, as shown in [Table 4.3](#). These are typedefs that make clear the type of access each peripheral register allows.

Table 4.3: 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

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 Doxygen. This is a free download released under a GPL license. While Doxygen cannot actually write the documentation for you, it does do much of the dull, boring stuff for you (leaving you to do the exciting documentation work). Doxygen 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 Doxygen is a browsable HTML, but this can be converted to other forms if desired.

The CMSIS source code comments contain specific tags prefixed by the @ symbol for example @brief. These tags are used by Doxygen 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 Doxygen tool is run, it analyses 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 (Fig. 4.7). These include the default startup code with the CMSIS standard vector table. The second file is the system_<device>.c file that contains the necessary code to initialize the microcontroller system peripherals. Finally, the <device>.h header file, which imports the CMSIS header files that contain the CMSIS-Core functions and macros. Generally, these files will be part of the initial project configuration when you create a new project.

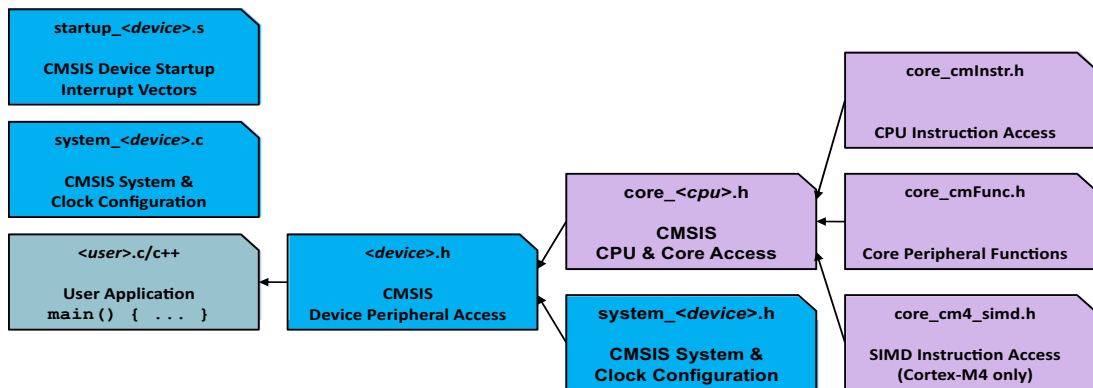


Figure 4.7

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 main stack pointer by loading the value stored in the first four 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 after it leaves the reset vector. Typically, this includes setting up the internal phase-locked loops, configuring the microcontroller clock tree and internal bus structure, and enabling the external bus if required. 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 system peripherals and the Cortex-M processor will be in a fully configured state when the program reaches main(). In the past, this system initializing code was something you would have had to write or crib from example code. On a new microcontroller, this would have been a few days' work, so 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. CMSIS-Core also defines 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 type def, 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 layout 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)
```

The register symbols for each GPIO port can then 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)
```

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 GPIO clock */
    GPIO->MODER  &= ~( ((3UL << 2*12) |
                       (3UL << 2*13) |
                       (3UL << 2*14) |
                       (3UL << 2*15) );       /* PD.12...15 is output */
    GPIO->MODER |= ((1UL << 2*12) |
                  (1UL << 2*13) |
                  (1UL << 2*14) |
                  (1UL << 2*15));
}
```

The microcontroller `<device>.h` 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 in 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 Vendor 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 working quickly and minimize 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 or 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 are “`stdint.h`” a “CMSIS-Core” file for the Cortex-M 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 or Cortex-M7, 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 which are used in the CMSIS definitions and should be used through your application code.

CMSIS-Core Header files

Within the CMSIS-Core specification, there are a small number of defines that are set up for a given microcontroller ([Table 4.4](#)). These can be found in the <device>.h processor include file.

Table 4.4: CMSIS configuration values

CMSIS Define	Description
__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 : Advanced Architecture Features)
__FPU_PRESENT	Defines if an FPU is present (see Chapter 5 : Advanced Architecture Features)
__Vendor_SysTickConfig	Defines if there is a vendor-specific SysTick Configuration

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 in [Table 4.5](#).

Table 4.5: 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 FPU Functions Level 1 Cache Functions (Cortex-M7 only) Cortex-M Vector Extensions Trust Zone for Armv8-M MPU Functions for Armv6/v7-M MPU Functions for Armv6/v7-M PMU Functions for Armv8.1-M

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 with dedicated intrinsic functions and are contained in the CPU instructions group. An extended set of intrinsics are also provided for the Cortex-M4 and Cortex-M7 to access the SIMD instructions. Finally, some standard functions are provided to access the debug ITM.

Interrupts and Exceptions

Management of the NVIC registers may be done by the functions provided in the interrupt and exception group (Table 4.6). These functions allow you to set up an NVIC interrupt channel and manage its priority as well as interrogate the NVIC registers during run time.

Table 4.6: CMSIS Interrupt and exception group

CMSIS Function	Description
NVIC_SetPriorityGrouping	Set the priority grouping
NVIC_GetPriorityGrouping	Read the priority grouping
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
NVIC_SetPriority	Set the active status of an interrupt channel
NVIC_GetPriority	Get the priority of an interrupt channel
NVIC_EncodePriority	Encodes the priority value in terms of priority Group and subgroup
NVIC_DecodePriority	Decodes the priority value in terms of priority Group and subgroup
NVIC_SystemReset	Forces a system reset
NVIC_ClearTargetState	Clears the Interrupt target field in the nonsecure NVIC (Armv8-m secure state only)
NVIC_SetTargetState	Sets the Interrupt target field in the nonsecure NVIC (Armv8-m secure state only)
NVIC_GetVector	Read the address of an interrupt service routine
NVIC_GetEnabledIRQ	Returns the current interrupt enable status of a specified Interrupt channel
NVIC_GetPendingIRQ	Returns the current pending status of a specified Interrupt channel

A configuration function is also provided for the SysTick timer (Table 4.7).

Table 4.7: CMSIS systick function

CMSIS Function	Description
SysTick_Config	Configures the timer and enables the interrupt

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 vector table.

```
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 the 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 `<device>.h`. 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. The SysTick timer input frequency is usually derived from the CPU clock frequency. This allows us to easily setup the SysTick timer to generate a desired periodic interrupt using the

SystemCoreClock frequency. So a one millisecond 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-M processor exceptions. This allows it to program either the system handler priority registers in the system control block or the Interrupt priority registers in the NVIC itself. The NVIC_SetPriority() function also uses NVIC_PRIO_BITS definition to shift the priority value into the active priority bits which have been implemented by the Silicon Vendor.

```
_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 */
    }
```

For Cortex-M3/M4 and Cortex-M7, we have the option to set priority Groups and subgroups as discussed in [Chapter 3](#), Cortex-M Architecture. 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 “Application Interrupt and Reset Control” Register. As discussed in [Chapter 3](#), Cortex-M Architecture, 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->AICR;    /* read old register configuration */
    reg_value &= ~(SCB_AICR_VECTKEY_Msk | SCB_AICR_PRIGROUP_Msk); /* clear bits to change */
    reg_value = (reg_value | ((uint32_t)0x5FA << SCB_AICR_VECTKEY_Pos) | /* Insert write
                                                                    key and priority group */
                (PriorityGroupTmp << 8));
    SCB->AICR = 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 two 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 processor. 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 to ensure that the microcontroller peripherals do in fact go back to their reset configuration. Bit zero of the same register will do a warm reset of the Cortex-M processor. That is, force a reset of the Cortex-M processor but leave the microcontroller registers configured.

Exercise 4.1: 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 Pack Installer.

Select the Boards:Designers Guide Tutorial.

Select the example tab and copy “Ex 4.1 CMSIS 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);
```

We can compare this to the equivalent non CMSIS code...

```
SysTick->VAL = 0x9000;           //Start value for the sys Tick 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->AIRCRCR;
Temp &= ~(SCB_AIRCRCR_VECTKEY_Msk | SCB_AIRCRCR_PRIGROUP_Msk);
Temp = (Temp | ((uint32_t)0x5FA << SCB_AIRCRCR_VECTKEY_Pos) | (5 << 8));
SCB->AIRCRCR = Temp;
```

Although both blocks of code achieve the same thing, the CMSIS version is much faster to write, 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, portability, and maintainability.

CMSIS-Core Register Access

The next group of CMSIS functions gives you direct access to the processor CPU registers (Table 4.8).

Table 4.8: CMSIS CPU register functions

Core Function	Description
__get_Control	Read the CPU 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_PSPLIM	Read the value of the Process stack limit register(Armv8-M only)
__set_PSPLIM	Write a value to the Process stack limit register (Armv8-M only)
__get_MSP	Read the main stack pointer
__set_MSP	Write to the main stack pointer

(Continued)

Table 4.8: (Continued)

Core Function	Description
__get_MSPLIM	Read the value of the Main stack limit register(Armv8-M only)
__set_MSPLIM	Write a value to the Main stack limit register (Armv8-M only)
__get_PRIMASK	Read the PRIMASK
__set_PRIMASK	Write to the PRIMASK
__get_BASEPRI	Read the BASEPRI register
__set_BASEPRI	Write to the BASEPRI register
__set_BASEPRI_MAX	Writes a value to the BASEPRI mask 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	Enables interrupts and all fault handlers
__disable_fault_irq	Disables interrupts and all fault handlers

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()
```

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 Non-Maskable interrupt. The `__set_FAULTMASK()` function can be used to disable all interrupts except the Non-Maskable Interrupt. We will see later how this can be useful when we want to bypass the Memory protection unit. 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 any interrupt at the same priority level or lower will be disabled.

These functions allow you to read the program status register 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 and control register if you are using the Cortex-M4 or Cortex-M7. We will have a closer look at the more advanced operating modes of the Cortex-M processor in [Chapter 5](#), Advanced Architecture Features.

CMSIS-Core CPU Intrinsic Instructions

The CMSIS-Core header also provides two groups of standardized intrinsic functions ([Table 4.9](#)). The first group is common to all Cortex-M processors, and the second provides standard intrinsic for the Cortex-M4 SIMD instructions.

Table 4.9: CMSIS instruction intrinsics

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

The CPU intrinsics provide direct access to Cortex-M processor instructions that are not directly reachable from the “C” language. Using an intrinsic will allow a dedicated single cycle instruction to replace multiple instructions generated by standard “C” code.

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’s instructions that we met in [Chapter 3](#), Cortex-M Architecture. The intrinsic functions also give access to the execution barrier instructions that ensure completion of a data write or instruction 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 have a look at these in [Chapter 5](#), Advanced Architecture Features. The remainder of the CPU intrinsics support single cycle data manipulation functions such as the rotate and reverse bit order instructions.

Exercise 4.2: Intrinsic Bit Manipulation

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

Open the Pack Installer.

Select the Boards:Designers Guide Tutorial.

Select the example tab and Copy “Ex 4.2: 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.

Consider how you would code each intrinsic using standard “C” instructions.

CMSIS SIMD Intrinsics

The next group of CMSIS intrinsics provides direct access to the Cortex-M4 and Cortex-M7 SIMD instructions.

The SIMD instructions provide simultaneous calculations for two sixteen-bit operations or four eight-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 9](#), Practical DSP for Cortex-M Microcontrollers.

CMSIS-Core Debug Functions

The CMSIS-Core functions also provide enhanced debug support through the CoreSight ITM. The CMSIS standard has two dedicated debug specifications CMSIS-SVD and CMSIS-DAP which we will look at in [Chapter 8](#), Debugging with CoreSight. However, the CMSIS-Core specification contains some helpful debug support.

Hardware Breakpoint

First of all, there is a dedicated intrinsic to add a hardware breakpoint to your code.

```
__BKPT(uint8_t value)
```

Using this intrinsic will place a hardware breakpoint instruction at this location in your code. When this point is reached, execution will be halted, and the “value” will be passed to the debugger. During development, the `__BKPT()` intrinsic can be used to trap error conditions and halt the debugger.

Instrumentation Trace

As part of its hardware debug system, the Cortex-M3, Cortex-M4, and Cortex-M7 provide an “ITM” unit. This can be thought of as a debug UART which 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 and from the debugger while the code is running. We will look at using the ITM for additional debug and software testing in [Chapter 8](#), Debugging with CoreSight. For now, there are a number of CMSIS functions that standardize communication with the ITM ([Table 4.10](#)).

Table 4.10: CMSIS debug functions

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

CMSIS Core Functions for Corex-M7

With the release of the Cortex-M7 processor at the end of 2014, the CMSIS-Core specification was extended to provide some additional functions to support the new features introduced by the Corex-M7 ([Table 4.11](#)).

Table 4.11: Cortex-M7 CMSIS core support

CMSIS Cortex-M7 function	Description
Cache Functions	Eleven functions to support the Instruction and Data Caches
FPU Function	One function to support the FPU

As we will see in [Chapter 6](#), Cortex-M7 Processor, the Cortex-M7 introduces Data and Instruction caches to the Cortex-M processor family. The CMSIS cache functions allow you to enable and disable the caches and manage them as your code executes. We will look at these functions in [Chapter 6](#), Cortex-M7 Processor.

MPU Support

All of the Cortex-M processors may be fitted with a Memory protection Unit when the microcontroller is designed. The CMSIS core specification defined a set of support functions that are used to configure and manage the MPU. However, there are two versions of the MPU an original version used by the Armv7-M-based devices (M0 + /M3/M4 and M7) and a later version used by Armv8.x-M-based devices (M23/M33/M55/M85). A set of functions are provided for each MPU version. We will look at the CMSIS support for Armv7-M devices in [Chapter 5](#), Advanced Architecture Features, and support for Armv8.x-M devices in [Chapter 7](#), Armv8-M Architecture.

Armv8-M Support

The CMSIS core specification provides additional support for new Armv8.x-M features ([Table 4.12](#)). We will cover these features in [Chapter 7](#), Armv8-M Architecture.

Table 4.12: Armv8-M CMSIS core support

Feature	Description
MVE	Cortex-M Vector Extensions for Armv8.1
PMU	Performance Monitoring Unit for Armv8.1
TrustZone	Support for the Armv8.x security extension

Conclusion

A good understanding of each CMSIS specification is key to effectively developing applications for any Cortex-M-based microcontroller. In this chapter we have introduced each CMSIS specification and taken a detailed look at the CMSIS-Core specification. We will look at the remaining CMSIS specifications throughout the rest of this book.