Cortex Microcontroller Software Interface Standard

This file describes the Cortex Microcontroller Software Interface Standard (CMSIS).

Version: 1.10 - 24. Feb. 2009

Revision History

- Version 1.00: initial release.
- Version 1.01: added __LDREX*x*, __STREX*x*, and __CLREX.
- Version 1.02: added Cortex-M0.
- Version 1.10: second review.

Contents

About

The **Cortex Microcontroller Software Interface Standard (CMSIS)** answers the challenges that are faced when software components are deployed to physical microcontroller devices based on a Cortex-M0 / Cortex-M1 or Cortex-M3 processor. The CMSIS will be also expanded to future Cortex-M processor cores (the term Cortex-Mx is used to indicate that). The CMSIS is defined in close co-operation with various silicon and software vendors and provides a common approach to interface to peripherals, real-time operating systems, and middleware components.

ARM provides as part of the CMSIS the following software layers that are available for various compiler implementations:

- **Core Peripheral Access Layer**: contains name definitions, address definitions and helper functions to access core registers and peripherals. It defines also an device independent interface for RTOS Kernels that includes debug channel definitions.
- **Middleware Access Layer:** provides common methods to access peripherals for the software industry. The Middleware Access Layer is adapted by the Silicon Vendor for the device specific peripherals used by middleware components. The middleware access layer is currently in development and not yet part of this documentation

These software layers are expanded by Silicon partners with:

- **Device Peripheral Access Layer**: provides definitions for all device peripherals
- **Access Functions for Peripherals (optional)**: provides additional helper functions for peripherals

CMSIS defines for a Cortex-Mx Microcontroller System:

- A common way to access peripheral registers and a common way to define exception vectors.
- The register names of the **Core Peripherals** and the names of the **Core Exception Vectors**.
- An device independent interface for RTOS Kernels including a debug channel.
- Interfaces for middleware components (TCP/IP Stack, Flash File System).

By using CMSIS compliant software components, the user can easier re-use template code. CMSIS is intended to enable the combination of software components from multiple middleware vendors.

Coding Rules and Conventions

The following section describes the coding rules and conventions used in the CMSIS implementation. It contains also information about data types and version number information.

Essentials

- The CMSIS C code conforms to MISRA 2004 rules. In case of MISRA violations, there are disable and enable sequences for PC-LINT inserted.
- ANSI standard data types defined in the ANSI C header file **<stdint.h>** are used.
- #define constants that include expressions must be enclosed by parenthesis.
- Variables and parameters have a complete data type.
- All functions in the **Core Peripheral Access Layer** are re-entrant.
- The **Core Peripheral Access Layer** has no blocking code (which means that wait/query loops are done at other software layers such as the **Middleware Access Layer**).
- For each exception/interrupt there is definition for:
  - an exception/interrupt handler with the postfix **_Handler** (for exceptions) or **_IRQHandler** (for interrupts).
  - a default exception/interrupt handler (weak definition) that contains an endless loop.
  - a #define of the interrupt number with the postfix **_IRQn**.

Recommendations

The CMSIS recommends the following conventions for identifiers.

- **CAPITAL** names to identify Core Registers, Peripheral Registers, and CPU Instructions.
- **CamelCase** names to identify peripherals access functions and interrupts.
- **PERIPHERAL_** prefix to identify functions that belong to specify peripherals.
- **Doxygen** comments for all functions are included as described under **Function Comments** below.

Comments

- Comments use the ANSI C90 style (*/* comment */*) or C++ style (*// comment*). It is assumed that the programming tools support today consistently the C++ comment style.
- **Function Comments** provide for each function the following information:
  - one-line brief function overview.
  - detailed parameter explanation.
  - detailed information about return values.
  - detailed description of the actual function.

Doxygen Example:

```
/**
 * @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.
 */
```

Data Types and IO Type Qualifiers

The **Cortex-Mx HAL** uses the standard types from the standard ANSI C header file **<stdint.h>**. **IO Type Qualifiers** are used to specify the access to peripheral variables. IO Type Qualifiers are indented to be used for automatic generation of debug information of peripheral registers.

IO Type Qualifier	#define	Description
__I	volatile const	Read access only
__O	volatile	Write access only
__IO	volatile	Read and write access

CMSIS Version Number

File **core_cm3.h** contains the version number of the CMSIS with the following define:

```
#define __CM3_CMSIS_VERSION_MAIN  (0x00)      /* [31:16] main version       */
#define __CM3_CMSIS_VERSION_SUB   (0x03)      /* [15:0]  sub version        */
#define __CM3_CMSIS_VERSION       ((__CM3_CMSIS_VERSION_MAIN << 16) | __CM3_CMSIS_VERSION_SUB)
```

File **core_cm0.h** contains the version number of the CMSIS with the following define:

```
#define __CM0_CMSIS_VERSION_MAIN  (0x00)      /* [31:16] main version       */
#define __CM0_CMSIS_VERSION_SUB   (0x00)      /* [15:0]  sub version        */
#define __CM0_CMSIS_VERSION       ((__CM0_CMSIS_VERSION_MAIN << 16) | __CM0_CMSIS_VERSION_SUB)
```

CMSIS Cortex Core

File **core_cm3.h** contains the type of the CMSIS Cortex-Mx with the following define:

```
#define __CORTEX_M                (0x03)
```

File **core_cm0.h** contains the type of the CMSIS Cortex-Mx with the following define:

```
#define __CORTEX_M                (0x00)
```

CMSIS Files

This section describes the Files provided in context with the CMSIS to access the Cortex-Mx hardware and peripherals.

File	Provider	Description
device.h	Device specific (provided by silicon partner)	Defines the peripherals for the actual device. The file may use several other include files to define the peripherals of the actual device.
core_cm0.h	ARM (for RealView ARMCC, IAR, and GNU GCC)	Defines the core peripherals for the Cortex-M0 CPU and core peripherals.
core_cm3.h	ARM (for RealView ARMCC, IAR, and GNU GCC)	Defines the core peripherals for the Cortex-M3 CPU and core peripherals.
core_cm0.c	ARM (for RealView ARMCC, IAR, and GNU GCC)	Provides helper functions that access core registers.
core_cm0.c	ARM (for RealView ARMCC, IAR, and GNU GCC)	Provides helper functions that access core registers.
startup_*device*	ARM (adapted by compiler partner / silicon partner)	Provides the Cortex-Mx startup code and the complete (device specific) Interrupt Vector Table
system_*device*	ARM (adapted by silicon partner)	Provides a device specific configuration file for the device. It configures the device initializes typically the oscillator (PLL) that is part of the microcontroller device

device.h

The file ***device.h*** is provided by the silicon vendor and is the **central include file** that the application programmer is using in the C source code. This file contains:

- **Interrupt Number Definition**: provides interrupt numbers (IRQn) for all core and device specific exceptions and interrupts.
- **Configuration for core_cm0.h / core_cm3.h**: reflects the actual configuration of the Cortex-Mx processor that is part of the actual device. As such the file **core_cm0.h / core_cm3.h** is included that implements access to processor registers and core peripherals.
- **Device Peripheral Access Layer**: provides definitions for all device peripherals. It contains all data structures and the address mapping for the device specific peripherals.
- **Access Functions for Peripherals (optional)**: provides additional helper functions for peripherals that are useful for programming of these peripherals. Access Functions may be provided as inline functions or can be extern references to a device specific library provided by the silicon vendor.

Interrupt Number Definition

To access the device specific interrupts the device.h file defines IRQn numbers for the complete device using a enum typedef as shown below:

```
typedef enum IRQn
{
/******  Cortex-M3 Processor Exceptions/Interrupt Numbers ***********************************************/
  NonMaskableInt_IRQn             = -14,      /*!< 2 Non Maskable Interrupt                              */
  MemoryManagement_IRQn           = -12,      /*!< 4 Cortex-M3 Memory Management Interrupt               */
  BusFault_IRQn                   = -11,      /*!< 5 Cortex-M3 Bus Fault Interrupt                       */
  UsageFault_IRQn                 = -10,      /*!< 6 Cortex-M3 Usage Fault Interrupt                     */
  SVCall_IRQn                     = -5,       /*!< 11 Cortex-M3 SV Call Interrupt                        */
  DebugMonitor_IRQn               = -4,       /*!< 12 Cortex-M3 Debug Monitor Interrupt                  */
  PendSV_IRQn                     = -2,       /*!< 14 Cortex-M3 Pend SV Interrupt                        */
  SysTick_IRQn                    = -1,       /*!< 15 Cortex-M3 System Tick Interrupt                    */
/******  STM32 specific Interrupt Numbers ***************************************************************/
  WWDG_STM_IRQn                   = 0,        /*!< Window WatchDog Interrupt                             */
  PVD_STM_IRQn                    = 1,        /*!< PVD through EXTI Line detection Interrupt             */
  :
  :
  } IRQn_Type;
```

Configuration for core_cm0.h / core_cm3.h

The Cortex-Mx core configuration options which are defined for each device implementation. Some configuration options are reflected in the CMSIS layer using the #define settings described below.

To access core peripherals file ***device.h*** includes file **core_cm0.h / core_cm3.h**. Several features in **core_cm0.h / core_cm3.h** are configured by the following defines that must be defined before **#include <core_cm0.h>** / **#include <core_cm3.h>** preprocessor command.

#define	File	Value	Description
__NVIC_PRIO_BITS	core_cm0.h	(2)	Number of priority bits implemented in the NVIC (device specific)
__NVIC_PRIO_BITS	core_cm3.h	(2 ... 8)	Number of priority bits implemented in the NVIC (device specific)
__MPU_PRESENT	core_cm0.h, core_cm3.h	(0, 1)	Defines if an MPU is present or not
__Vendor_SysTickConfig	core_cm0.h, core_cm3.h	(1)	When this define is setup to 1, the **SysTickConfig** function in **core_cm3.h** is excluded. In this case the ***device.h*** file must contain a vendor specific implementation of this function.

Device Peripheral Access Layer

Each peripheral uses a **PERIPHERAL_** prefix to identify peripheral registers and functions that access this specific peripheral. If more than one peripheral of the same type exists, identifiers have a postfix (digit or letter). For example:

- UART_Type: defines the generic register layout for all UART channels in a device.
- UART1: is a pointer to a register structure that refers to a specific UART. For example UART1->DR is the data register of UART1.
- UART_SendChar(UART1, c): is a generic function that works with all UART's in the device. To communicate the UART that it accesses the first parameter is a pointer to the actual UART register structure.
- UART1_SendChar(c): is an UART1 specific implementation (in this case the send function).

Minimal Requiements

To access the peripheral registers and related function in a device the files ***device.h*** and **core_cm0.h** / **core_cm3.h** defines as a minimum:

- The **Register Layout Typedef** for each peripheral that defines all register names. Names that start with RESERVE are used to introduce space into the structure to adjust the addresses of the peripheral registers. For example:

```
typedef struct {
  __IO uint32_t CTRL;      /* SysTick Control and Status Register */
  __IO uint32_t LOAD;      /* SysTick Reload Value Register       */
  __IO uint32_t VAL;       /* SysTick Current Value Register      */
  __I  uint32_t CALIB;     /* SysTick Calibration Register        */
  } SysTick_Type;
```

- **Base Address** for each peripheral (in case of multiple peripherals that use the same **register layout typedef** multiple base addresses are defined). For example:

```
#define SysTick_BASE (SCS_BASE + 0x0010)            /* SysTick Base Address */
```

- **Access Definition** for each peripheral (in case of multiple peripherals that use the same **register layout typedef** multiple access definitions exist, i.e. UART0, UART1). For Example:

```
#define SysTick ((SysTick_Type *) SysTick_BASE)     /* SysTick access definition */
```

These definitions allow to access the peripheral registers from user code with simple assignments like:

```
SysTick->CTRL = 0;
```

Optional Features

In addition the ***device.h*** file may define:

- #define constants that simplify access to the peripheral registers. These constant define bit-positions or other specific patterns are that required for the programming of the peripheral registers. The identifiers used start with the name of the **PERIPERHAL_**. It is recommended to use CAPITAL letters for such #define constants.
- Functions that perform more complex functions with the peripheral (i.e. status query before a sending register is accessed). Again these function start with the name of the **PERIPHERAL_**.

core_cm0.h and core_cm0.c

File **core_cm0.h** describes the data structures for the Cortex-M0 core peripherals and does the address mapping of this structures. It also provides basic access to the Cortex-M0 core registers and core peripherals with efficient functions (defined as **static inline**).

File **core_cm0.c** defines several helper functions that access processor registers.

Together these files implement the Core Peripheral Access Layer for a Cortex-M0.

core_cm3.h and core_cm3.c

File **core_cm3.h** describes the data structures for the Cortex-M3 core peripherals and does the address mapping of this structures. It also provides basic access to the Cortex-M3 core registers and core peripherals with efficient functions (defined as **static inline**).

File **core_cm3.c** defines several helper functions that access processor registers.

Together these files implement the Core Peripheral Access Layer for a Cortex-M3.

startup_*device*

A template file for **startup_*device*** is provided by ARM for each supported compiler. It is adapted by the silicon vendor to include interrupt vectors for all device specific interrupt handlers. Each interrupt handler is defined as ***weak*** function to an dummy handler. Therefore the interrupt handler can be directly used in application software without any requirements to adapt the **startup_*device*** file.

The following exception names are fixed and define the start of the vector table for a Cortex-M0:

```
__Vectors       DCD     __initial_sp              ; Top of Stack
                DCD     Reset_Handler             ; Reset Handler
                DCD     NMI_Handler               ; NMI Handler
                DCD     HardFault_Handler         ; Hard Fault Handler
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     SVC_Handler               ; SVCall Handler
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     PendSV_Handler            ; PendSV Handler
                DCD     SysTick_Handler           ; SysTick Handler
```

The following exception names are fixed and define the start of the vector table for a Cortex-M3:

```
__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
                DCD     BusFault_Handler          ; Bus Fault Handler
                DCD     UsageFault_Handler        ; Usage Fault Handler
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     SVC_Handler               ; SVCall Handler
                DCD     DebugMon_Handler          ; Debug Monitor Handler
                DCD     0                         ; Reserved
                DCD     PendSV_Handler            ; PendSV Handler
                DCD     SysTick_Handler           ; SysTick Handler
```

In the following examples for device specific interrupts are shown:

```
; External Interrupts
                DCD     WWDG_IRQHandler           ; Window Watchdog
                DCD     PVD_IRQHandler            ; PVD through EXTI Line detect
                DCD     TAMPER_IRQHandler         ; Tamper
```

Device specific interrupts must have a dummy function that can be overwritten in user code. Below is an example for this dummy function.

```
Default_Handler PROC
                EXPORT WWDG_IRQHandler   [WEAK]
                EXPORT PVD_IRQHandler    [WEAK]
                EXPORT TAMPER_IRQHandler [WEAK]
                :
                :
                WWDG_IRQHandler
                PVD_IRQHandler
                TAMPER_IRQHandler
                :
```

```
                :
                B .
                ENDP
```

The user application may simply define an interrupt handler function by using the handler name as shown below.

```
void WWDG_IRQHandler(void)
{
  :
  :
}
```

system_*device*.c

A template file for **system_*device*.c** is provided by ARM but adapted by the silicon vendor to match their actual device. As a **minimum requirement** this file must provide a device specific system configuration function and a global variable that contains the system frequency. It configures the device and initializes typically the oscillator (PLL) that is part of the microcontroller device.

The file **system_*device*.c** must provide as a minimum requirement the SystemInit function as shown below.

Function Definition	Description
void SystemInit (void)	Setup the microcontroller system. Typically this function configures the oscillator (PLL) that is part of the microcontroller device. For systems with variable clock speed it also updates the variable SystemFrequency.

Also part of the file **system_*device*.c** is the variable **SystemFrequency** which contains the current CPU clock speed shown below.

Variable Definition	Description
uint32_t SystemFrequency	Contains the system frequency (which is the system clock frequency supplied to the SysTick timer and the processor core clock). This variable can be used by the user application after the call to the function SystemInit() to setup the SysTick timer or configure other parameters. It may also be used by debugger to query the frequency of the debug timer or configure the trace clock speed. This variable may also be defined in the **const** space. The compiler must be configured to avoid the removal of this variable in case that the application program is not using it. It is important for debug systems that the variable is physically present in memory so that it can be examined to configure the debugger.

Note

- The above definitions are the minimum requirements for the file **system_*device*.c**. This file may export more functions or variables that provide a more flexible configuration of the microcontroller system.

Core Peripheral Access Layer

Cortex-Mx Core Register Access

The following functions are defined in **core_cm0.h** / **core_cm3.h** and provide access to Cortex-Mx core registers.

Function Definition	Core	Core Register	Description
void __enable_irq (void)	M0, M3	PRIMASK = 0	Global Interrupt enable (using the instruction **CPSIE i**)
void __disable_irq (void)	M0, M3	PRIMASK = 1	Global Interrupt disable (using the instruction **CPSID i**)
void __set_PRIMASK (uint32_t value)	M0, M3	PRIMASK = value	Assign value to Priority Mask Register (using the instruction **MSR**)
uint32_t __get_PRIMASK (void)	M0, M3	return PRIMASK	Return Priority Mask Register (using the instruction **MRS**)
void __enable_fault_irq (void)	M3	FAULTMASK = 0	Global Fault exception and Interrupt enable (using the instruction **CPSIE f**)
void __disable_fault_irq (void)	M3	FAULTMASK = 1	Global Fault exception and Interrupt disable (using the instruction **CPSID f**)
void __set_FAULTMASK (uint32_t value)	M3	FAULTMASK = value	Assign value to Fault Mask Register (using the instruction **MSR**)
uint32_t __get_FAULTMASK (void)	M3	return FAULTMASK	Return Fault Mask Register (using the instruction **MRS**)
void __set_BASEPRI (uint32_t value)	M3	BASEPRI = value	Set Base Priority (using the instruction **MSR**)
uiuint32_t __get_BASEPRI (void)	M3	return BASEPRI	Return Base Priority (using the instruction **MRS**)
void __set_CONTROL (uint32_t value)	M0, M3	CONTROL = value	Set CONTROL register value (using the instruction **MSR**)
uint32_t __get_CONTROL (void)	M0,	return CONTROL	Return Control Register Value (using the instruction **MRS**)

	M3		
void __set_PSP (uint32_t TopOfProcStack)	M0, M3	PSP = TopOfProcStack	Set Process Stack Pointer value (using the instruction **MSR**)
uint32_t __get_PSP (void)	M0, M3	return PSP	Return Process Stack Pointer (using the instruction **MRS**)
void __set_MSP (uint32_t TopOfMainStack)	M0, M3	MSP = TopOfMainStack	Set Main Stack Pointer (using the instruction **MSR**)
uint32_t __get_MSP (void)	M0, M3	return MSP	Return Main Stack Pointer (using the instruction **MRS**)

Cortex-Mx Instruction Access

The following functions are defined in **core_cm0.h** / **core_cm3.h**and generate specific Cortex-Mx instructions. The functions are implemented in the file **core_cm0.c** / **core_cm3.c**.

Name	Core	Generated CPU Instruction	Description
void __WFI (void)	M0, M3	WFI	Wait for Interrupt
void __WFE (void)	M0, M3	WFE	Wait for Event
void __SEV (void)	M0, M3	SEV	Set Event
void __ISB (void)	M0, M3	ISB	Instruction Synchronization Barrier
void __DSB (void)	M0, M3	DSB	Data Synchronization Barrier
void __DMB (void)	M0, M3	DMB	Data Memory Barrier
uint32_t __REV (uint32_t value)	M0, M3	REV	Reverse byte order in integer value.
uint32_t __REV16 (uint16_t value)	M0, M3	REV16	Reverse byte order in unsigned short value.
sint32_t __REVSH (sint16_t value)	M0, M3	REVSH	Reverse byte order in signed short value with sign extension to integer.
uint32_t __RBIT (uint32_t value)	M3	RBIT	Reverse bit order of value
uint8_t __LDREXB (uint8_t *addr)	M3	LDREXB	Load exclusive byte
uint16_t __LDREXH (uint16_t *addr)	M3	LDREXH	Load exclusive half-word
uint32_t __LDREXW (uint32_t *addr)	M3	LDREXW	Load exclusive word
uint32_t __STREXB (uint8_t value, uint8_t *addr)	M3	STREXB	Store exclusive byte
uint32_t __STREXB (uint16_t value, uint16_t *addr)	M3	STREXH	Store exclusive half-word
uint32_t __STREXB (uint32_t value, uint32_t *addr)	M3	STREXW	Store exclusive word
void __CLREX (void)	M3	CLREX	Remove the exclusive lock created by __LDREXB, __LDREXH, or __LDREXW

NVIC Access Functions

The CMSIS provides access to the NVIC via the register interface structure and several helper functions that simplify the setup of the NVIC. The CMSIS HAL uses IRQ numbers (IRQn) to identify the interrupts. The first device interrupt has the IRQn value 0. Therefore negative IRQn values are used for processor core exceptions.

For the IRQn values of core exceptions the file ***device.h*** provides the following enum names.

Core Exception enum Value	Core	IRQn	Description
NonMaskableInt_IRQn	M0, M3	-14	Cortex-Mx Non Maskable Interrupt
MemoryManagement_IRQn	M3	-12	Cortex-Mx Memory Management Interrupt
BusFault_IRQn	M3	-11	Cortex-Mx Bus Fault Interrupt
UsageFault_IRQn	M3	-10	Cortex-Mx Usage Fault Interrupt
SVCall_IRQn	M0, M3	-5	Cortex-Mx SV Call Interrupt
DebugMonitor_IRQn	M3	-4	Cortex-Mx Debug Monitor Interrupt
PendSV_IRQn	M0, M3	-2	Cortex-Mx Pend SV Interrupt
SysTick_IRQn	M0, M3	-1	Cortex-Mx System Tick Interrupt

The following functions simplify the setup of the NVIC. The functions are defined as **static inline**.

Name	Core	Parameter	Description
void NVIC_SetPriorityGrouping(uint32_t priority_grouping)	M0, M3	Priority Grouping Value	Set the Priority Grouping (Groups . Subgroups)
void NVIC_EnableIRQ(IRQn_Type IRQn)	M0, M3	IRQ Number	Enable IRQn
void NVIC_DisableIRQ(IRQn_Type IRQn)	M0, M3	IRQ Number	Disable IRQn
uint32_t NVIC_GetPendingIRQ (IRQn_Type IRQn)	M0, M3	IRQ Number	Return true (IRQ-Number) if IRQn is pending
void NVIC_SetPendingIRQ (IRQn_Type IRQn)	M0, M3	IRQ Number	Set IRQn Pending
void NVIC_ClearPendingIRQ (IRQn_Type IRQn)	M0, M3	IRQ Number	Clear IRQn Pending Status
uint32_t NVIC_GetActive (IRQn_Type IRQn)	M3	IRQ Number	Return the IRQn of the active interrupt
void NVIC_SetPriority (IRQn_Type IRQn, uint32_t priority)	M0, M3	IRQ Number, Priority	Set Priority for IRQn (not threadsafe for Cortex-M0)
uint32_t NVIC_GetPriority (IRQn_Type IRQn)	M0, M3	IRQ Number	Get Priority for IRQn
void NVIC_SystemReset (void)	M0, M3	(void)	Resets the System

Note

- The processor exceptions have negative enum values. Device specific interrupts have positive enum values and start with 0. The values are defined in ***device.h*** file.

SysTick Configuration Function

The following function is used to configure the SysTick timer and start the SysTick interrupt.

Name	Parameter	Description
uint32_t SysTickConfig (uint32_t ticks)	ticks is SysTick counter reload value	Setup the SysTick timer and enable the SysTick interrupt. After this call the SysTick timer creates interrupts with the specified time interval. Return: 0 when successful, 1 on failure.

Cortex-M3 ITM Debug Access

The Cortex-M3 incorporates the Instrumented Trace Macrocell (ITM) that provides together with the Serial Viewer Output trace capabilities for the microcontroller system. The ITM has 32 communication channels; two ITM communication channels are used by CMSIS to output the following information:

- ITM Channel 0: implements the **ITM_putchar** function which can be used for printf-style output via the debug interface.
- ITM Channel 31: is reserved for the RTOS kernel and can be used for kernel awareness debugging.

Note

- The ITM channel 31 is selected for the RTOS kernel since some kernels may use the Privileged level for program execution. ITM channels have 4 groups with 8 channels each, whereby each group can be configured for access rights in the Unprivileged level. The ITM channel 0 may be therefore enabled for the user task whereas ITM channel 31 may be accessible only in Privileged level from the RTOS kernel itself.

The prototype of the **ITM_putchar** routine is shown in the table below.

Name	Parameter	Description
void uint32_t ITM_putchar(uint32_t chr)	character to output	The function outputs a character via the ITM channel 0. The function returns when no debugger is connected that has booked the output. It is blocking when a debugger is connected, but the previous character send is not transmitted. Return: the input character 'chr'.

Example for the usage of the ITM Channel 31 for RTOS Kernels:

```
// check if debugger connected and ITM channel enabled for tracing
if ((CoreDebug->DEMCR & CoreDebug_DEMCR_TRCENA) &&
(ITM->TCR & ITM_TCR_ITMENA) &&
(ITM->TER & (1UL << 31))) {
  // transmit trace data
  while (ITM->PORT31_U32 == 0);
  ITM->PORT[31].u8 = task_id;       // id of next task
  while (ITM->PORT[31].u32 == 0);
  ITM->PORT[31].u32 = task_status; // status information
}
```

CMSIS Example

The following section shows a typical example for using the CMSIS layer in user applications.

```
#include <device.h>                                 // file name depends on the device used.

void SysTick_Handler (void)  {                      // SysTick Interrupt Handler
  ;
}

void TIM1_UP_IRQHandler (void)  {                   // Timer Interrupt Handler
  ;
}

void timer1_init(int frequency) {
                                                    // set up Timer (device specific)
  NVIC_SetPriority (TIM1_UP_IRQn, 1);               // Set Timer priority
  NVIC_EnableIRQ (TIM1_UP_IRQn);                    // Enable Timer Interrupt
}

void main (void) {
  SystemInit ();

  if (SysTick_Config (SystemFrequency / 1000)) { // Setup SysTick Timer for 1 msec interrupts
    :                                               // Handle Error
    :
    while (1);
  }

  timer1_init ();                                   // device specific timer

  while (1);
}
```