

CMSIS

The **Cortex Microcontroller Software Interface Standard** (CMSIS) provides a ground-up software framework for embedded applications that run on Cortex-M based microcontrollers. CMSIS enables consistent and simple software interfaces to the processor and the peripherals, simplifying software reuse, reducing the learning curve for microcontroller developers.

NOTE

This chapter is a reference section. The chapter [Create Applications](#) on page 45 shows you how to use CMSIS for creating application code.

The CMSIS, defined in close cooperation with various silicon and software vendors, provides a common approach to interface peripherals, real-time operating systems, and middleware components.

The CMSIS application software components are:

- **CMSIS-CORE:** Defines the API for the Cortex-M processor core and peripherals and includes a consistent system startup code. The software components **::CMSIS:CORE** and **::Device:Startup** are all you need to create and run applications on the native processor that uses exceptions, interrupts, and device peripherals.
- **CMSIS-RTOS RTX:** Provides a standardized real-time operating system API and enables software templates, middleware, libraries, and other components that can work across supported RTOS systems. This manual explains the usage of the CMSIS-RTOS RTX implementation.
- **CMSIS-DSP:** Is a library collection for digital signal processing (DSP) with over 60 Functions for various data types: fix-point (fractional q7, q15, q31) and single precision floating-point (32-bit).
- **CMSIS-Driver:** Is a software API that describes peripheral driver interfaces for middleware stacks and user applications. The CMSIS-Driver API is designed to be generic and independent of a specific RTOS making it reusable across a wide range of supported microcontroller devices.

CMSIS-CORE

This section explains the usage of CMSIS-CORE in applications that run natively on a Cortex-M processor. This type of operation is known as *bare-metal*, because it uses no real-time operating system.

Using CMSIS-CORE

A native Cortex-M application with CMSIS uses the software component **::CMSIS:CORE**, which should be used together with the software component **::Device:Startup**. These components provide the following central files:

NOTE

In actual file names, <device> is the name of the microcontroller device.

The `startup_<device>.s` file with reset handler and exception vectors.

The `system_<device>.c` configuration file for basic device setup (clock and memory BUS).

The `<device>.h` include file for user code access to the microcontroller device.

The `<device>.h` header file is included in C source files and defines:

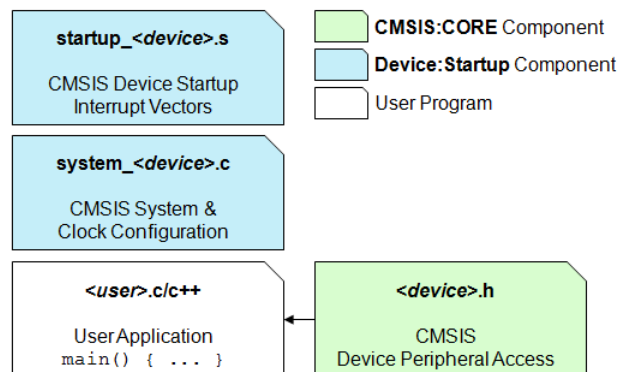
Peripheral access with standardized register layout.

Access to interrupts and exceptions, and the Nested Interrupt Vector Controller (NVIC).

Intrinsic functions to generate special instructions, for example to activate sleep mode.

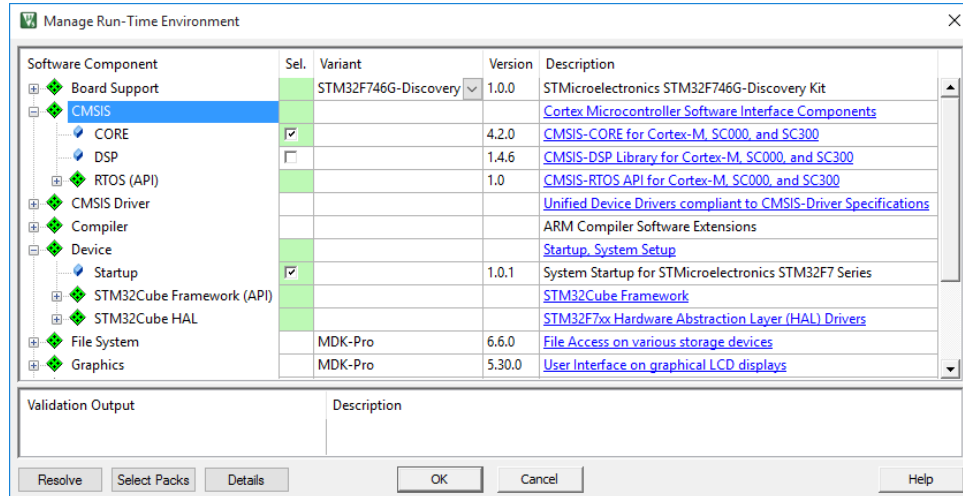
Systick timer (SYSTICK) functions to configure and start a periodic timer interrupt.

Debug access for *printf*-style I/O and ITM communication via on-chip CoreSight™.



Adding Software Components to the Project

The files for the components **::CMSIS:CORE** and **::Device:Startup** are added to a project using the μ Vision dialog **Manage Run-Time Environment**. Just select the software components as shown below:



The μ Vision environment adds the related files.

Source Code Example

The following source code lines show the usage of the CMSIS-CORE layer.

Example of using the CMSIS-CORE layer

```
#include "stm32f4xx.h"           // File name depends on device used

uint32_t volatile msTicks;      // Counter for millisecond Interval
uint32_t volatile frequency;    // Frequency for timer

void SysTick_Handler (void) {   // SysTick Interrupt Handler
    msTicks++;                  // Increment Counter
}

void WaitForTick (void) {
    uint32_t curTicks;
    curTicks = msTicks;
    while (msTicks == curTicks) { // Save Current SysTick Value
        __WFE ();                // Wait for next SysTick Interrupt
    }                             // Power-Down until next Event
}

void TIM1_UP_IRQHandler (void) { // Timer Interrupt Handler
    ; // Add user code here
}
```

```

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
}

// Configure & Initialize the MCU
void Device_Initialization (void) {
    if (SysTick_Config (SystemCoreClock / 1000)) {    // SysTick lms
        : // Handle Error
    }
    timer1_init (frequency);    // Setup device-specific timer
}

// The processor clock is initialized by CMSIS startup + system file
int main (void) {    // User application starts here
    Device_Initialization ();    // Configure & Initialize MCU

    while (1) {    // Endless Loop (the Super-Loop)
        disable_irq ();    // Disable all interrupts
        // Get_InputValues ();
        enable_irq ();    // Enable all interrupts
        // Process_Values ();
        WaitForTick ();    // Synchronize to SysTick Timer
    }
}

```

For more information, right-click the group CMSIS in the Project window, and choose **Open Documentation**, or refer to the CMSIS-CORE documentation <http://www.keil.com/cmsis/core>.

CMSIS-CORE Version 4.20
CMSIS-CORE support for Cortex-M processor-based devices

General | **Core** | Driver | DSP | RTOS API | RTX | Pack | SVD | DAP

Main Page | Usage and Description | Reference

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

Generated on Fri Sep 11 2015 14:39:59 for CMSIS-CORE by ARM Ltd. All rights reserved.

CMSIS-RTOS RTX

This section introduces the CMSIS-RTOS RTX Real-Time Operating System, describes its features and advantages, and explains configuration settings of this RTOS.

NOTE

MDK is compatible with many third-party RTOS solutions. However, CMSIS-RTOS RTX is well integrated into MDK, is feature-rich and tailored towards the requirements of deeply embedded systems.

Software Concepts

There are two basic design concepts for embedded applications:

Infinite Loop Design: involves running the program as an endless loop. Program functions (threads) are called from within the loop, while interrupt service routines (ISRs) perform time-critical jobs including some data processing.

RTOS Design: involves running several threads with a **Real-Time Operating System (RTOS)**. The RTOS provides inter-thread communication and time management functions. A preemptive RTOS reduces the complexity of interrupt functions, because high-priority threads can perform time-critical data processing.

Infinite Loop Design

Running an embedded program in an endless loop is an adequate solution for simple embedded applications. Time-critical functions, typically triggered by hardware interrupts, execute in an ISR that also performs any required data processing. The main loop contains only basic operations that are not time-critical and run in the background.

Advantages of an RTOS Kernel

RTOS kernels, like the CMSIS-RTOS RTX, are based on the idea of parallel execution threads (tasks). As in the real world, your application will have to fulfill multiple different tasks. An RTOS-based application recreates this model in your software with various benefits:

Thread priority and run-time scheduling is handled by the RTOS Kernel, using a proven code base.

The RTOS provides a well-defined interface for communication between threads.

A pre-emptive multi-tasking concept simplifies the progressive enhancement of an application even across a larger development team. New functionality can be added without risking the response time of more critical threads.

Infinite loop software concepts often poll for occurred interrupts. In contrast, RTOS kernels themselves are interrupt driven and can largely eliminate polling. This allows the CPU to sleep or process threads more often.

Modern RTOS kernels are transparent to the interrupt system, which is mandatory for systems with hard real-time requirements. Communication facilities can be used for IRQ-to-task communication and allow top-half/bottom-half handling of your interrupts.

Using CMSIS-RTOS RTX

CMSIS-RTOS RTX is implemented as a library and exposes the functionality through the header file *cmsis_os.h*.

Execution of the CMSIS-RTOS RTX starts with the function *main()* as the first thread. This has the benefit that developers can initialize other middleware libraries that create threads internally, but the remaining part of the user application uses just the **main** thread. Consequently, the usage of the RTOS can be invisible to the application programmer, but libraries can use CMSIS-RTOS RTX features.

The software component **::CMSIS:RTOS:Keil RTX** must be used together with the components **::CMSIS:CORE** and **::Device:Startup**. Selecting these components provides the following central CMSIS-RTOS RTX files:

NOTE

In the actual file names, <device> is the name of the microcontroller device; <device core> represents the device processor family.

The file *RTX_<core>.lib* is the library with RTOS functions.

The configuration file *RTX_Conf_CM.c* for defining thread options, timer configurations, and RTX kernel settings.

The header file *cmsis_os.h* exposes the RTX functionality to the user application.

The function *main()* is executed as a thread.

Once these files are part of the project, developers can start using the CMSIS-RTOS RTX functions. The code example shows the use of CMSIS-RTOS RTX functions:

Example of using CMSIS-RTOS RTX functions

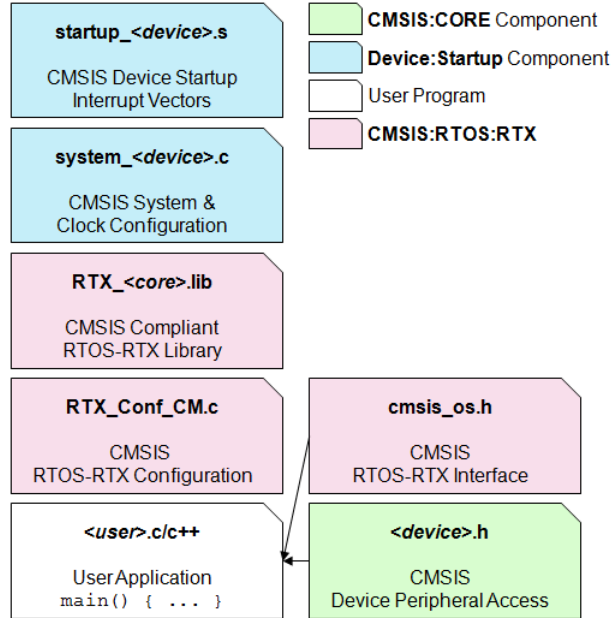
```
#include "cmsis_os.h"           // CMSIS RTOS header file

void job1 (void const *argument) {    // Function 'job1'
    // execute some code
    osDelay (10);                  // Delay execution for 10ms
}

osThreadDef(job1, osPriorityLow, 1, 0); // Define job1 as thread

int main (void) {
    osKernelInitialize ();          // Initialize RTOS kernel
    // setup and initialize peripherals
    osThreadCreate (osThread(job1), NULL); // Create the thread

    osKernelStart ();               // Start kernel & job1 thread
}
```



Header File `cmsis_os.h`

The file `cmsis_os.h` is a template header file for the CMSIS-RTOS RTX and contains:

CMSIS-RTOS API function definitions.

Definitions for parameters and return types.

Status and priority values used by CMSIS-RTOS API functions.

Macros for defining threads and other kernel objects such as mutex, semaphores, or memory pools.

All definitions are prefixed with **os** to give a unique name space for the CMSIS-RTOS functions. Definitions that are prefixed **os_** are not be used in the application code but are local to this header file. All definitions and functions that belong to a module are grouped and have a common prefix, for example, **osThread** for threads.

Define and Reference Object Definitions

With the **#define osObjectsExternal**, objects are defined as external symbols. This allows creating a consistent header file for the entire project as shown below:

Example of a header file: `osObjects.h`

```
#include "cmsis_os.h"                // CMSIS RTOS header

extern void thread_1 (void const *argument); // Function prototype
osThreadDef (thread_1, osPriorityLow, 1, 100); // Thread definition

osPoolDef (MyPool, 10, long);        // Pool definition
```

This header file, called `osObjects.h`, defines all objects when included in a C/C++ source file. When **#define osObjectsExternal** is present before the header file inclusion, the objects are defined as external symbols. Thus, a single consistent header file can be used throughout the entire project.

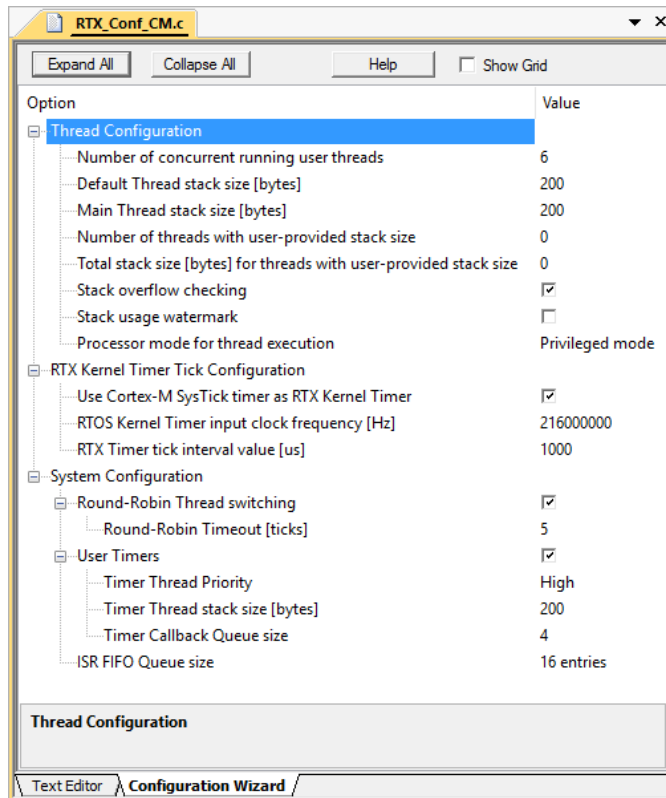
Consistent header file usage in a C file

```
#define osObjectExternal // Objects defined as external symbols
#include "osObjects.h"   // Reference to the CMSIS-RTOS objects
```

For details, refer to the online documentation www.keil.com/cmsis/rtos, section **Header File Template: `cmsis_os.h`**.

CMSIS-RTOS RTX Configuration

The file *RTX_Conf_CM.c* contains the configuration parameters of the CMSIS-RTOS RTX. A copy of this file is part of every project using the RTX component.



You can set parameters for the thread stack, configure the Tick Timer, set Round-Robin time slice, and define user timer behaviour for threads.

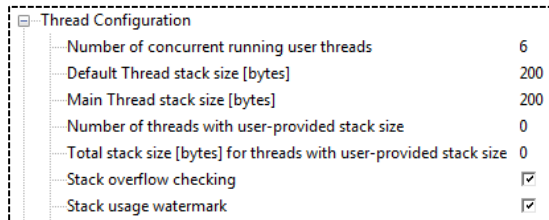
For more information about configuration options, open the RTX documentation from the **Manage Run-Time Environment** window. The section **Configuration of CMSIS-RTOS RTX** describes all available settings. The following highlights the most important settings that need adaptation in your application.

Thread Stack Configuration

Threads are defined in the code with the function *osThreadDef()*. The parameter *stacksz* specifies the stack requirement of a thread and has an impact on the method for allocating stack. CMSIS-RTOS RTX offers two methods for allocating stack requirements in the file *RTX_Conf_CM.c*:

Using a fixed memory pool: if the parameter *stacksz* is 0, then the value specified for **Default Thread stack size [bytes]** sets the stack size for the thread function.

Defining a user space: if *stacksz* is not 0, then the thread stack is allocated from a user space. The total size of this user space is specified by **Total stack size [bytes] for threads with user-provided stack size**.



Thread Configuration	
Number of concurrent running user threads	6
Default Thread stack size [bytes]	200
Main Thread stack size [bytes]	200
Number of threads with user-provided stack size	0
Total stack size [bytes] for threads with user-provided stack size	0
Stack overflow checking	<input checked="" type="checkbox"/>
Stack usage watermark	<input checked="" type="checkbox"/>

Number of concurrent running threads specifies the maximum number of threads that allocate the stack from the fixed size memory pool.

Default Thread stack size [bytes] specifies the stack size (in words) for threads defined without a user-provided stack.

Main Thread stack size [bytes] is the stack requirement for the *main()* function.

Number of threads with user-provided stack size specifies the number of threads defined with a specific stack size.

Total stack size [bytes] for threads with user-provided stack size is the combined requirement (in words) of all threads defined with a specific stack size.

Stack overflow checking enables stack overflow check at a thread switch. Enabling this option slightly increases the execution time of a thread switch.

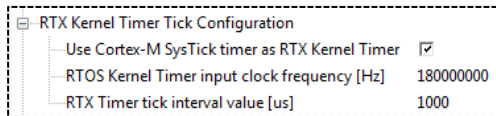
Stack usage watermark initializes the thread stack with a watermark pattern at the time of the thread creation. This enables monitoring of the stack usage for each thread (not only at the time of a thread switch) and helps to find stack overflow problems within a thread. Enabling this option increases significantly the execution time of *osThreadCreate()*.

NOTE

Consider these settings carefully. If you do not allocate enough memory or you do not specify enough threads, your application will not work.

RTX Kernel Timer Tick Configuration

CMSIS-RTOS RTX functions provide delays in units of milliseconds derived from the **Timer tick value**. We recommend configuring the Timer tick value to generate 1-millisecond intervals. Configuring a longer interval may reduce energy consumption, but has an impact on the granularity of the timeouts.




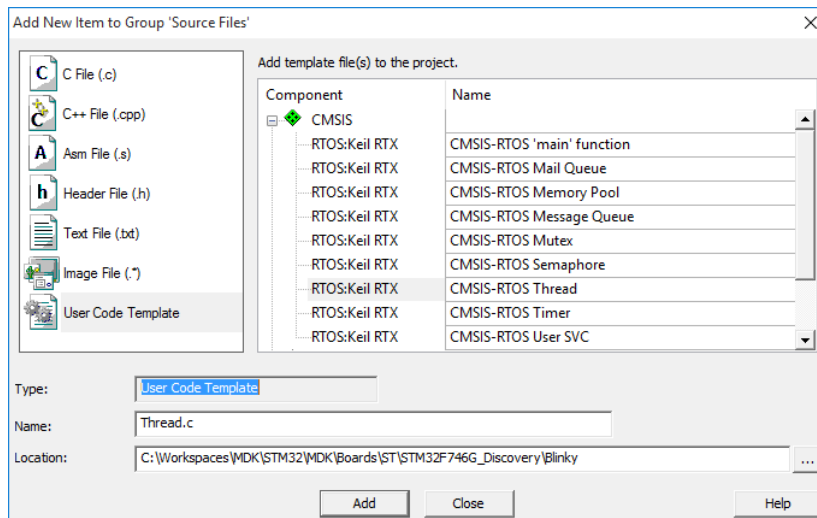
It is good practise to enable **Use Cortex-M SysTick timer as RTX Kernel Timer**. This selects the built-in SysTick timer with the processor clock as the clock source. In this case, the **RTOS Kernel Timer input clock frequency** should be **identical** to the CMSIS variable *SystemCoreClock* of the startup file *system_<device>.c*.

For details, refer to the online documentation section **Configuration of CMSIS-RTOS RTX – Tick Timer Configuration**.

CMSIS-RTOS User Code Templates

MDK provides user code templates you can use to create C source code for the application.

 In the **Project** window, right click a group, select **Add New Item to Group**, choose **User Code Template**, select any template and click **Add**.



CMSIS-RTOS RTX API Functions

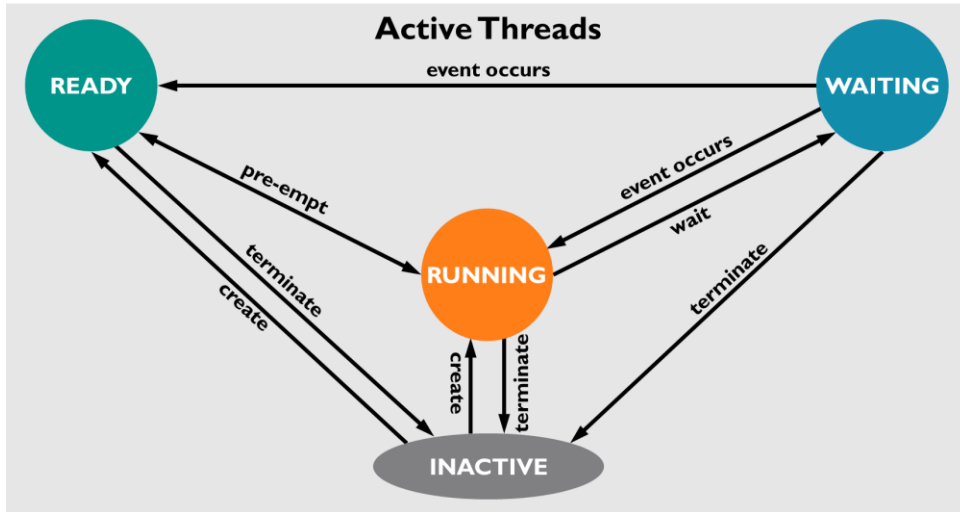
The table below lists the various API function categories that are available with the CMSIS-RTOS RTX.

API Category	Description
Thread Management	Define, create, and control thread functions.
Timer Management	Create and control timer and callback functions.
Signal Management	Control or wait for signal flags.
Mutex Management	Synchronize thread execution with a Mutex.
Semaphore Management	Control access to shared resources.
Memory Pool Management	Define and manage fixed-size memory pools
Message Queue Management	Control, send, receive, or wait for messages.
Mail Queue Management	Control, send, receive, or wait for mail.

TIP: The CMSIS-RTOS RTX tutorial available at www.keil.com/pack/doc/CMSIS/RTOS/html/index.html explains the usage of the API functions.

Thread Management

The thread management functions allow you to define, create, and control your own thread functions in the system. The function *main()* is a special thread function that is started at system initialization and has the initial priority *osPriorityNormal*.



CMSIS-RTOS RTX assumes that threads are scheduled as shown in the figure above. Thread states change as described below:

A thread is created using the function *osThreadCreate()*. This puts the thread into the **READY** or **RUNNING** state (depending on the thread priority).

CMSIS-RTOS is pre-emptive. The active thread with the highest priority becomes the **RUNNING** thread provided it is not waiting for any event. The initial priority of a thread is defined with the *osThreadDef()* but may be changed during execution using the function *osThreadSetPriority()*.

The **RUNNING** thread transfers into the **WAITING** state when it is waiting for an event.

Active threads can be terminated any time using the function *osThreadTerminate()*. Threads can also terminate by exit from the usual *forever loop* and just a *return* from the thread function. Threads that are terminated are in the **INACTIVE** state and typically do not consume any dynamic memory resources.

Single Thread Program

A standard C program starts execution with the function *main()*. For an embedded application, this function is usually an endless loop and can be thought of as a single thread that is executed continuously.

Preemptive Thread Switching

Threads with the same priority need a round robin timeout or an explicit call of the *osDelay()* function to execute other threads. In the following example, if *job2* has a higher priority than *job1*, execution of *job2* starts instantly. *job2* preempts execution of *job1* (this is a very fast task switch requiring a few ms only).

Simple RTX Program using Round-Robin Task Switching

```
#include "cmsis_os.h"

int counter1;
int counter2;

void job1 (void const *arg) {
    while (1) {                                // Loop forever
        counter1++;                            // Increment counter1
    }
}

void job2 (void const *arg) {
    while (1) {                                // Loop forever
        counter2++;                            // Increment counter2
    }
}

osThreadDef (job1, osPriorityNormal, 1, 0); // Define thread for job1
osThreadDef (job2, osPriorityNormal, 1, 0); // Define thread for job2

int main (void) {                             // main() runs as thread
    osKernelInitialize ();                    // Initialize RTX

    osThreadCreate (osThread (job1), NULL);   // Create and start job1
    osThreadCreate (osThread (job2), NULL);   // Create and start job2

    osKernelStart ();                        // Start RTX kernel

    while (1) {
        osThreadYield ();                    // Next thread
    }
}
```

Start *job2* with Higher Thread Priority

```
:
osThreadDef (osThread (job2), osPriorityAboveNormal, 1, 0);
:
```

CMSIS-RTOS System and Thread Viewer

The CMSIS-RTOS RTX Kernel has built-in support for RTOS aware debugging. During debugging, open **Debug → OS Support** and select **System and Thread Viewer**. This window shows system state information and the running threads.

System and Thread Viewer

Property

Value

System

Item	Value
Tick Timer:	1.000 mSec
Round Robin Timeout:	5.000 mSec
Default Thread Stack Size:	200
Thread Stack Overflow Check:	Yes
Thread Usage:	Available: 7, Used: 6 + os_idle_demon

Threads

ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Usage
1	osTimerThread	High	Wait_MBX				cur: 32%, max: 32% [64/200]
2	main	Normal	Wait_DLY	84			cur: 26%, max: 84% [432/512]
3	USBD_HID0_Thread	AboveNor...	Wait_OR		0x0000	0xFFFF	cur: 12%, max: 12% [64/512]
4	USBD0_Core_Thread	AboveNor...	Wait_OR		0x0000	0xFFFF	cur: 12%, max: 12% [64/512]
5	USBD_HID1_Thread	AboveNor...	Wait_OR		0x0000	0xFFFF	cur: 12%, max: 12% [64/512]
6	USBD1_Core_Thread	AboveNor...	Wait_OR		0x0000	0xFFFF	cur: 12%, max: 12% [64/512]
255	os_idle_demon	None	Running				

The **System** property shows general information about the RTOS configuration in the application. **Thread Usage** shows the number of available and threads and the used threads that are currently active.

The **Threads** property shows details about thread execution of the application. It shows for each thread information about priority, execution state and stack usage.

If the option **Stack usage watermark** is enabled for **Thread Configuration** in the file *RTX_Conf_CM.c*, the field **Stack Usage** shows **cur:** and **max:** stack load. The value **cur:** is the current stack usage at the actual program location. The value **max:** is the maximum stack load that occurred during thread execution, based on overwrites of the stack usage watermark pattern. This allows you to:

- Identify stack overflows during thread execution or
- Optimize and reduce the stack space used for threads.

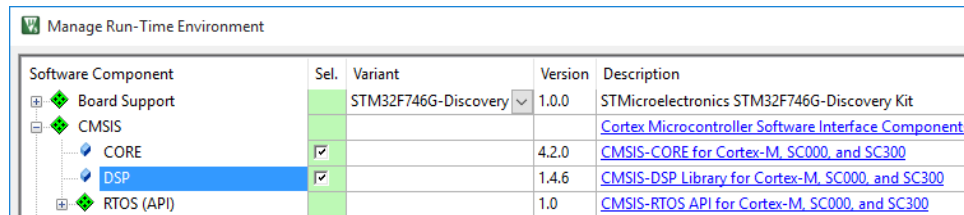
NOTE

*Using Trace, the debugger provides also a view with detailed timing information. Refer to **Event Viewer** on page 75 for more information.*

CMSIS-DSP

The CMSIS-DSP library is a suite of common digital signal processing (DSP) functions. The library is available in several variants optimized for different ARM Cortex-M processors.

When enabling the software component **::CMSIS:DSP** in the **Manage Run-Time Environment** dialog, the appropriate library for the selected device is automatically included into the project.



The code example below shows the use of CMSIS-DSP library functions.

Multiplication of two matrixes using DSP functions

```
#include "arm_math.h"           // ARM:::CMSIS:DSP

const float32_t buf_A[9] = {    // Matrix A buffer and values
    1.0, 32.0, 4.0,
    1.0, 32.0, 64.0,
    1.0, 16.0, 4.0,
};

float32_t buf_AT[9];             // Buffer for A Transpose (AT)
float32_t buf_ATmA[9];          // Buffer for (AT * A)

arm_matrix_instance_f32 A;       // Matrix A
arm_matrix_instance_f32 AT;      // Matrix AT (A transpose)
arm_matrix_instance_f32 ATmA;    // Matrix ATmA ( AT multiplied by A)

uint32_t rows = 3;              // Matrix rows
uint32_t cols = 3;              // Matrix columns

int main(void) {
    // Initialize all matrixes with rows, columns, and data array
    arm_mat_init_f32 (&A, rows, cols, (float32_t *)buf_A); // Matrix A
    arm_mat_init_f32 (&AT, rows, cols, buf_AT);             // Matrix AT
    arm_mat_init_f32 (&ATmA, rows, cols, buf_ATmA);          // Matrix ATmA


    arm_mat_trans_f32 (&A, &AT); // Calculate A Transpose (AT)
    arm_mat_mult_f32 (&AT, &A, &ATmA); // Multiply AT with A

    while (1);
}
```


For more information, refer to the CMSIS-DSP documentation on www.keil.com/cmsis/dsp.

Reference

file:///C:/Keil_v5/ARM/PACK/ARM/CMSIS/4.4.0/CMSIS/Documentation/DSP/html/modules.html



CMSIS

COMPLIANT

ARM Cortex-M Processors

Software Developer's Kit

CMSIS-DSP

Version 1.4.6

CMSIS DSP Software Library

General

Core

Driver

DSP

RTOS API

RTX

Pack

SVD

DAP

Main Page

Usage and Description

Reference

Search

CMSIS-DSP

CMSIS DSP Software Library

Change Log

Deprecated List

Reference

Data Structures

Data Fields

Reference

Here is a list of all modules:

Basic Math Functions

Fast Math Functions

Complex Math Functions

Filtering Functions

Matrix Functions

Transform Functions

Controller Functions

Statistics Functions

Support Functions

Interpolation Functions

Examples

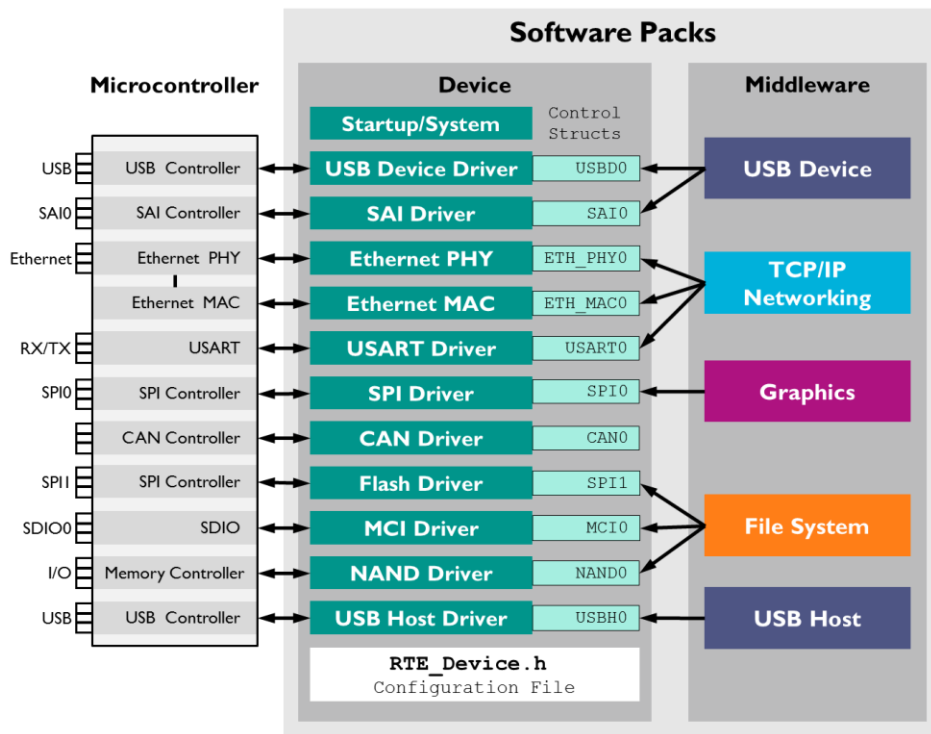
[detail level 1 2]

Generated on Fri Sep 11 2015 14:40:16 for CMSIS-DSP by ARM Ltd. All rights reserved.

CMSIS-Driver

Device-specific **CMSIS-Drivers** provide the interface between the middleware and the microcontroller peripherals. These drivers are not limited to the MDK Middleware and are useful for various other middleware stacks to utilize those peripherals.

The device-specific drivers are usually part of the Software Pack that supports the microcontroller device and comply with the CMSIS-Driver standard. The Device Database on www.keil.com/dd2 lists drivers included in the Software Pack for the device.



Middleware components usually have various configuration files that connect to these drivers. For most devices, the *RTE_Device.h* file configures the drivers to the actual pin connection of the microcontroller device.

The middleware/application code connects to a driver instance via a *control struct*. The name of this *control struct* reflects the peripheral interface of the device. Drivers for most of the communication peripherals are part of the Software Packs that provide device support.

Use traditional C source code to implement missing drivers according to the CMSIS-Driver standard.

Refer to www.keil.com/cmsis/driver for detailed information about the API interface of these CMSIS drivers.

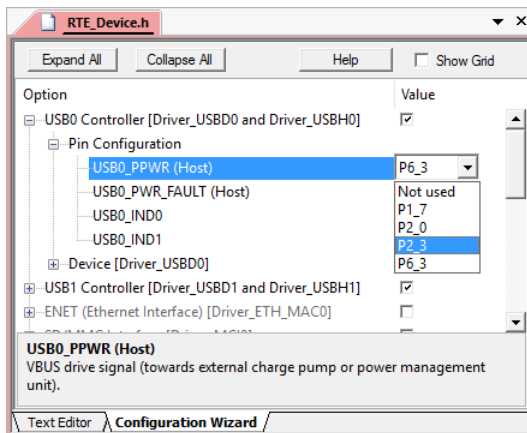
Configuration

There are multiple ways to configure a CMSIS-Driver. The classical method is using the *RTE_Device.h* file that comes with the device support.

Other devices may be configured using third party graphical configuration tools that allow the user to configure the device pin locations and the corresponding drivers. Usually, these configuration tools automatically create the required C code for import into the μ Vision project.

Using RTE_Device.h

For most devices, the *RTE_Device.h* file configures the drivers to the actual pin connection of the microcontroller device:



Using the Configuration Wizard view, you can configure the driver interfaces in a graphical mode without the need to edit manually the `#defines` in this header file.

Using STM32CubeMX

MDK supports CMSIS-Driver configuration using STM32CubeMX. This graphical software configuration tool allows you to generate C initialization code using graphical wizards for STMicroelectronics devices.

Simply select the required CMSIS-Driver in the Manage Run-Time Environment window and choose **Device:STM32Cube Framework (API):STM32CubeMX**. This will open STM32CubeMX for device and driver configuration. Once finished, generate the configuration code and import it into μ Vision.

For more information, visit the online documentation at www.keil.com/pack/doc/STM32Cube/General/html/index.html.

Validation

A Software Pack for CMSIS-Driver validation tests is available from www.keil.com/pack. It contains the source code and documentation of the CMSIS-Driver validation suite along with a required configuration file, and examples that shows the usage on various target platforms.

The CMSIS-Driver Validation Suite performs the following tests:

- Generic validation of API function calls
- Validation of configuration parameters
- Validation of communication with loopback tests
- Validation of communication parameters such as baudrate
- Validation of event functions

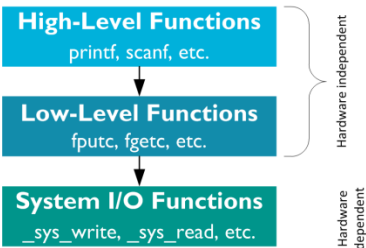
The test results can be printed to a console, output via ITM printf, or output to a memory buffer. Refer to the section **Driver Validation** in the CMSIS-Driver documentation available at www.keil.com/cmsis/driver.

Software Components

Compiler

The software component **Compiler** allows you to retarget I/O functions of the standard C run-time library. Application code frequently uses standard I/O library functions, such as *printf()*, *scanf()*, or *fgetc()* to perform input/output operations.

The structure of these functions in the standard ARM Compiler C run-time library is:



The high-level and low-level functions are not target-dependent and use the system I/O functions to interface with hardware.

The MicroLib of the ARM Compiler C run-time library interfaces with the hardware via low-level functions. The MicroLib implements a reduced set of high-level functions and therefore does not implement system I/O functions.

The software component **Compiler** retargets the I/O functions for the various standard I/O channels: File, STDERR, STDIN, STDOUT, and TTY:

