CMSIS-RTOS

Niall Cooling

Feabhas Limited

www.feabhas.com

Abstract

In early 2012 ARM announced the latest iteration of the Cortex Microcontroller Interface Standard (CMSIS), version 3.0. The major evolution of the standard is the introduction of standardized Application Programming Interface (API) for Real-Time Operating Systems (RTOS). This paper examines the CMSIS-RTOS API, looks at how different RTOSs are supported and reviews the implications for both application engineers and RTOS vendors.

All product names are trademarks or registered trademarks of their respective owners.

Introduction

Over the last decade, ARM based processors and microcontrollers have grown to become one of the most dominant architectures in the embedded computing world. From their very early days, ARM has looked to standardize, where possible, certain aspects of software development and have openly published the Base Standard Application Binary Interface (BSAPI) and the ARM Architecture Procedure Calling Standard (AAPCS). These documents detail how compilers (specifically C and C++) must use registers, the stack, and memory layout for structures (as an example).

With the release and development of the new generation of ARM cores, collectively known as the Cortex family, ARM have looked to further this approach by publishing the Cortex ™ Microcontroller Software Interface Standard (CMSIS), pronounced "cim-sis". In February of 2012, ARM announced the third version of CMSIS, which added to the definition of an abstract layer for Real-Time Operating Systems (RTOS).

ARM Cortex™ Family

The Cortex family breaks down into three major profiles:

- Cortex-A - "Application" processors that implement a virtual memory system based on a Memory Management Unit (MMU) and symmetrical Multi-core options. These are powerful enough to run higher-level Operating Systems such as Linux, Android, and some Windows variants.
- Cortex-R - "Real-Time" processors that are targeting high performance coupled with a safety model based on a Memory Protection Unit (MPU – think MMU without virtual memory).
- Cortex-M - "Microcontroller" aimed at the smaller faster system, where cost, size and power are major factors in processor choice. The M-Profile optionally supports an MPU; there are also Smartcard variants.

It is important to note that CMSIS is designed <u>only</u> to support the Cortex-M profile of microcontrollers. Within the Cortex-M profile there are currently seven variants:

- Cortex-M0
- Cortex-M0+
- Cortex-M1
- SC000
- Cortex-M3
- Cortex-M4
- SC300

with the majority of designs being based around the Cortex-M3 (as it was also the first Cortex-M IP available).

CMSIS Versions

CMSIS v1.x

The initial version of CMSIS was published in November 2008, covering the Cortex-M3 and Cortex-M0 cores. CMSIS v1.x (Figure 1 CMSIS v1.x) defined:

- Core Peripheral Access Layer
- Core Register Access
- Instruction Access
- NVIC Access Functions

- SysTick Configuration Function
- Instrumented Trace Macrocell (ITM)
- Cortex-M3 ITM Debug Access (ITM_SendChar / ITM_ReceiveChar)

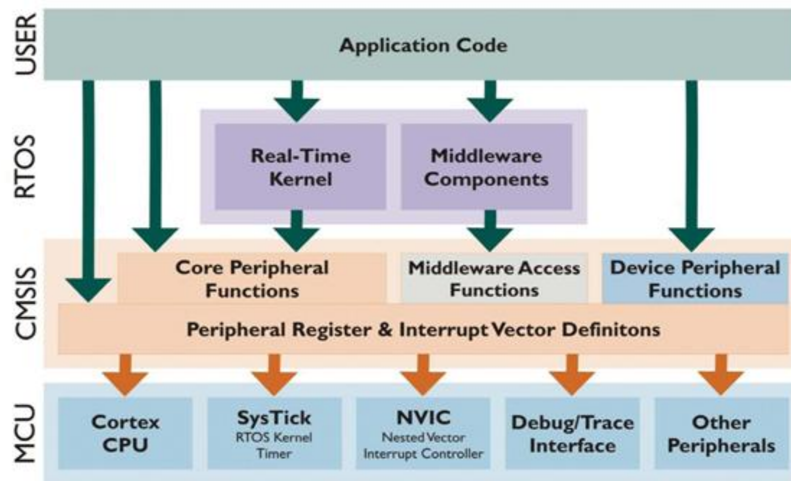

Figure 1 CMSIS v1.x

CMSIS v2.x

With the development and announcement of the Cortex-M4, CMSIS was extended to take account of the Single-Instruction Multiple-Data (SIMD) capabilities and the (optional) FPU of the M4. CMSIS v2.x added a DSP software library, optimised for the M4, to the v1.x core functions. Features supported by the DSP library include:

- Basic math functions
- Fast math functions
- Complex math functions
- Filters
- Matrix functions
- Transforms
- Motor control functions
- Statistical functions
- Support functions
- Interpolation functions

CMSIS v3.x

Version 3.0 of CMSIS was announced at Embedded World 2012 in Nuremburg, Germany. Its major content was a standardized API for Real-Time Operating Systems. The initial release supported Keil's *RTX* RTOS with the CMSIS-RTOS API under Open Source License.

CMSIS v3.x also added support for System View Description (SVD) XML files.

Quasi-concurrent programming

Unfortunately the term "RTOS" is widely misused and abused. The context here is a single program that is split in to multiple threads of execution using processor time-sharing (quasi-concurrent), known either as multi-threading or multi-tasking. However the key is that there is no MMU support, thus no virtual memory.

The term "Real-Time Operating System" or more commonly, "RTOS" can actually mean different things to different people. In this paper we shall use the following terminology:

- Kernel (RTK)
    - Scheduling (Priority - pre-emptive, Round-robin)
    - Mutual exclusion (semaphore, mutex)
- Executive (RTX)
    - Inter-task communication & synchronisation (Message Queue, Flags)
    - Dynamic memory management (fixed-block heap)
- Real-Time Operating System (RTOS)
    - File management System, e.g. Flash file system, fprintf
    - Networking, e.g. TCP/IP, CAN
    - Graphical User Interface support, e.g. OpenGL, Embedded Qt

At Feabhas, we refer to this as the "RTOS Onion", Figure 2. This distinction is important as the current CMSIS-RTOS API covers the RTK and RTE functionally, it does not include APIs for file management, networking, etc. Nevertheless, for simplicity, I'll continue to use the term RTOS in its general sense (RTK, RTX and RTOS).

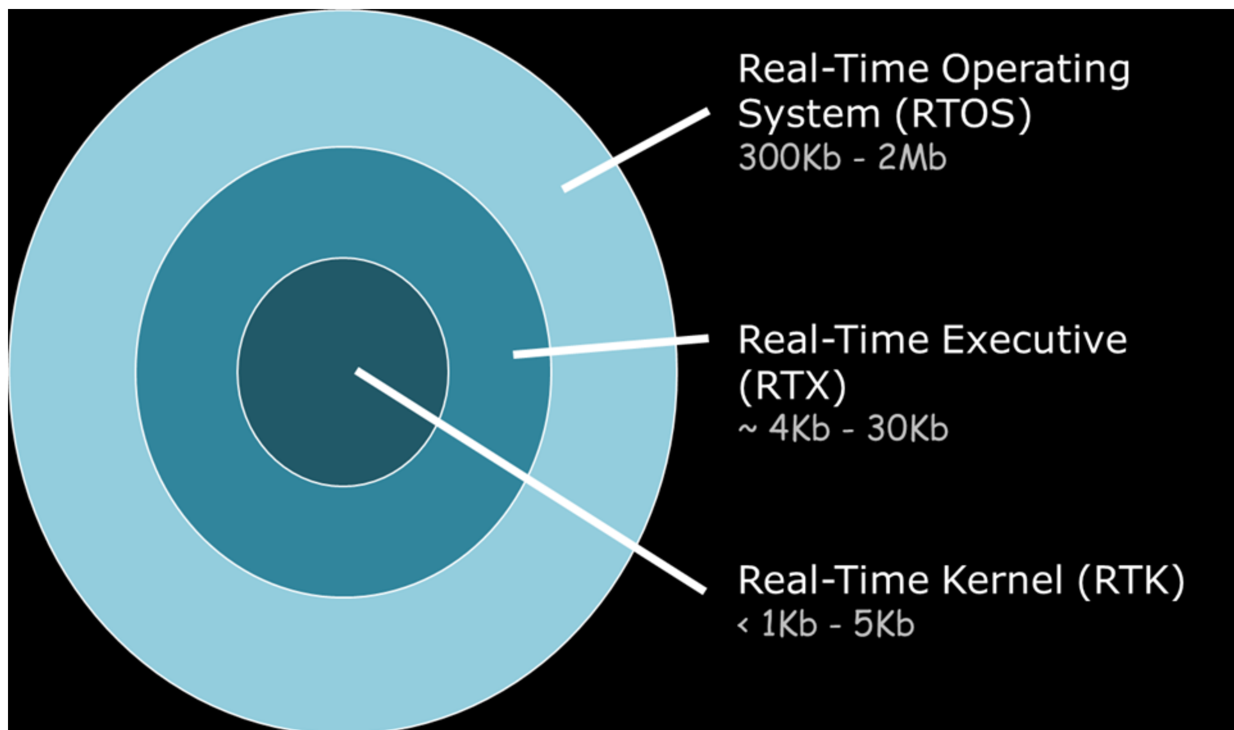

Figure 2 RTOS Onion

RTOS APIs

So why create an abstract API? If we first look at a couple of example RTOS API's we can observe that each one is propriety. For example if we take a minimal amount of code to create a task and do a short delay we can see from how different the code looks:

```
// Example 1: Keil RTX Example

OS_TID t_blinky; // Declare a task ID for blink

__task void blinky(void) {
 while(1) {
    LPC_GPIO2->MASKED_ACCESS[1] = ~LPC_GPIO2->MASKED_ACCESS[1]; // Toggle bit 0
    os_dly_wait (50);  // delay 50 clock ticks
 }
}

__task void init (void) {
  t_blinky = os_tsk_create (blinky, 1); // Create a task "blinky" with priority 1
  os_tsk_delete_self ();
}
```

```
/* Example 2: FreeRTOS Example */

void vTaskCode( void* pvParameters ) {
  for( ;; ) {
    LPC_GPIO2->MASKED_ACCESS[1] = ~LPC_GPIO2->MASKED_ACCESS[1];
    vTaskDelay(50);  // delay
  }
}

/* Function that creates a task. */
void vOtherFunction( void ) {
   static unsigned char ucParameterToPass;
   xTaskHandle xHandle;
   xTaskCreate( vTaskCode, "NAME", STACK_SIZE, &ucParameterToPass, tskIDLE_PRIORITY, &xHandle );
   vTaskDelete( xHandle );
 }
```

```
/* Example 3: Segger emBOS Example */

OS_STACKPTR int StackHP[128];          /* Task stacks */
OS_TASK TCBHP;                         /* Task-control-blocks */

/**********************************************************************/
static void TaskEx(void* pData) {
  while (1) {
    LPC_GPIO2->MASKED_ACCESS[1] = ~LPC_GPIO2->MASKED_ACCESS[1];
    OS_Delay (50);
  }
}

int main(void) {
  OS_IncDI();                       /* Initially disable interrupts  */
  OS_InitKern();                    /* initialize OS                 */
  OS_InitHW();                      /* initialize Hardware for OS    */
  OS_CREATETASK_EX(&TCBHP, "HP Task", TaskEx, 100, StackHP, (void*) 50);
  OS_Start();                       /* Start multitasking            */
  return 0;
}
```

In the above examples, the task-create functions take differing number of parameters and/or different parameter arguments. In addition, even though the task-delay functions take a simple argument, their naming convention is quite different. This naming convention, naturally, propagates through each RTOS's API.

This is an obvious problem if either a codebase has to be ported to a new RTOS, or must support different platforms with different RTOSs. It is not uncommon to find in the more mature/experience organisations that specify their own, company specific, RTOS APIs and develop their own adaption layers[1].

CMSIS v3.x Architecture

Figure 3 shows an overview of the current CMSIS architecture. It is important to understand that CMSIS-RTOS is not an RTOS itself, but purely an adaption layer. There shall always be a requirement for an actual RTOS. The main interface to the application is through the supplied file "cmsis_os.h"

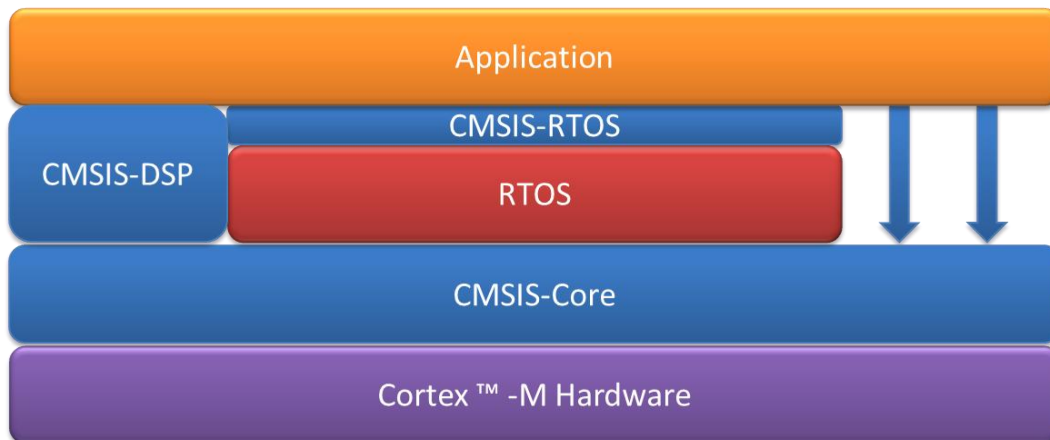

Figure 3 CMSIS v3.x Architecture

As an example, to create the example code shown previously using the CMSIS-RTOS API, the code would look similar to the following (this code is based on the *RTX* CMSIS-RTOS example header):

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

osThreadId thread1_id;

void job1 (void const *argument)  {            // thread function 'job1'
  while (1)  {
      :                                        // execute some code
    osDelay (10);                              // delay execution for 10ms
  }
}

// define job1 as thread function
osThreadDef(job1, osPriorityAboveNormal, 1, 0);

int main (void) {
  …
  thread1_id = osThreadCreate(osThread(job1), NULL);
  …
}
```

It is interesting to note that CMSIS-RTOS has chosen to use the term "thread" in preference to the more common term "task" to represent a unit of execution.

[1] In the automotive would there is an existing "standard" API called OSEK.

CMSIS-RTOS Features

The API devised by ARM and the CMSIS group covers the following functionality:

- Thread Management
  - Define and create threads
  - Thread control, e.g. delay, yield and priority management
  - Timers for callbacks
- Mutual Exclusion
  - Counting Semaphore
  - mutex
- Inter-task Communication & Synchronisation
  - Signals
  - Message Queues
  - Mail queue (mailbox)
- Memory Management
  - Memory pools

As previously mentioned, an application programming to the CMSIS-RTOS API needs to include the file "`cmsis_os.h`". This file declares the function prototypes and defines necessary structures, types and enums, etc. Breaking down the simple task create code, we can examine some of the key items.

First, in `cmsis_os.h` we can see the declaration for `osThreadCreate`, taking two arguments, a pointer to the type `osThreadDef_t` and a `void` pointer:

```
osThreadId     osThreadCreate (osThreadDef_t *thread_def, void *argument);
```

The `osThread_t` is a `typedef` for the main structure used in the task create. As different RTOS require a different number of parameters, two options are available; either use a void* or use a modifiable `typedef`; CMSIS-RTOS went with the second option. This means that each RTOS supporting the CMSIS-RTOS may require its own variant of the `cmsis_os.h`. We shall look at this shortly under the section on porting. The definition of `osThreadDef_t` is shown below:

```
/// Thread Definition structure contains startup information of a thread.
/// \note CAN BE CHANGED: \b os_thread_def is implementation specific in every CMSIS-RTOS.
typedef const struct os_thread_def  {
  os_pthread               pthread;    ///< start address of thread function
  osPriority             tpriority;    ///< initial thread priority
  uint32_t               instances;    ///< maximum number of instances of that thread
  uint32_t               stacksize;    ///< stack size requirements in bytes
} osThreadDef_t;
```

Within the `osThreadDef_t` structure are two more CMSIS-RTOS types, `os_pthread` and `osPriority`. `os_pthread` is a `typedef` for a function pointer, taking a `const void*` as a parameter with a void return type.

```
/// Entry point of a thread.
/// \note MUST REMAIN UNCHANGED: \b os_pthread shall be consistent in every CMSIS-RTOS.
typedef void (*os_pthread) (void const *argument);
```

The `osPriority` is an `enum` for the available priorities for each thread. Note that currently only 7 priority levels are supported, with a high number being the higher priority.

```
/// Priority used for thread control.
/// \note MUST REMAIN UNCHANGED: \b osPriority shall be consistent in every CMSIS-RTOS.
typedef enum  {
  osPriorityIdle          = -3,          ///< priority: idle (lowest)
  osPriorityLow           = -2,          ///< priority: low
```

```
  osPriorityBelowNormal   = -1,         ///< priority: below normal
  osPriorityNormal        =  0,         ///< priority: normal (default)
  osPriorityAboveNormal   = +1,         ///< priority: above normal
  osPriorityHigh          = +2,         ///< priority: high
  osPriorityRealtime      = +3,         ///< priority: realtime (highest)
  osPriorityError         =  0x84,      ///< system cannot determine priority or thread has…
} osPriority;
```

Finally, there are two key macro's designed to make thread definition and creation simple. osThreadDef creates an instance of the osThreadDef_t structure with a unique name and osThread wraps the structure instance to be passed to the osThreadCreate function (remember the ## is macro concatenation).

```
#define osThreadDef(name, priority, instances, stacksz)  \
osThreadDef_t os_thread_def_##name = \
{ (name), (priority), (instances), (stacksz)  };

#define osThread(name)  \
&os_thread_def_##name
```

Taking the originally CMSIS-RTOS application code:

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

void job1 (void const *argument)  {            // thread function 'job1'
  …
}

// define job1 as thread function
osThreadDef(job1, osPriorityAboveNormal, 1, 0);

int main (void) {
  …
  thread1_id = osThreadCreate(osThread(job1), NULL);
  …
}
```

we can examine the API macro expansion to see the structure object definition and the address of the object being passed as an argument:

```
void job1 (void const *argument)  {            // thread function 'job1'
  …
}

// define job1 as thread function
osThreadDef_t os_thread_def_job1 = \
{ (job1), (osPriorityAboveNormal), (1), (0)  };

int main (void) {
  …
  thread1_id = osThreadCreate(&os_thread_def_job1 , NULL);
  …
}
```

Finally, looking at the CMSIS-RTOS adaption layer example provided with CMSIS v3.x, we can trace the code through to osThreadCreate to a call to svcThreadCreate[2] (in file rt_CMSIS.c):

```
// Thread Public API
/// Create a thread and add it to Active Threads and set it to state READY
osThreadId osThreadCreate (const osThreadDef_t *thread_def, void *argument) {
  if (__get_IPSR() != 0) return NULL;          // Not allowed in ISR
  if (((__get_CONTROL() & 1) == 0) && (os_running == 0)) {
    // Privileged and not running
    return   svcThreadCreate(thread_def, argument);
  } else {
    return __svcThreadCreate(thread_def, argument);
  }
}
```

[2] The two variants svc and __svc manage the Cortex-M protection model, but __svc ultimately calls svc

Which finally calls the *RTX* API rt_tsk_create (this is the same as os_tsk_create seen earlier):

```
// Thread Service Calls
/// Create a thread and add it to Active Threads and set it to state READY
osThreadId svcThreadCreate (const osThreadDef_t *thread_def, void *argument) {
  P_TCB  ptcb;
…
  tsk = rt_tsk_create(                                       // Create task
    (FUNCP)thread_def->pthread,                      // Task function pointer
    (thread_def->tpriority-osPriorityIdle+1) |       // Task priority
    (thread_def->stacksize << 8),                    // Task stack size in bytes
    stk,                                                          // Pointer to task's stack
    argument                                                 // Argument to the task
  );
…
}
```

CMSIS v3.x Adaption

The downloadable version CMSIS-RTOS, by default, supports ARM/Keil's RTOS *RTX*. However, what if we would like to use a different RTOS but continue to use CMSIS-RTOS? For an RTOS to support the CMSIS-RTOS API two files need adapting; `cmsis_os.h` and `cmsis_os.c`. As an example, we will look at calling the *FreeRTOS* API (this is based on work by the FiFi-SDR Project). To create a CMSIS-RTOS thread, the adaption layer needs to map the `osThreadCreate` function call to the *FreeRTOS* `xTaskCreate` function call (Figure 4).

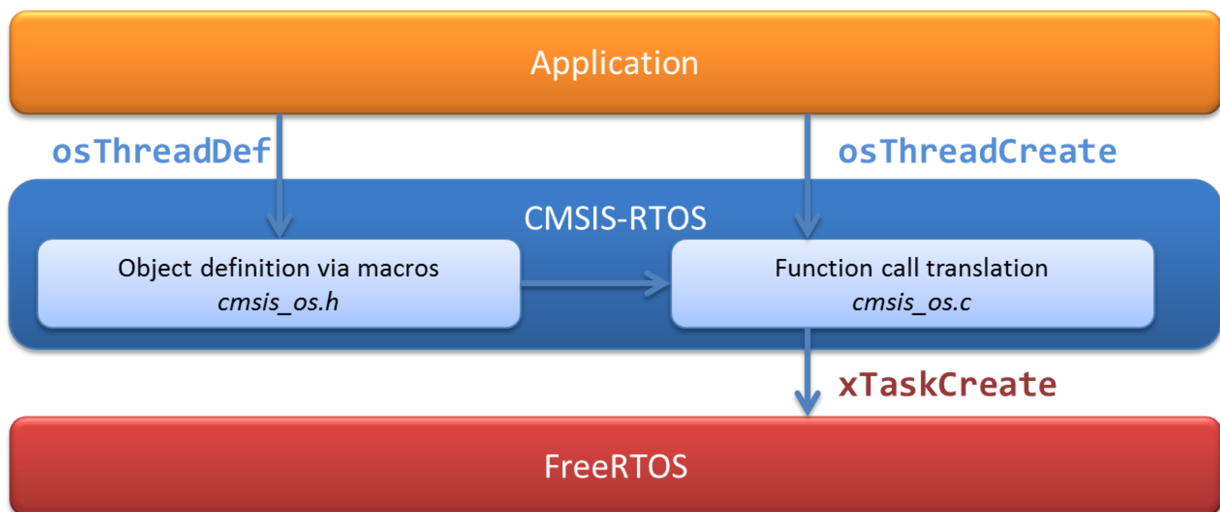

Figure 4 CMSIS-RTOS Adaption

The prototype for xTaskCreate is:

```
portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode,
                           const signed portCHAR * const pcName,
                           unsigned portSHORT usStackDepth,
                           void *pvParameters,
                           unsigned portBASE_TYPE uxPriority,
                           xTaskHandle *pvCreatedTask
                         );
```

Figure 5 xTaskCreate Function Prototype

A couple of key changes are required compared to the *RTX* adaption:

- *FreeRTOS* takes as a parameter (pcName) as task name as a C Null-Terminated Byte String (NTBS)[3].
- The stack size for *FreeRTOS* is in 32-bit words, whereas `osThreadDef_t stacksize` is in bytes; also the stack size must not be smaller than the *FreeRTOS* constant `configMINIMAL_STACK_SIZE`.
- The priorities for *FreeRTOS* cannot be a negative number (`osPriorityIdle` is -3), with 0 (zero) being the lowest priority for *FreeRTOS*.

Assuming we wanted to support task naming with *FreeRTOS*, then first `cmsis_os.h` needs copying and modifying. To allow a task name to be supplied as a parameter, we modify the `osThreadDef_t` to include a `const char*` element and modify the macro `osThreadDef` to automatically create the thread name from the thread function name (using the macro expansion #).

```
/// Thread Definition structure contains startup information of a thread.
/// \note CAN BE CHANGED: \b os_thread_def is implementation specific in every CMSIS-RTOS.
typedef const struct os_thread_def  {
  const char *           name;
  os_pthread             pthread;    ///< start address of thread function
  osPriority             tpriority;    ///< initial thread priority
  uint32_t               instances;    ///< maximum number of instances of that thread
  uint32_t               stacksize;    ///< stack size requirements in bytes
} osThreadDef_t;

#define osThreadDef(name, priority, instances, stacksz)  \
osThreadDef_t os_thread_def_##name = \
{ (#name), (name), (priority), (instances), (stacksz)  };
```

In `cmsis_os.c` the CMSIS-RTOS attributes are now mapped to the *FreeRTOS* attributes. The stack size is converted from bytes to words; the extra `osThreadDef_t` attribute is passed as the task name, and a helper function converts CMSIS-RTOS priorities to *FreeRTOS* priorities.

```
static unsigned portBASE_TYPE makeFreeRTOSPriority (osPriority priority);

/// Create a thread and add it to Active Threads and set it to state READY.
/// \param[in]     thread_def    thread definition referenced with \ref osThread.
/// \param[in]     argument      pointer that is passed to the thread function as start argument.
/// \return thread ID for reference by other functions or NULL in case of error.
/// \note MUST REMAIN UNCHANGED: \b osThreadCreate shall be consistent in every CMSIS-RTOS.
osThreadId osThreadCreate (osThreadDef_t *thread_def, void *argument)
{
    xTaskHandle handle;
    uint32_t stackSize;

    stackSize = thread_def->stacksize ? thread_def->stacksize / 4 : configMINIMAL_STACK_SIZE;

    xTaskCreate((pdTASK_CODE)thread_def->pthread,
                (const signed portCHAR *)thread_def->name,
                stackSize,
                argument,
                makeFreeRTOSPriority(thread_def->tpriority),
                &handle);

    return handle;
}

/* Convert from CMSIS type osPriority to FreeRTOS priority number */
static unsigned portBASE_TYPE makeFreeRTOSPriority (osPriority priority)
{
    unsigned portBASE_TYPE fpriority = tskIDLE_PRIORITY;

    if (priority != osPriorityError) {
        fpriority += (priority - osPriorityIdle);
    }

    return fpriority;
}
```

[3] FreeRTOS doesn't actually make any use of this, it is included purely as a debugging aid

Further ports of the CMSIS-RTOS are appearing, for example:

- *Abassi RTOS* - a commercial RTOS from Code Time Technologies, Ottawa, Ontario, Canada
- *RT-Thread* - an open source real-time operating system developed by the RT-Thread Studio based in China

Code Time Technologies (Abssi RTOS) have published the code size requirements when using CMSIS, as shown in the table below. They attribute much of the code size due to the requirements of the CMSIS-RTOS API error return codes, which need converting and verifying in the CMSIS layer rather than the native RTOS API.

Compiler / Tools	Verison	Optimization	Code Size
Code Composer Studio	5.1.1.00031	-O 3 -mf 0	< 1475 bytes
GCC (Ride 7)	7.30.10.0159	-Os	< 1450 bytes
IAR Embedded Workbench	6.30.7.3447	Level High / Size	< 1400 bytes
Keil uVision	4.50.0.0	Level 3 (-O3)	< 1400 bytes

There are three major issues to be addressed when adapting the CMSIS-RTOS layer to support another RTOS:

1) Features that CMSIS-RTOS API defines but the RTOS doesn't support
2) Features that the RTOS supports but CMSIS-RTOS API doesn't define
3) Behaviour defined for CMSIS-RTOS API differs from RTOS API behaviour

Item (1) is addressed through a set of #define in cmsis_os.h , however, it is not clear what the policy is regarding items (2) and (3).

Some Reactions

In terms of CMSIS-RTOS, it is still relatively early days; so whether large scale adoption of CMSIS-RTOS will happen is yet to be clear (there are currently nearly 30 RTOSs listed on the ARM website supporting Cortex-M). An interesting view was given by Richard Barry, creator of *FreeRTOS*, in the EE Times "Who wins when Cortex-M adds RTOS?"

The creator of ChibiOS/RT RTOS, Giovanni Di Sirio, is stronger in his viewpoint:

I must say that I don't particularly like the CMSIS RTOS API, it requires things that in ChibiOS I intentionally left out and that would hurt performance and/or safety, others things are simply too limiting.

Summary

The development of CMSIS-RTOS opens up lots of possibilities.

First, and foremost, it allows application programmers to more readily port code from one RTOS to another (possibly from a royalty based model to a royalty free model).

Second, for a company such as Feabhas, involved in teaching the principles of concurrency and Real-Time Operating Systems, being able to reference a standardised API (as opposed to either proprietary or POSIX based ones) allows us to concentrate on the principles rather than the API specifics.

Finally, and probably the most significant, is easier library support. Complex libraries such as networking and USB stacks have significant dependency on an RTOS API for managing interrupts and delays. Being able to standardize this dependency means once a protocol stack has been ported to CMSIS-RTOS, it should, in general, be a simpler job to support the same library across other CMSIS-RTOS implementations.

Nevertheless, some people have expressed concerns that we end up with a “lowest common denominator” with respect to the API and the cost of memory and performance isn’t appropriate when using an adaption layer for a small microcontroller.

It is probably too early to tell whether CMSIS-RTOS will get widespread support; ultimately it will be lead by customer demand.