CHAPTER

Using Embedded Operating Systems

19

CHAPTER OUTLINE

19.1 Introduction to embedded OSs

19.1.1 What are embedded OSs?

In Chapter 10 we covered various features of the Cortex®-M3 and Cortex-M4 processors that support the operations of embedded OSs. In general, an embedded OS can be anything from a simple task scheduler to a fully-featured OS like Linux.

The Definitive Guide to ARM® Cortex®-M3 and Cortex-M4 Processors. http://dx.doi.org/10.1016/B978-0-12-408082-9.00019-1

Currently there are more than 30 embedded OSs available for Cortex-M processors. Most of these are simple OSs that enable multi-tasking, but some of them are application platforms that also provide additional software support such as a communication protocol stack (e.g., TCP/IP), a file system (e.g., FAT), or even a graphical user interface.

Many embedded OSs are RealTime Operating Systems (RTOS). This means that when a certain event occurs, it can trigger a corresponding task and that this must happen within a certain timeframe. An RTOS typically has a small memory footprint and provides very fast context switching (the time required to switch from one task to another).

Unlike OSs for personal computers or mobile computing devices (e.g., tablets), most embedded OS do not have any user interface, although user interface components (e.g., a GUI) can be added as application tasks running on the system. Also, Cortex-M processors cannot support fully-featured OSs like Linux or Windows, which require virtual memory system support.

MEMORY MANAGEMENT UNIT (MMU) AND MEMORY PROTECTION UNIT (MPU)

In application processors such as the Cortex-A processor family, the Memory Management Unit, "MMU," enables dynamic remapping of flat virtual address spaces seen by each process into physical address spaces on the system. Managing virtual memory can introduce large delays because address mapping information needs to be located and transferred from the memory (page table) to a hardware in the MMU (called Translation Lookaside Buffer, or TLB). As a result, operating systems that use virtual memory cannot guarantee real time responsiveness.

MPUs on Cortex-M processors only provide memory protection, and do not have the memory address translation requirement and therefore are suitable for real-time applications.

There is a special version of Linux called μCLinux that does not require an MMU and can work on those Cortex-M devices which have sufficient memory resources. Since a μCLinux system typically requires at least 2MB of SRAM, it is less popular in low-cost embedded systems because of the memory cost.

19.1.2 When to use an embedded OS

An embedded OS divides the available CPU processing time into a number of time slots and carries out different tasks in different time slots. Because the switching between time slots may happen hundreds of times per second, or more, it appears to the user that the processor executes several tasks in parallel.

Many applications do not require an embedded OS at all. The key benefit of using an embedded OS is to provide a scalable way of enabling several concurrent tasks to run in parallel. If the tasks are all fairly short and don't overlap each other most of the time, you can simply use an interrupt-driven arrangement to support multiple tasks.

There are a number of factors to consider when deciding whether to use an embedded OS or not:

- An embedded OS requires extra memory overhead. For example, it could take anything from 5KB of program memory space to over 100KB, depending on the features available in the OS.
- An embedded OS requires execution time overhead. For example, some processing time is required for context switching as well as task scheduling. Usually the execution time overhead is very small.
- Some of the embedded OS require license fees and/or royalty fees. Many others are free.
- Some embedded OS can only work with certain microcontroller devices, or can be toolchain-specific. If portability of the software code is important then you need to select an embedded OS which is supported on multiple platforms.

In general, as software code gets more complex, use of an embedded OS can make handling of multiple tasks much easier. Also, some embedded OS have additional safety features (e.g., stack space checking, MPU support), which can enhance the reliability of a system.

19.1.3 Role of CMSIS-RTOS

CMSIS-RTOS is an API specification. The CMSIS-RTOS itself is not a product but companies can build an RTOS based on CMSIS-RTOS. In Chapter 2 we gave an overview of the Cortex®-M Software Interface Standard (CMSIS). One of the projects within CMSIS is the CMSIS-RTOS. CMSIS-RTOS is an extension of existing RTOS designs to allow middleware to be designed in a way that can work with multiple RTOS products.

Some of the middleware products are quite complex and need to utilize task scheduling features in OSs to work. For example, a TCP/IP stack might run as a task inside a multi-tasking system and might need to spawn out additional child tasks when certain service requests are received. Traditionally these middleware (e.g., lightweight IP, lwIP) can include an OS emulation layer (Figure 19.1) that a software integrator needs to port when using a different OS.

The work of porting the OS emulation layer creates additional work for software developers, and sometimes the middleware vendors, and can increase project risks as the porting might not be straightforward.

CMSIS-RTOS was created to solve this issue. It can be implemented as an additional set of APIs or a wrapper for existing OS APIs. Since the API is standardized, middleware can be developed based on it and the product should, in theory, be able to work with any embedded OS that supports CMSIS-RTOS (Figure 19.2).

The RTOS products can still have their own native API interface, and application codes can still use those directly for additional features or for higher performance. This is good news for application developers because it saves a lot of time in porting

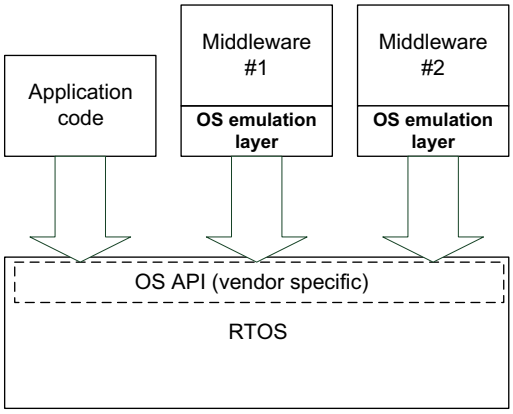

FIGURE 19.1

The need for OS emulation layer for middleware components

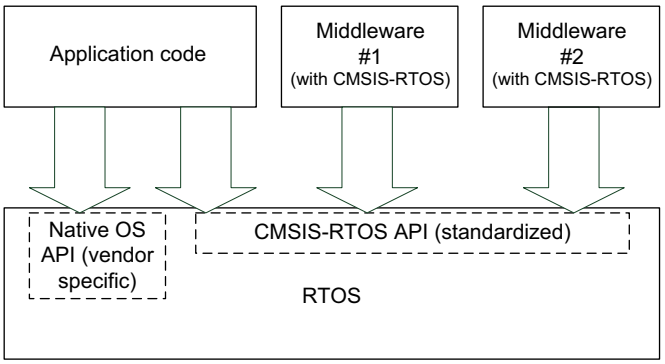

FIGURE 19.2

CMSIS-RTOS avoids the needs for OS emulation layer for each middleware component

middleware and reduces project risks. It is also good news for middleware vendors because it allows their products to work with more available OSs.

The CMSIS-RTOS also benefits RTOS vendors: As the amount of middleware that works with CMSIS-RTOS increases, having CMSIS-RTOS support in an embedded OS enables the OS product to work with more middleware. Also, as software in embedded systems increases in complexity and time-to-market becomes more important, porting of OS emulation layers for middleware will no longer be feasible for some projects because of the extra time needed and the associated project risk. CMSIS-RTOS enables RTOS products to reach these markets, which previous could only be covered by a few software platform solutions.

19.2 Keil™ RTX Real-Time Kernel

19.2.1 About RTX

The Keil™ RTX Real-Time Kernel is a royalty-free RTOS targeted at microcontroller applications. In older versions of Keil MDK-ARM, RTX is available as a precompiled library that is fully functional. In newer versions (mid-2012) of Keil MDK-ARM, the source code of the RTX kernel is also included in the installation.

The precompiled library is typically located in C:\Keil\ARM\RV31\LIB, and source code is typically located in C:\Keil\ARM\RL\RTX\SRC. In the C:\Keil\ARM\Boards directory, you can also find RTX examples for a number of Cortex®-M microcontroller boards.

Please also note that Keil RTX is now released under a simple, open source BSD license, so you can reuse and distribute RTX source code under the conditions described in the license document in the Keil MDK-ARM installation (C:\Keil\ARM\Hlp\license.rtf).

The Keil RTX kernel can be used as standalone RTOS or used with Keil Real-Time Library (RL-ARM, Figure 19.3), and can also work with third-party software products such as communication protocol stacks, data processing codecs, and other middleware.

19.2.2 Features overview

The RTX kernel is supported on all Cortex®-M processors in addition to traditional ARM® processors such as ARM7™ and ARM9™. It has the following features:

- Flexible scheduler: supporting pre-emptive, round-robin, and collaborative scheduling schemes
- Supports mailboxes, events (up to 16 per task), semaphores, mutex, and timers
- Unlimited number of defined tasks, with maximum of 250 active tasks at a time
- Up to 255 task priority levels
- Support for multi-threading and thread-safe operations
- Kernel-aware debug support in Keil™ MDK
- Fast context switching time

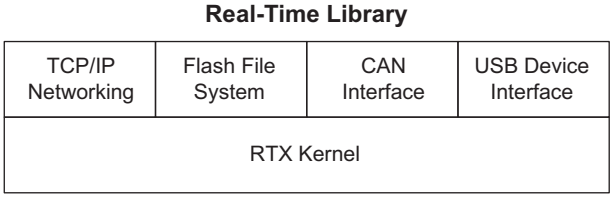

FIGURE 19.3

The RL-ARM product

- Small memory footprint (less than 4Kbytes for Cortex-M version, less than 5Kbytes for ARM7/ARM9)

In addition, the Cortex-M version of RTX kernel has the following features:

- SysTick timer support
- No interrupt lockout in Cortex-M versions (interrupt is not disabled by OS at any time)
- Since RTX has a very small memory footprint, it can be used even with Cortex-M microcontroller devices that have small memory capacity.

19.2.3 RTX and CMSIS-RTOS

In 2012, Keil™ released a trial version of the CMSIS-RTOS implementation for RTX. This design is to be finalized in 2013 so you can use RTX with CMSIS-RTOS API. At the time of writing (January, 2013), the CMSIS-RTOS RTX work has not been completed and the RTX included in the Keil MDK installation is still the old version, which does not have CMSIS-RTOS support. The RTX example code in the Keil MDK installation is also based on the previous proprietary APIs. At the moment, to use CMSIS-RTOS with RTX, you need to download the RTX implementation for CMSIS-RTOS from the Keil website separately at https://www.keil.com/demo/eval/rtx.htm.

Because the RTX source code in use at the time of writing is not the final version, please note that there is a small chance the CMSIS-RTOS RTX examples described in this chapter will need adjustment for the final version of CMSIS-RTOS RTX.

The CMSIS-RTOS package contains the source code, examples, and documentation. To make it easier for users, this CMSIS-RTOS package contains precompiled versions of the CMSIS-RTOS in the form of library. Table 19.1 listed the directories in RTX CMSIS-RTOS package from Keil® website.

Table 19.1 Directory Structure in Current CMSIS-RTOS Package

Directory	Content
Boards	CMSIS-RTOS RTX example projects for several evaluation boards. These examples are typically provided for several compilers.
Doc	Documentation for CMSIS-RTOS RTX.
Examples	Generic examples that show several features of CMSIS-RTOS RTX. These examples are typically provided for several compilers.
INC	The include files for CMSIS-RTOS RTX. cmsis_os.h is the central include file for user applications.
LIB	CMSIS-RTOS RTX Library files for ARMCC, GCC, and IAR Compiler.
SRC	Source code of the CMSIS-RTOS RTX Library along with project files for ARMCC, GCC, and IAR Compiler.
Templates	Templates for creating application projects with CMSIS-RTOS RTX.

Table 19.2 CMSIS-RTOS Precompiled Libraries

Library File	Processor Configuration
LIB\ARM\RTX_CM0.lib	CMSIS-RTOS RTX Library for ARMCC Compiler, Cortex-M0 and M1, little-endian.
LIB\ARM\RTX_CM0_B.lib	CMSIS-RTOS RTX Library for ARMCC Compiler, Cortex-M0 and M1, big-endian.
LIB\ARM\RTX_CM3.lib	CMSIS-RTOS RTX Library for ARMCC Compiler, Cortex-M3 and M4 without FPU, little-endian.
LIB\ARM\RTX_CM3_B.lib	CMSIS-RTOS RTX Library for ARMCC Compiler, Cortex-M3 and M4 without FPU, big-endian.
LIB\ARM\RTX_CM4.lib	CMSIS-RTOS RTX Library for ARMCC Compiler, Cortex-M4 with FPU, little-endian.
LIB\ARM\RTX_CM4_B.lib	CMSIS-RTOS RTX Library for ARMCC Compiler, Cortex-M4 with FPU, big-endian.

Table 19.3 Additional CMSIS-RTOS Files Required for Projects

File	Processor Configuration
INC\cmsis_os.h	CMSIS-RTOS header file for application code
Examples*\RTX_Conf_CM.c	RTX Kernel System Configuration file – can be edited by user. This file is coded with special tags in comments so that the RTX parameters can be edited easily with a Configuration Wizard.
INC\RTX_CM_LIB.h	RTX Kernel System Configuration code needed by RTX_Conf_CM.c
SRC\ARM\SVC_Tables.s	An assembly file to allow you to extend the SVC services available by adding a lookup Table of SVC functions. This is optional.

The library files provided in the package include pre-compiled versions for various Cortex®-M processors, and are available in little endian and big endian versions for ARM®, gcc and IAR toolchains. For example, the library files for the ARM toolchain are shown in Table 19.2.

Alternatively, you can also use the source code directly from the SRC directory. In addition, you also need couple of additional files from the INC directory and SRC directory, as shown in Table 19.3.

19.2.4 Thread

In the CMSIS-RTOS, we use the term "thread" for each of the concurrent (parallel processing) programs. From an academic view, a task or a process could contain multiple threads. But here we will just look at a relatively simple case where each task has just one thread.

Each thread has a programmable priority level. In the RTX implementation the thread priority is an enumerated value. The CMSIS-RTOS has a number of pre-defined enumerations for thread priorities, and this is mapped into the signed numerical priority levels in the file cmsis_os.h:

```
/// 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 illegal priority
} osPriority;
```

Note that the thread priority level arrangement is completely separated from the interrupt priority.

In the RTX environment, each thread can be in one of the states shown in Table 19.4.

The thread state transition diagram is shown in Figure 19.4.

In a simple single core processor system, there can be only one thread in Running state at a time.

Unlike some other RTOSs, "main()" can be of the threads, dependent on the actual implementation of CMSIS-RTOS. If that is the case, we can create additional threads from "main()". If the "main()" thread is not needed at some stage any point,

Table 19.4 Thread States in RTX Kernel

State	Description
RUNNING	The thread is currently running.
READY	The thread is in the queue of threads which are ready to run (waiting for a time slot). When the current running thread is completed, RTX will select the next highest priority thread in the ready queue and start it.
WAITING	The thread has previously executed a function that indicate it is waiting for a delay request to complete or an event (signal/semaphore/mailbox/etc.) from another thread. It can switch from Waiting to Ready/Running (depending on task priority) when the specified event has occurred.
INACTIVE	The thread has not been started or the thread has been terminated. A terminated task can be re-created.

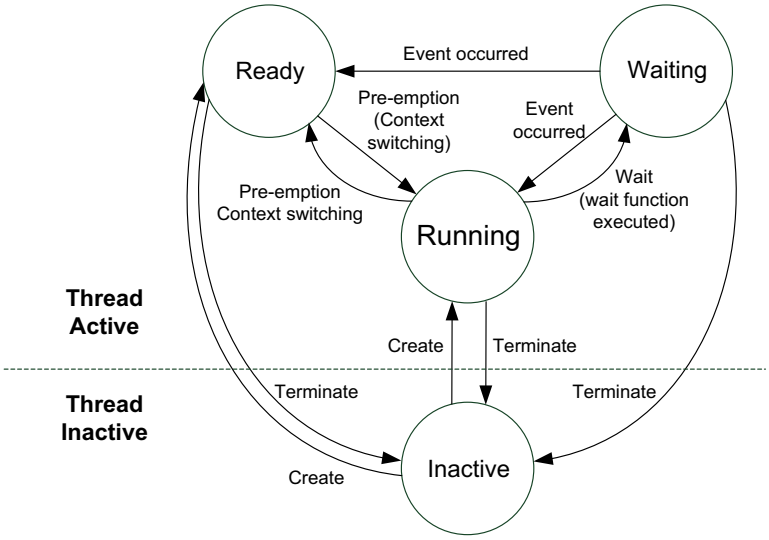

FIGURE 19.4

States of threads in CMSIS-RTOS

we can execute a wait function to put it in a waiting state, or even terminate it to prevent it from taking up execution time.

CMSIS-RTOS allows threads to execute in privileged state or unprivileged state. Please refer to the OS_RUNPRIV parameter in Table 19.6. Please note that with the current RTX implementation, if threads are configured to run in unprivileged state, "main()" will also start in unprivileged state. You can extend the SVC Handler service to support operations that require privileged state (e.g., access to NVIC or any registers in the System Control Space, SCS).

19.3 CMSIS-OS examples

19.3.1 Simple CMSIS-RTOS with two threads

The following examples are based on the Keil™ MDK-ARM development suite and CMSIS-RTOS RTX, using the STM32F4 Discovery board.

In the first example, we will look at a minimal setup with two threads: main() and a blinky thread. The threads each toggle an LED on the development board. To set up the first project, we use the precompiled version of CMSIS-RTOS RTX (library file RTX_CM4.lib) to simplify the compilation, as shown in Figure 19.5.

If you like, you can also use the source code version of the CMSIS-RTOS instead of using the precompiled library. In addition, we also need the files given in Table 19.5.

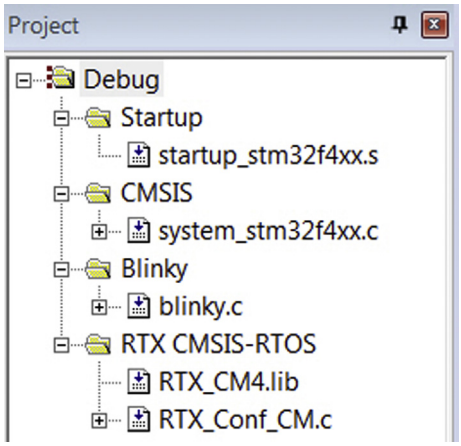

FIGURE 19.5

Project browser display with simple project

Table 19.5 Additional Files Needed in the First CMSIS-RTOS Project Example

Files	Descriptions
RTX_Conf_CM.c	RTX Kernel System Configuration file
cmsis_os.h	CMSIS-RTOS header file for application code
RTX_CM_lib.h	RTX Kernel System Configuration code needed by RTX_Conf_CM.c

We also configure the RTX kernel option in the project option, as shown in Figure 19.6. This enables us to use the OS-aware debugging features later.

Previously (Table 19.5) we mentioned that the file RTX_Config_CM.c defines some of the configurations of the RTX kernel operations. This file is configurable by users. We can either edit this file directly in the program text editor, or we can use the Configuration Wizard. This file is coded in such a way that it can be recognized by the Configuration Wizard. By clicking on the "Configuration Wizard" tab at the bottom of the editor window, we can see the Configuration Wizard as shown in Figure 19.7.

Table 19.6 shows listed a number of options in RTX_Conf_CM.c.

The actual code for the first example is very simple.

```
/* Simple CMSIS-RTOS RTX example that use two threads (including
main()) to toggle two LEDs */
#include "stm32f4xx.h"
#include <cmsis_os.h>

/* Thread IDs */
osThreadId t_blinky;  // Declare a thread ID for blink
```

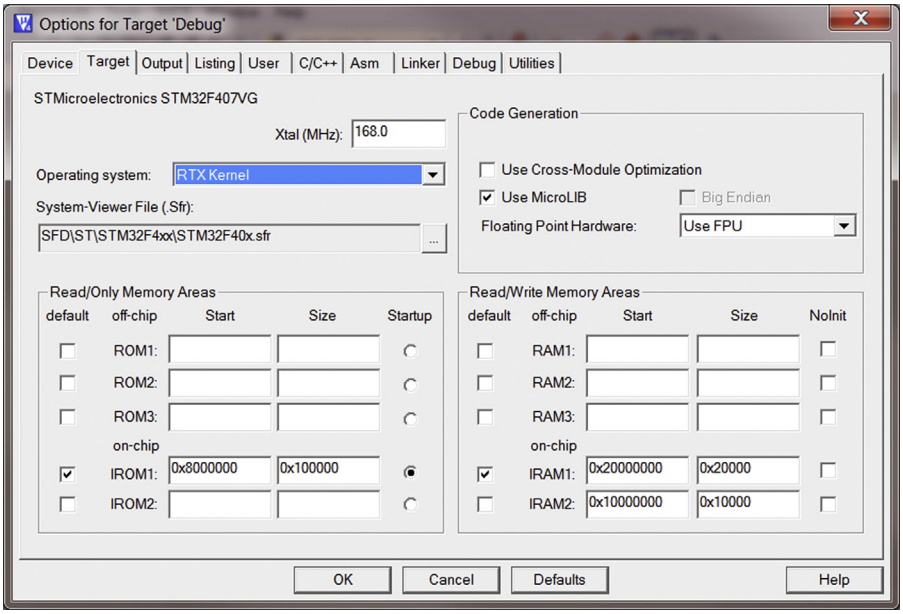

FIGURE 19.6

RTX Kernel project option

```
/* Function declaration */
void blinky(void const *argument); // Thread
void LedOutputCfg(void);           // LED output configuration

// ----------------------------------------------------------
// Blinky
// - toggle LED bit 12
// - Unprivileged Thread
void blinky(void const *argument) {
  while(1) {
    if (GPIOD->IDR & (1<<12)) {
      GPIOD->BSRRH = (1<<12); // Clear bit 12
    } else {
      GPIOD->BSRRL = (1<<12); // Set bit 12
      }
      osDelay(500);  // delay 500 msec
    }
  }
```

```
  // define blinky_1 as thread function
  osThreadDef(blinky, osPriorityNormal, 1, 0);

// ----------------------------------------------------------
// - toggle LED bit 13
// - Unprivileged Thread
int main(void)
{
 LedOutputCfg(); // Initialize LED output

 // Create a task "blinky"
 t_blinky = osThreadCreate(osThread(blinky), NULL);

 // main() itself is another thread
 while(1) {
     if (GPIOD->IDR & (1<<13)) {
     GPIOD->BSRRH = (1<<13); // Clear bit 13
     } else {
     GPIOD->BSRRL = (1<<13); // Set bit 13
     }
     osDelay(1000);  // delay 1000 msec
 }
} // end main
// ----------------------------------------------------------
void LedOutputCfg(void)
{
   // Configure LED outputs
   RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN; // Enable Port D clock
   // Set pin 12, 13, 14, 15 as general purpose output mode (pull-push)
   GPIOD->MODER |= (GPIO_MODER_MODER12_0 |
                    GPIO_MODER_MODER13_0 |
                    GPIO_MODER_MODER14_0 |
                    GPIO_MODER_MODER15_0 ) ;
   GPIOD->PUPDR = 0; // No pull up , no pull down
   return;
}
```

For each thread, there is an assoicated ID value with the data type `osThreadId`. This ID value is assigned when the thread is created and is needed for intertask communication, which will be demonstrated later. If no intertask communication is required, it is not necessary.

To create a new thread, we used the function `osThreadCreate`.

For each thread (apart from main), we also need to declare the function as a thread using osThreadDef. You can also define the priority of the thread using

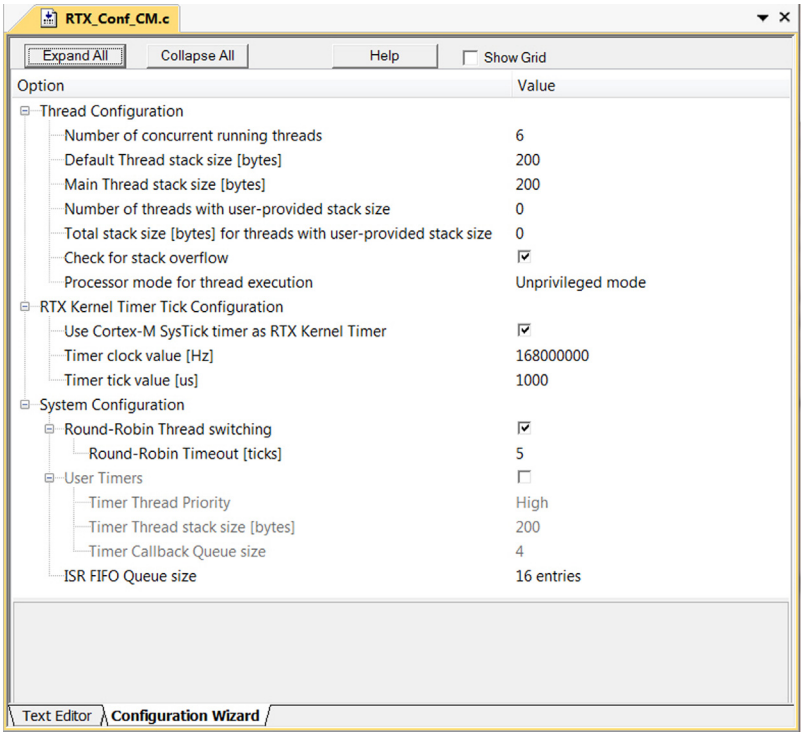

FIGURE 19.7

Configuration Wizard

`osThreadDef`. During run-time the priority of a thread can also be changed dynamically using CMSIS-RTOS API.

After setting up the project, you can then compile and test the application. The two LEDs on the development should toggle at different speeds.

In other CMSIS-RTOS implementations, it is possible that the OS kernel does not start when the processor enters the "main()" program. In such cases you will need to start the OS kernel specifically. CMSIS-RTOS provides a predefined constant called `osFeature_MainThread` to indicate whether thread execution starts with the function "main()." If this is 1, then the OS kernel starts with "main()."

For example, you can use the following code to start the OS kernel conditionally:

```
int main(void)
{
 ...
#if (osFeature_MainThread==0)
 osKernelStart(osThread(blinky), NULL); // Start OS Kernel explicity
 // not required in RTX
```

Table 19.6 CMSIS-RTOS RTX Options in RTX_Conf_CM.c

Parameter	Descriptions	Default Value
OS_TASKCNT	Number of concurrent running threads: Defines max number of threads that will run at the same time.	6
OS_STKSIZE	Default Thread stack size [bytes] <64-4096> (needs to be a multiple of 8). It is used if the "osThreadDef" statement does not specify stack size (stacksz set to 0).	200
OS_MAINSTKSIZE	Main Thread stack size [bytes] <64-4096> (needs to be a multiple of 8).	200
OS_PRIVCNT	Number of threads with user-provided stack size <0-250>	0
OS_PRIVSTKSIZE	Total combined stack size [bytes] for threads with user-provided stack size <0-4096> (needs to be a multiple of 8).	0
OS_STKCHECK	Enable check for stack overflow for threads. Note that additional code reduces the Kernel performance.	1
OS_RUNPRIV	Processor mode for thread execution: 0 = Unprivileged mode, 1 = privileged mode.	0
OS_SYSTICK	Set to 1 to use Cortex®-M SysTick timer as RTX Kernel Timer.	1
OS_CLOCK	Defines the Timer clock frequency [Hz] <1-1000000000>. Typically this is the same as the processor clock frequency if SysTick is used.	12000000 (12MHz)
OS_TICK	Defines the OS Timer tick interval [us] <1-1000000>	1000 (1 ms)
OS_ROBIN	Set to 1 to enable Round-Robin Thread switching	1
OS_ROBINTOUT	Round-Robin Timeout [ticks] <1-1000> (valid if OS_ROBIN is 1)	5
OS_TIMERS	Enables user Timers	0
OS_TIMERPRIO	Timer Thread Priority (valid if OS_TIMERS is 1) 1. Low 2. Below Normal 3. Normal 4. Above Normal 5. High 6. Real-time (highest)	5
OS_TIMERSTKSZ	Timer Thread stack size [bytes] <64-4096> (needs to be a multiple of 8).	200
OS_TIMERCBQS	Timer Callback Queue size- Number of concurrent active timer callback functions.	4
OS_FIFOSZ	ISR FIFO Queue size (4 = 4 entries. Can be 4, 8, 12, 16, 24, 32, 48, 64, 96). ISR functions store requests to this buffer when they are called from the interrupt handler.	16

```
#endif
 ...
```

or

```
int main(void)
{
 ...
 if (osFeature_MainThread==0) {
  osKernelStart(osThread(blinky), NULL); // Start OS Kernel explicity
  // not required in RTX
   }
 ...
```

The `osThread(name)` macro is used in the example for accessing a Thread definition. For example, when a function's input parameter needs to be a Thread (e.g., blinky), then we use `osThread(blinky)` to specify that the parameter is a Thread.

In this example, we also used a macro called `osThreadDef(name, priority, instances, stacksz)`. This is used to create a Thread definition with the specified function, priority level, and stack size requirements of the thread. If the stack size requirement is set to 0, the default stack size is used, as defined by OS_STKSIZE in RTX_Config_CM.c.

Table 19.7 lists some of the commonly used functions for OS kernel management and Thread management.

Some of these functions use a enumeration type called `osStatus`. The definition of osStatus is listed in Table 19.8. Most of the functions will only be able to return a subset of these enumerations.

19.3.2 Inter-thread communciation overview

In most applications with an RTOS, there will be lots of interactions between threads. Instead of using shared data and polling loops to check the status of other tasks, or passing information, we should use the inter-thread communication features provided in the OS to make the operation more efficient. Otherwise, a thread waiting for input from another thread could stay in the READY queue for a long time and this can consume a lot of processing time.

Modern RTOSs typically provide a number of methods to support communications between threads. In CMSIS-RTOS, the supported methods include:

- Signal events
- Semaphores
- Mutex
- Mailbox/message

Table 19.7 CMSIS-RTOS Functions for OS Kernel and Thread Management

	Function	Description
osThreadID	osThreadCreate(osThreadDef_t *thread_def, void *argument)	Create a thread and add it to Active Threads and set it to state READY.
osThreadID	osThreadGetId(void)	Return the thread ID of the current running thread.
osStatus	osThreadTerminate (osThreadId thread_id)	Terminate execution of a thread and remove it from Active Threads.
osStatus	osThreadSetPriority (osThreadId thread_id, osPriority priority)	Change priority of an active thread.
osPriority	osThreadGetPriority (osThreadId thread_id)	Get current priority of an active thread.
osStatus	osThreadYield (void)	Pass control to the next thread that is in state READY.
osStatus	osKernelStart (osThreadDef_t *thread_def, void *argument)	Start the RTOS Kernel and execute the specified thread.
int32_t	osKernelRunning(void)	Check if the RTOS kernel is already started. Returns 0 if the RTOS is not started. Returns 1 if started.

Table 19.8 osStatus Enumeration Definition

osStatus Enumerator	Description
osOK	Function completed; no event occurred.
osEventSignal	Function completed; signal event occurred.
osEventMessage	Function completed; message event occurred.
osEventMail	Function completed; mail event occurred.
osEventTimeout	Function completed; timeout occurred.
osErrorParameter	Parameter error: a mandatory parameter was missing or specified an incorrect object.
osErrorResource	Resource not available: a specified resource was not available.
osErrorTimeoutResource	Resource not available within given time: a specified resource was not available within the timeout period.
osErrorISR	Not allowed in ISR context: the function cannot be called from interrupt service routines.
osErrorISRRecursive	Function called multiple times from ISR with same object.
osErrorPriority	System cannot determine priority or thread has illegal priority.

Table 19.8 osStatus Enumeration Definition—*Cont'd*

osStatus Enumerator	Description
osErrorNoMemory	System is out of memory: it was impossible to allocate or reserve memory for the operation.
osErrorValue	Value of a parameter is out of range.
osErrorOS	Unspecified RTOS error: run-time error but no other error message fits.
os_status_reserved	Reserved error value to prevent from enum down-size compiler optimization.

In addition, there are additional features to support some of these communication methods such as memory pool management features, which are often used with mailboxes.

19.3.3 Signal event communication

In CMSIS-RTOS, each thread can have up to 31 signal events (depending on configuration via a macro called `osFeature_Signals` in RTX). A thread enters WAIT state when it executes the function `osSignalWait`. One of the input parameters, a 32-bit value called "signals," defines the signal events required to put the thread back to READY state. Each bit (apart from the MSB) of the "signals" parameter defines the signal events required, and if this parameter is set to 0 any signal event can put this thread back to READY state. Table 19.9 listed the CMSIS-RTOS functions for signal event communications.

The signal event functions `osSignalSet`, `osSignalClear`, and `osSignalGet` return 0x80000000 in case of incorrect parameters.

By default the "cmsis_os.h" in RTX specifies `osFeature_Signals` as 16. So it can work with 16 signal events (from 0x00000001 to 0x00008000).

Please note that signal flags used as events for waking up a thread from the WAITING state are cleared automatically. For example, in the following example, event flag 0x0001 is used to enable "main()" thread to send a signal to blinky event as shown in Figure 19.8.

```
/* Example code for simple signal event communication */

#include "stm32f4xx.h"
#include <cmsis_os.h>

/* Thread IDs */
osThreadId t_blinky; // Declare a thread ID for blink
/* Function declaration */
void blinky(void const *argument); // Thread
```

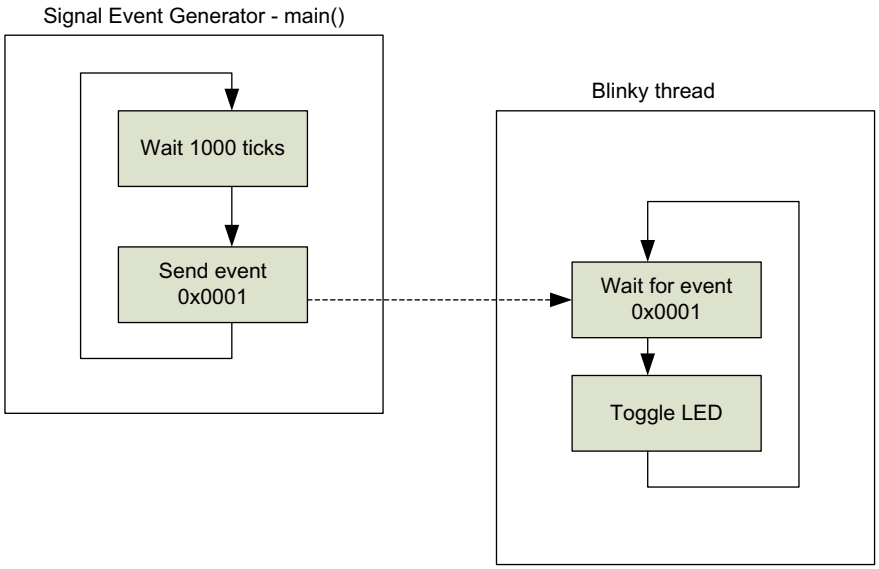

FIGURE 19.8

Simple signal event communication

Table 19.9 Signal Event Functions

	Function	Description
osEvent	osSignalWait (int32_t signals, uint32_t millisec)	Wait for one or more Signal Flags to become signaled for the current RUNNING thread. If "signals" is non-zero, all specified signal flags need to be set to return to READY state. If "signals" is zero, any signal flag can put the thread back to READY. "millisec" is the timeout value. Set to `osWaitForever` in case of no time-out, or zero to return immediately
int32_t	osSignalSet (osThreadId thread_id, int32_t signal)	Set the specified Signal Flags of an active thread.
int32_t	osSignalClear (osThreadId thread_id, int32_t signal)	Clear the specified Signal Flags of an active thread.
int32_t	osSignalGet (osThreadId thread_id)	Get Signal Flags status of an active thread.

```
void LedOutputCfg(void);           // LED output configuration

// ----------------------------------------------------------
// Blinky
// - toggle LED bit 12
// - Unprivileged Thread
void blinky(void const *argument) {
 while(1) {
   osSignalWait(0x0001, osWaitForever);
   if (GPIOD->IDR & (1<<12)) {
   GPIOD->BSRRH = (1<<12); // Clear bit 12
   } else {
   GPIOD->BSRRL = (1<<12); // Set bit 12
   }
   }
 }

 // define blinky_1 as thread function
 osThreadDef(blinky, osPriorityNormal, 1, 0);

// ----------------------------------------------------------
// - toggle LED bit 13
// - Unprivileged Thread
int main(void)
{
 LedOutputCfg(); // Initialize LED output

 // Create a task "blinky"
 t_blinky = osThreadCreate(osThread(blinky), NULL);

 // main() itself is another thread
 while(1) {
   if (GPIOD->IDR & (1<<13)) {
   GPIOD->BSRRH = (1<<13); // Clear bit 13
   } else {
   GPIOD->BSRRL = (1<<13); // Set bit 13
   }
   osSignalSet(t_blinky, 0x0001); // Set Signal
   osDelay(1000);  // delay 1000 msec
   }
 } // end main
```

A thread can wait for multiple signal events and use `osSignalGet()` to determine what actions should be taken on return to READY state, as shown in Figure 19.9.

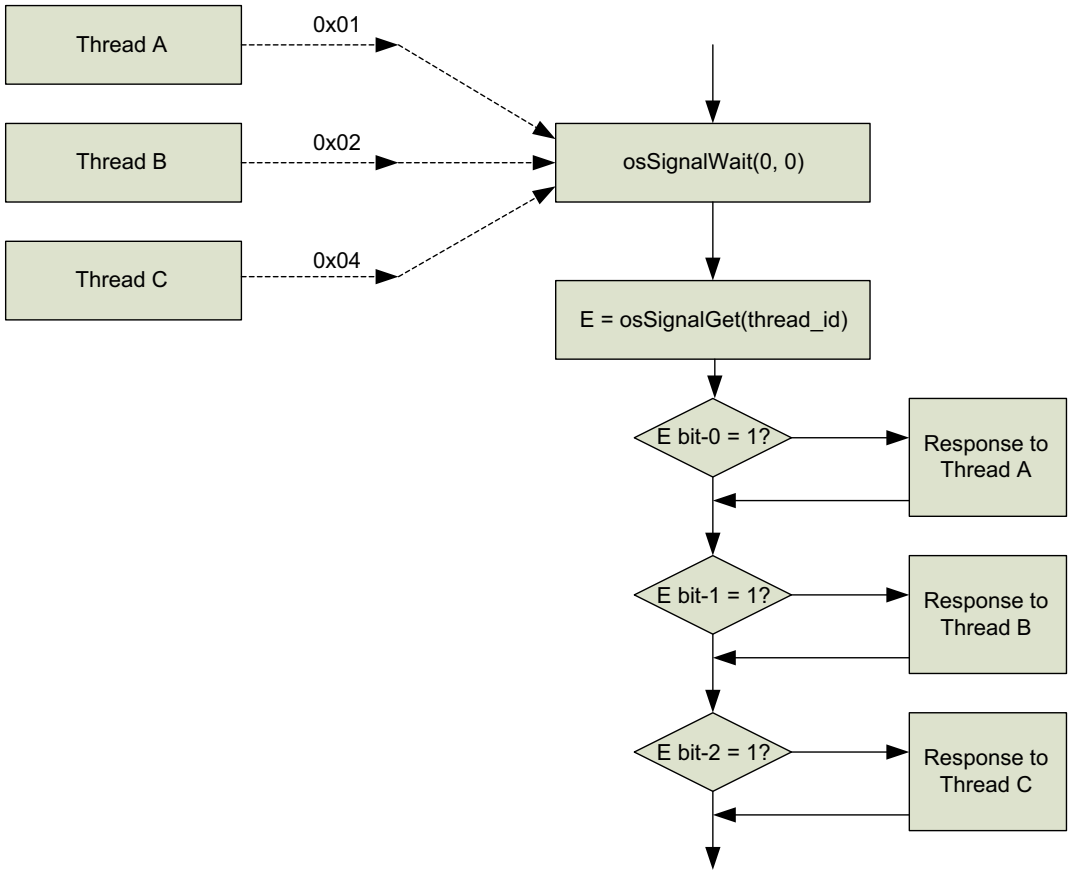

FIGURE 19.9

Using the osSignalGet function to detect which thread generated the signal

19.3.4 Mutual Exclusive (Mutex)

Mutual Exclusive, or commonly known as Mutex, is a common resource management feature in all types of OSs. Many resources in a processor system can only be used by one thread at a time. For example, a "printf" output communication channel (as shown in Figure 19.10) can only be used by one thread at a time.

Before using a Mutex, we first need to define a Mutex object using "`osMutexDef(name)`." When referencing a Mutex using the CMSIS-RTOS Mutex API, we need to use the "`osMutex(name)`" macro. Each Mutex also has an ID value that is needed by some of the Mutex functions. Table 19.10 listed the CMSIS-RTOS functions for Mutex operations.

In the following example, the program code contains two threads. Both of them use the ITM (Instrumentation Trace Macrocell) to output text messages.

```
/* Simple Mutex example – each printf statement is guarded by mutex
to make sure no two printf statements are executed concurrently */

#include "stm32f4xx.h"
#include <cmsis_os.h>
#include "stdio.h"

/* Thread IDs */
osThreadId t_blinky_id; // Declare a thread ID for blink
/* Declare Mutex */
osMutexDef(PrintLock); // Declare a Mutex for printf control
/* Mutex IDs */
osMutexId PrintLock_id; // Declare a Mutex ID for printf control

/* Function declaration */
void blinky(void const *argument); // Thread
void LedOutputCfg(void);           // LED output configuration

// ----------------------------------------------------------
// Blinky
// - toggle LED bit 12
// - Unprivileged Thread
void blinky(void const *argument) {
 while(1) {
   if (GPIOD->IDR & (1<<12)) {
   GPIOD->BSRRH = (1<<12); // Clear bit 12
   } else {
   GPIOD->BSRRL = (1<<12); // Set bit 12
   }
   osDelay(50);  // delay 50 msec
   osMutexWait(PrintLock_id, osWaitForever);
```

```
    printf ("blinky is running\n");
    osMutexRelease(PrintLock_id);
    }
 }
 // define blinky_1 as thread function
 osThreadDef(blinky, osPriorityNormal, 1, 0);

// ----------------------------------------------------------
// - toggle LED bit 13
// - Unprivileged Thread
int main(void)
{
 LedOutputCfg(); // Initialize LED output
 // Create the printf control Mutex before starting blinky thread
 PrintLock_id = osMutexCreate(osMutex(PrintLock));
 osMutexWait(PrintLock_id, osWaitForever);
 printf ("\nMutex Demo\n");
 osMutexRelease(PrintLock_id);

 // Create a task "blinky"
 t_blinky_id = osThreadCreate(osThread(blinky), NULL);

 // main() itself is another thread
 while(1) {
   if (GPIOD->IDR & (1<<13)) {
   GPIOD->BSRRH = (1<<13); // Clear bit 13
   } else {
   GPIOD->BSRRL = (1<<13); // Set bit 13
   }
   osDelay(50);  // delay 50 msec
   osMutexWait(PrintLock_id, osWaitForever);
   printf ("main() is running\n");
   osMutexRelease(PrintLock_id);
   }
   } // end main
```

19.3.5 Semaphore

In some cases we would like to allow a limited number of threads to access certain resources. For example, a DMA controller might be able to support multiple DMA channels. Or a simple embedded server might be able to support a limited number of simultaneous requests due to memory size constraints. In these cases, we can use a semaphore instead of a Mutex.

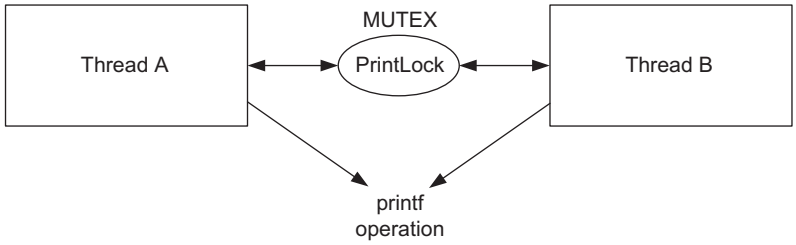

FIGURE 19.10

Using Mutex to control hardware resource sharing

Table 19.10 Mutex Functions

	Function	Description
osMutexId	osMutexCreate(const osMutexDef_t *mutex_def)	Create and Initialize a Mutex object.
osStatus	osMutexWait (osMutexId mutex_id, uint32_t millisec)	Wait until a Mutex becomes available.
osStatus	osMutexRelease (osMutexId mutex_id)	Release a Mutex that was obtained by osMutexWait.
osStatus	osMutexDelete (osMutexId mutex_id)	Delete a Mutex that was created by osMutexCreate.

The semaphore feature is very similar to Mutex. Whereas a Mutex permits just one thread to access to a shared resource at any one time, a semaphore can be used to permit a fixed number of threads to access a pool of shared resources. So a Mutex is a special case of a semaphore for which the maximum number of available tokens is 1.

A semaphore object needs to be initialized to the maximum number of available tokens, and each time a thread needs to use a shared resource, it uses the semaphore to check out a token and then checks it back in when it has finished using the resource. If the number of available tokens reaches zero, then all the available resources have been allocated and the next thread requesting the shared resource must wait for a token to become available.

In the following example, we create four threads that each toggle a LED on the development board, and use a semaphore to limit the number of active LEDs to 2.

Semaphore objects are defined using "`osSemaphoreDef(`*name*`)`." When referencing a semaphore object using the CMSIS-RTOS semaphore API, we need to use the "`osSemaphore(`*name*`)`" macro. Each semaphore also has an ID value that is needed by some of the semaphore functions, as shown in Table 19.11.

In the following example, the program code contains five threads, including main(). Four of them are used to toggle LEDs, and a semaphore is used to limit the number of LEDs that are turned on at any point in time to be two or fewer.

Table 19.11 Semaphore Functions

	Function	Description
osSemaphoreId	osSemaphoreCreate(const osSemaphoreDef_t *semaphore_def, int32_t count)	Create and Initialize a semaphore object.
int32_t	osSemaphoreWait(osSemaphoreId semaphore_id, uint32_t millisec)	Wait until a semaphore becomes available. Returns number of available tokens or −1 in case of incorrect parameters
osStatus	osSemaphoreRelease(osSemaphoreId semaphore_id)	Release a semaphore that was obtained by osSemaphoreWait.
osStatus	osSemaphoreDelete(osSemaphoreId semaphore_id)	Delete a semaphore that was created by osSemaphoreCreate.

```
/* Semaphore example */

#include "stm32f4xx.h"
#include <cmsis_os.h>

/* Thread IDs */
osThreadId t_blinky_id1;
osThreadId t_blinky_id2;
osThreadId t_blinky_id3;
osThreadId t_blinky_id4;
/* Declare Semaphore */
osSemaphoreDef(two_LEDs);  // Declare a Semaphore for LED control
/* Semaphore IDs */
osSemaphoreId two_LEDs_id; // Declare a Semaphore ID for LED control

/* Function declaration */
void blinky(void const *argument); // Thread
void LedOutputCfg(void);   // LED output configuration

// ------------------------------------------------------
// Blinky_1 - toggle LED bit 12
void blinky_1(void const *argument) {
 while(1) {
 // LED on
   osSemaphoreWait(two_LEDs_id, osWaitForever);
   GPIOD->BSRRL = (1<<12); // Set bit 12
   osDelay(500);           // delay 500 msec
```

```
   GPIOD->BSRRH = (1<<12); // Clear bit 12
   osSemaphoreRelease(two_LEDs_id);

   // LED off
   osDelay(500);           // delay 500 msec
   }
  }
// ------------------------------------------------------
// Blinky_2 - toggle LED bit 13
void blinky_2(void const *argument) {
 while(1) {
   // LED on
   osSemaphoreWait(two_LEDs_id, osWaitForever);
   GPIOD->BSRRL = (1<<13); // Set bit 13
   osDelay(600);           // delay 600 msec
   GPIOD->BSRRH = (1<<13); // Clear bit 13
   osSemaphoreRelease(two_LEDs_id);

   // LED off
   osDelay(600);   // delay 600 msec
   }
 }
// ------------------------------------------------------
// Blinky_3 - toggle LED bit 14
void blinky_3(void const *argument) {
 while(1) {
 // LED on
   osSemaphoreWait(two_LEDs_id, osWaitForever);
   GPIOD->BSRRL = (1<<14); // Set bit 14
   osDelay(700);           // delay 700 msec
   GPIOD->BSRRH = (1<<14); // Clear bit 14
   osSemaphoreRelease(two_LEDs_id);

   // LED off
   osDelay(700);   // delay 700 msec
   }
 }
// ------------------------------------------------------
// Blinky_4 - toggle LED bit 15
void blinky_4(void const *argument) {
 while(1) {
   // LED on
   osSemaphoreWait(two_LEDs_id, osWaitForever);
   GPIOD->BSRRL = (1<<15); // Set bit 15
   osDelay(800);           // delay 800 msec
```

```
    GPIOD->BSRRH = (1<<15); // Clear bit 15
    osSemaphoreRelease(two_LEDs_id);

    // LED off
    osDelay(800);   // delay 800 msec
    }
 }
// ----------------------------------------------------------
 // define thread functions
 osThreadDef(blinky_1, osPriorityNormal, 1, 0);
 osThreadDef(blinky_2, osPriorityNormal, 1, 0);
 osThreadDef(blinky_3, osPriorityNormal, 1, 0);
 osThreadDef(blinky_4, osPriorityNormal, 1, 0);
// ----------------------------------------------------------
// - toggle LED bit 13
// - Unprivileged Thread
int main(void)
{
  LedOutputCfg(); // Initialize LED output
  // Create Semaphore with 2 tokens
  two_LEDs_id = osSemaphoreCreate(osSemaphore(two_LEDs), 2);

  // Create "blinky" threads
  t_blinky_id1 = osThreadCreate(osThread(blinky_1), NULL);
  t_blinky_id2 = osThreadCreate(osThread(blinky_2), NULL);
  t_blinky_id3 = osThreadCreate(osThread(blinky_3), NULL);
  t_blinky_id4 = osThreadCreate(osThread(blinky_4), NULL);

  // main() itself is another thread
  while(1) {
  osDelay(osWaitForever); // delay
 }
} // end main
```

19.3.6 Message queue

A message queue can be used to pass a sequence of data from one thread to another in a FIFO-like operation (Figure 19.11). The data can be of integer or pointer type.

Message queue objects are defined using "`osMessageQDef(name, queue_size, type)`." When referencing a message queue object using the CMSIS-RTOS API, we need to use the "`osMessageQ(name)`" macro. Each message queue also has an ID value that is needed by some of the message queue functions, as shown in Table 19.12.

In the following example, a number sequence 1, 2, 3, … is sent from "main()" to another thread called "receiver."

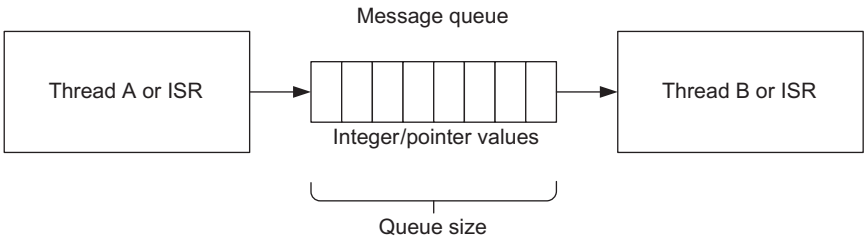

FIGURE 19.11

Message queue

Table 19.12 Message Queue Functions

	Function	Description
osMessageQId	osMessageCreate (const osMessageQDef_t *queue_def, osThreadId thread_id)	Create and Initialize a Message Queue.
osStatus	osMessagePut (osMessageQId queue_id, uint32_t info, uint32_t millisec)	Put a Message to a Queue.
os_InRegs osEvent	osMessageGet (osMessageQId queue_id, uint32_t millisec)	Get a Message or Wait for a Message from a Queue.

```
/* Simple message queue demo */

#include "stm32f4xx.h"
#include "stdio.h"
#include <cmsis_os.h>

/* Declare message queue */
osMessageQDef(numseq_q, 4, uint32_t); // Declare a Message queue
osMessageQId numseq_q_id;             // Declare a ID for message queue

/* Function declaration */
void receiver(void const *argument); // Thread
/* Thread IDs */
osThreadId t_receiver_id;

// ------------------------------------------------------
// Receiver thread
void receiver(void const *argument) {
 while(1) {
   osEvent evt = osMessageGet(numseq_q_id, osWaitForever);
```

```
    if (evt.status == osEventMessage) { // message received
    printf("%d\n", evt.value.v); // ".v" indicate message as 32-bit value
    }
  } // end while
}
 // define thread function
 osThreadDef(receiver, osPriorityNormal, 1, 0);
// ----------------------------------------------------------
int main(void)
{
 uint32_t i=0;
 // Create Message queue
 numseq_q_id = osMessageCreate(osMessageQ(numseq_q), NULL);

 // Create "receiver" thread
 t_receiver_id = osThreadCreate(osThread(receiver), NULL);

 // main() itself is a thread that send out message
 while(1) {
   i++;
   osMessagePut(numseq_q_id, i, osWaitForever);
   osDelay(1000);   // delay 1000 msec
 }
} // end main
// ----------------------------------------------------------
```

An additional example of using a message queue to pass pointers is given in section 19.3.8.

19.3.7 Mail queue

A mail queue (Figure 19.12) is very similar to a message queue, but the information being transferred consists of memory blocks that need to be allocated before putting data in, and freed after taking data out. Memory blocks can hold more information, for example, a data structure, whereas in a message queue the information transferred can only be a 32-bit value or a pointer.

The mail queue object is defined using "`osMailQDef(name, queue_size, type)`." When referencing a mail queue using CMSIS-RTOS API, we need to use the "`osMailQ(name)`" macro. Each mail queue also has an ID value that is needed by some of the mail queue functions, as shown in Table 19.13.

The following example showing how to use a mail queue to pass a block of memory containing a data structure with three elements.

```
/* Mail queue example */

#include "stm32f4xx.h"
```

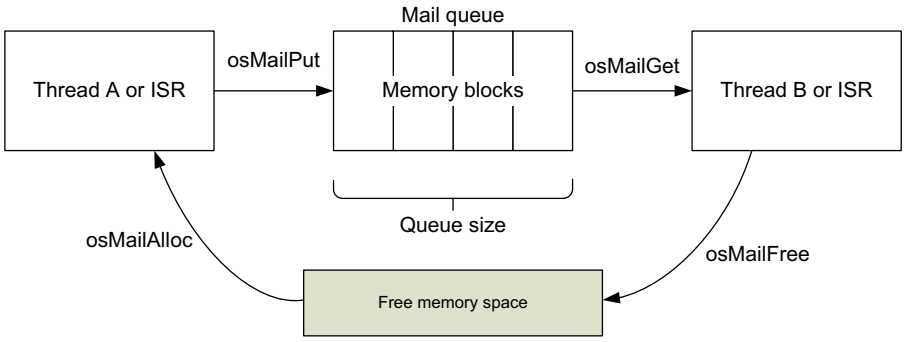

FIGURE 19.12

Mail queue

```
#include "stdio.h"
#include <cmsis_os.h>

typedef struct {
 uint32_t length;
 uint32_t width;
 uint32_t height;
} dimension_t;

/* Declare message queue */
osMailQDef(dimension_q, 4, dimension_t); // Declare a Mail queue
osMailQId dimension_q_id; // Declare a ID for Mail queue
```

Table 19.13 Mail Queue Functions

	Function	Description
osMailQId	osMailCreate (const osMailQDef_t *queue_def, osThreadId thread_id)	Create and initialize a mail queue.
void *	osMailAlloc (osMailQId queue_id, uint32_t millisec)	Allocate a memory block from a mail.
void *	osMailCAlloc (osMailQId queue_id, uint32_t millisec)	Allocate a memory block from a mail and set memory block to zero.
osStatus	osMailPut (osMailQId queue_id, void *mail)	Put a mail to a queue.
os_InRegs osEvent	osMailGet (osMailQId queue_id, uint32_t millisec)	Get a mail from a queue.
osStatus	osMailFree (osMailQId queue_id, void *mail)	Free a memory block from a mail.

```
/* Function declaration */
void receiver(void const *argument); // Thread
/* Thread IDs */
osThreadId t_receiver_id;

// ----------------------------------------------------------
// Receiver thread
void receiver(void const *argument) {
 while(1) {
   osEvent evt = osMailGet(dimension_q_id, osWaitForever);
   if (evt.status == osEventMail) { // mail received
   dimension_t *rx_data = (dimension_t *) evt.value.p;
   // ".p" indicate message as pointer
   printf ("Received data: (L) %d, (W), %d, (H) %d\n",
   rx_data->length,rx_data->width,rx_data->height);
   osMailFree(dimension_q_id, rx_data);
   }
 } // end while
}
 // define thread function
 osThreadDef(receiver, osPriorityNormal, 1, 0);
// ----------------------------------------------------------
int main(void)
{
 uint32_t i=0;
 dimension_t *tx_data;
 // Create Message queue
 dimension_q_id = osMailCreate(osMailQ(dimension_q), NULL);

 // Create "receiver" thread
 t_receiver_id = osThreadCreate(osThread(receiver), NULL);

 // main() itself is a thread that send out message
 while(1) {
   osDelay(1000);       // delay 1000 msec
   i++;
  tx_data=(dimension_t*)osMailAlloc(dimension_q_id,osWaitForever);
   tx_data->length = i; // fake data generation
   tx_data->width = i + 1;
   tx_data->height = i + 2;
   osMailPut(dimension_q_id, tx_data);
   }
 } // end main
// ----------------------------------------------------------
```

19.3.8 Memory pool management feature

CMSIS-RTOS has a feature called Memory Pool Management, which you can use to define a memory pool with a certain number of memory blocks and allocate these blocks during run-time.

The memory pool object is defined using "`osPoolDef(name, pool_size, type)`." When referencing a memory pool object using CMSIS-RTOS API, we need to use the "`osPool(name)`" define. Each memory pool also has an ID value that is needed by some of the memory pool functions, as shown in Table 19.14.

For example, we can repeat the data structure passing in the mail queue example using the message queue feature, and use the memory pool feature to manage the data block in the information transfer.

```
/* Example of message queue passing of data structures using memory
pool */

#include "stm32f4xx.h"
#include "stdio.h"
#include <cmsis_os.h>

typedef struct {
 uint32_t length;
 uint32_t width;
 uint32_t height;
} dimension_t;

/* Declare memory pool */
osPoolDef(mpool, 4, dimension_t);
osPoolId mpool_id;
```

Table 19.14 Memory Pool Functions

	Function	Description
osPoolQId	osPoolCreate (const osPoolDef_t *pool_def)	Create and initialize a memory pool.
void *	osPoolAlloc (osPoolId pool_id)	Allocate a memory block from a memory pool.
void *	osPoolCAlloc (osPoolId pool_id)	Allocate a memory block from a memory pool and set memory block to zero.
osStatus	osPoolFree (osPoolId pool_id, void *block)	Return an allocated memory block back to a specific memory pool.

```
  /* Declare message queue */
 osMessageQDef(dimension_q, 4, dimension_t); // Declare a message
queue
 osMessageQId dimension_q_id; // Declare a ID for message queue
 /* Note: Message queue has 4 entries, same as memory pool size */

 /* Function declaration */
 void receiver(void const *argument); // Thread
 osThreadId t_receiver_id;/* Thread IDs */

 // ------------------------------------------------------
 // Receiver thread
 void receiver(void const *argument) {
  while(1) {
    osEvent evt = osMessageGet(dimension_q_id, osWaitForever);
    if (evt.status == osEventMessage) { // message received
    dimension_t *rx_data = (dimension_t *) evt.value.p;
      // ".p" indicate message as pointer
    printf ("Received data: (L) %d, (W), %d, (H) %d\n",
       rx_data->length,rx_data->width,rx_data->height);
    osPoolFree(mpool_id, rx_data);
    }
   } // end while
 }
  // define thread function
  osThreadDef(receiver, osPriorityNormal, 1, 0);
 // ------------------------------------------------------
 int main(void)
 {
  uint32_t i=0;
  dimension_t *tx_data;
  // Create Message queue
  dimension_q_id = osMessageCreate(osMessageQ(dimension_q), NULL);

  // Create Memory pool
  mpool_id = osPoolCreate(osPool(mpool));

  // Create "receiver" thread
  t_receiver_id = osThreadCreate(osThread(receiver), NULL);

  // main() itself is a thread that send out message
  while(1) {
    osDelay(1000);   // delay 1000 msec
    i++;
```

```
    tx_data = (dimension_t *) osPoolAlloc(mpool_id);
    tx_data->length = i; // fake data generation
    tx_data->width = i + 1;
    tx_data->height = i + 2;
    osMessagePut(dimension_q_id, (uint32_t)tx_data, osWaitForever);
    }
} // end main
// ------------------------------------------------------
```

19.3.9 Generic wait function and time-out value

In all the previous examples we have used a generic function called `osDelay` (Table 19.15).

This is commonly used to put a thread in WAITING state. The input parameter is "millisec" (milli-second).

There is also an osWait function (Table 19.16). However, at the time of writing this function is not supported by the current version of CMSIS-RTOS RTX so cannot be demonstrated here.

In many CMSIS-API functions there is an input parameter called "millisec" to specify the waiting time; for example, `osSemaphoreWait`, `osMessageGet`, etc. In the normal value range it defines the time duration that will trigger a time-out, which causes the function to return. This parameter can be set to a constant definition called osWaitForever, which is defined as `0xFFFFFFFF` in cmsis_os.h. When "millisec" is set to `osWaitForever`, the function will not time out.

When "millisec" is set to 0, the function returns immediately and does not wait. You can use the function return value to determine whether the required operation has succeeded or not.

It is undesirable and disallowed to enter WAITING state in any exception handler. As a result, when using CMSIS-RTOS APIs that have the millisec input parameter, the millisec parameter should be set to 0 so that they return immediately without stopping. Functions that are intended to create delay like `osDelay` should not be used in any interrupt handler.

19.3.10 Timer feature

In addition to the wait and delay functions, CMSIS-RTOS also supports Timer objects. A timer object can trigger the execution of a function. (Note: It is not a thread, although it is possible to send an event to a thread from that function.)

Table 19.15 osDelay Function

	Function	Description
osStatus	osDelay (uint32_t millisec)	Wait for a time period

Table 19.16 osWait Function

	Function	Description
os_InRegs osEvent	osWait (uint32_t millisec)	Wait for Signal, Message, Mail, or Timeout. Return event that contains signal, message, mail information or error code.

A Timer object can operate in periodic timer mode or one-shot mode. In periodic timer mode, the timer repeats its operation until it is deleted/terminated. In one-shot mode the timer triggers its function only once.

A Timer object is defined using "`osTimerDef(name, type, *argument)`." When referencing a timer object using CMSIS-RTOS API, we need to use "`osTimer(name)`" define. Each timer object also has an ID value that is needed by some of the timer functions, as shown in Table 19.17.

The following example shows simple use of a Timer object in both periodic mode and one-shot mode:

```
/* Example for timer objects. The 4 LEDs switch on 1 by 1 in sequence */

#include "stm32f4xx.h"
#include <cmsis_os.h>
/* Function declaration */
void toggle_led(void const *argument); // Toggle LED
void LedOutputCfg(void);               // LED output configuration

/* Declare Semaphore */
osTimerDef(LED_1, toggle_led); // Declare a Timer for LED control
osTimerDef(LED_2, toggle_led); // Declare a Timer for LED control
osTimerDef(LED_3, toggle_led); // Declare a Timer for LED control
osTimerDef(LED_4, toggle_led); // Declare a Timer for LED control
osTimerDef(LED_5, toggle_led); // Declare a Timer for LED control

/* Timer IDs */
osTimerId LED_1_id, LED_2_id, LED_3_id, LED_4_id,LED_5_id ;
// ------------------------------------------------------
// For each round this function get executed 5 times,
// with argument = 1,2,3,4,5
void toggle_led(void const *argument)
{
 switch ((int)argument){
  case 1:
   GPIOD->BSRRL = (1<<12); // Set   bit 12
   osTimerStart(LED_2_id, 500);
```

```
    break;
   case 2:
    GPIOD->BSRRH = (1<<12); // Clear bit 12
    GPIOD->BSRRL = (1<<13); // Set   bit 13
    osTimerStart(LED_3_id, 500);
    break;
   case 3:
    GPIOD->BSRRH = (1<<13); // Clear bit 13
    GPIOD->BSRRL = (1<<14); // Set   bit 14
    osTimerStart(LED_4_id, 500);
    break;
   case 4:
    GPIOD->BSRRH = (1<<14); // Clear bit 14
    GPIOD->BSRRL = (1<<15); // Set   bit 15
    osTimerStart(LED_5_id, 500);
    break;
   default:
    GPIOD->BSRRH = (1<<15); // Clear bit 15
   }
}
// ------------------------------------------------------
int main(void)
{
 LedOutputCfg(); // Initialize LED output

 // Timers
 LED_1_id=osTimerCreate(osTimer(LED_1),osTimerPeriodic,(void*)1);
 LED_2_id = osTimerCreate(osTimer(LED_2), osTimerOnce,  (void *)2);
 LED_3_id = osTimerCreate(osTimer(LED_3), osTimerOnce,  (void *)3);
 LED_4_id = osTimerCreate(osTimer(LED_4), osTimerOnce,  (void *)4);
 LED_5_id = osTimerCreate(osTimer(LED_5), osTimerOnce,  (void *)5);

 osTimerStart(LED_1_id, 3000); // Start first timer

 // main() itself is another thread
 while(1) {
  osDelay(osWaitForever); // delay
  }
} // end main
```

If you are using CMSIS-RTOS RTX, when using timer objects, you should check that the configuration in RTX_Conf_CM.c has the `OS_TIMERS` parameter set to 1. You might also need to configure settings for the Timer thread.

Table 19.17 Timer Functions

	Function	Description
osTimerId	osTimerCreate (const osTimerDef_t *timer_def, os_timer_type type, void *argument))	Create and initialize a timer.
osStatus	osTimerStart (osTimerId timer_id, uint32_t millisec)	Start or restart a timer.
osStatus	osTimerStop (osTimerId timer_id)	Stop the timer.
osStatus	osTimerDelete (osTimerId timer_id)	Delete a timer that was created by osTimerCreate.

19.3.11 Access privileged devices

Depending on the setting of CMSIS-RTOS RTX, "main()" can start in unprivileged state. In this case you cannot access any registers in the NVIC or the System Control Space (SCS), or some of the special registers in the processor core.

To enable "main()" and various threads to run in privileged state, you should set the `OS_RUNPRIV` parameter in RTX_Conf_CM.c to 1. However, there are many applications that require some threads to run in unprivileged state, for example, to enable the system to utilize memory protection features. In this case, it is very likely that you still want to execute some of the procedures in privileged state so that you can set up the NVIC or access other registers in SCS, or special registers in the processor.

In order to solve this problem, the CMSIS-RTOS RTX provides an extendable SVC mechanism. SVC #0 is used by the CMSIS-RTOS RTX, but other SVC services can be used by user-defined functions. The application code can use SVC calls to execute these user-defined functions inside the SVC handler, which executes in privileged state.

A SVC table code needs to be added to the project that carries out the SVC service look-up and defines the name of the user-defined SVC service.

```
  SVC_table.s: Here we only added one user-defined SVC service, but you
 can add more if needed. The name of the user-defined SVC service code
 is called __SVC_1.
            AREA   SVC_TABLE, CODE, READONLY

            EXPORT SVC_Count

  SVC_Cnt   EQU    (SVC_End-SVC_Table)/4
  SVC_Count DCD    SVC_Cnt

  ; Import user SVC functions here.
            IMPORT __SVC_1
```

```
          EXPORT SVC_Table
SVC_Table
; Insert user SVC functions here. SVC 0 used by RTL Kernel.
          DCD  __SVC_1                          ; user SVC function
SVC_End

    END
```

And inside the application code, we define `user_defined_svc(void)` as SVC #1, and implement `__SVC_1`, which is referenced in the SVC table.

```
/* Example of using SVC service to initialize a NVIC register */

#include "stm32f4xx.h"
#include <cmsis_os.h>

void __svc(0x01) user_defined_svc(void); // Define SVC #1 as
user_defined_svc

/* Thread IDs */
osThreadId t_blinky; // Declare a thread ID for blink
/* Function declaration */
void blinky(void const *argument); // Thread
void LedOutputCfg(void);           // LED output configuration

// ---------------------------------------------------------
// Blinky
// - toggle LED bit 12
// - Unprivileged Thread
void blinky(void const *argument) {
 while(1) {
   if (GPIOD->IDR & (1<<12)) {
     GPIOD->BSRRH = (1<<12); // Clear bit 12
   } else {
     GPIOD->BSRRL = (1<<12); // Set bit 12
   }
   osDelay(500);  // delay 500 msec
 }
 }

 // define blinky_1 as thread function
 osThreadDef(blinky, osPriorityNormal, 1, 0);
// ---------------------------------------------------------
// User defined SVC service (#1)
```

```
  // Note that the name must match the SVC service name defined in
// SVC_Table.s
void __SVC_1(void)
{
 // add your NVIC/SCS initialization code here ...
 NVIC_EnableIRQ(EXTI0_IRQn);
 return;
}
// ----------------------------------------------------------
// - toggle LED bit 13
// - Unprivileged Thread
int main(void)
{
 user_defined_svc(); // User defined SVC service (#1)
 LedOutputCfg();     // Initialize LED output

 // Create a task "blinky"
 t_blinky = osThreadCreate(osThread(blinky), NULL);

 // main() itself is another thread
 while(1) {
   if (GPIOD->IDR & (1<<13)) {
     GPIOD->BSRRH = (1<<13); // Clear bit 13
   } else {
     GPIOD->BSRRL = (1<<13); // Set bit 13
   }
   osDelay(1000);  // delay 1000 msec
 }
} // end main
```

19.4 OS-aware debugging

In order to make debugging applications with an RTOS easier, the ITM stimulus port #31 (the last channel) is commonly reserved for OS events in debuggers. This allows the debugger to determine which task is being executed and which events have occurred.

For example, in Keil™ MDK-ARM the µVision debugger has a RTOS task and System view window and an Event Viewer window. They can be accessed using the pull-down menu in the debugger screen: "Debug → OS support → RTX Tasks and System, Debug → OS support → Event Viewer." To use these functions, the "RTX kernel" option needs to be set in the "Target" tab of the project settings (Figure 19.13).

You also need to have trace support in the debug adaptor (either Serial Wire Viewer or Trace Port interface).

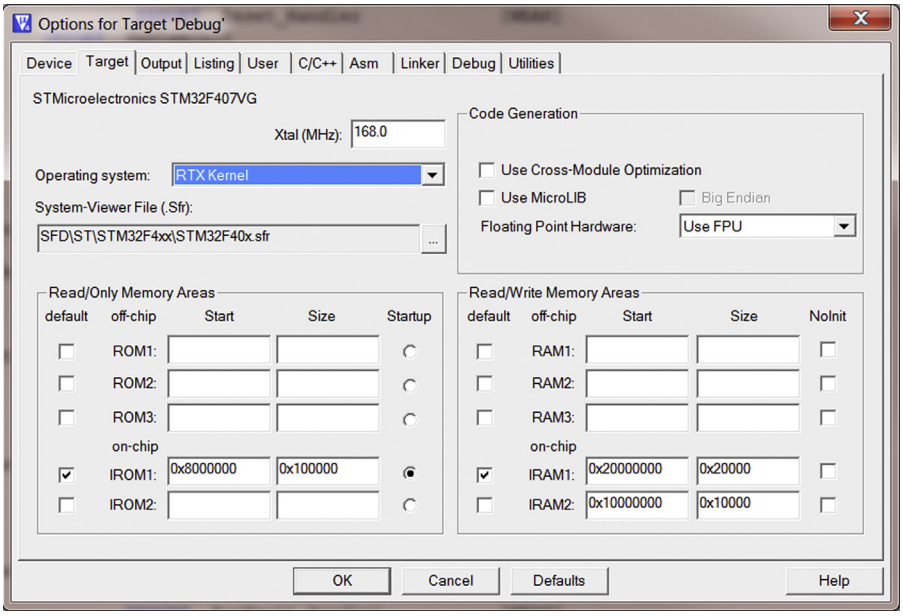

FIGURE 19.13

RTX Kernel option must be set to use OS-aware debug feature

During debugging, the RTX Tasks and System view provides a number of useful pieces of information about the current status of the OS kernel as well as some of the configuration details, as shown in Figure 19.14.

The Event viewer provides a time chart of which thread is currentlty executing, as shown in Figure 19.15.

19.5 Troubleshooting

Chapter 12 and Appendix I cover most of the common issues and troubleshooting techniques. Here are a few more areas that are more specific to embedded OS applications. If an application does not work properly, remember to check the items covered in the following sections.

19.5.1 Stack size and stack alignment

In a number of toolchains you can generate reports to see how much stack each thread requires. You should check this against the stack size setting of your project. This includes the stack size setting in startup code (e.g., Keil™ MDK) or linker configuration (e.g., IAR), the default stack size for main and thread, and stack size options in `osThreadDef` definitions.

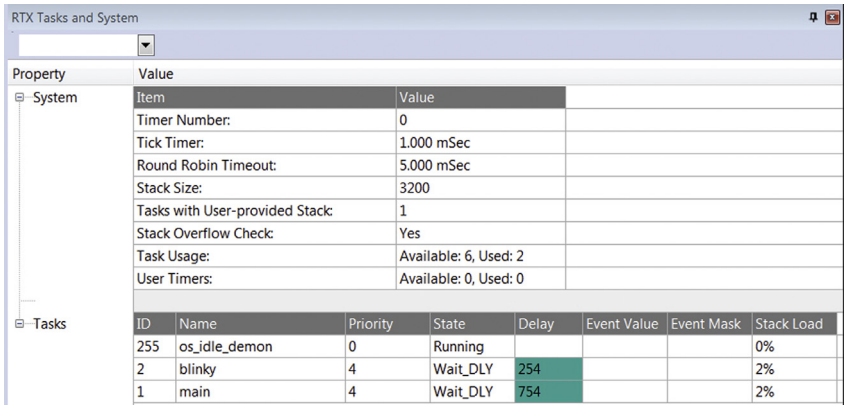

FIGURE 19.14

RTX Tasks and System view

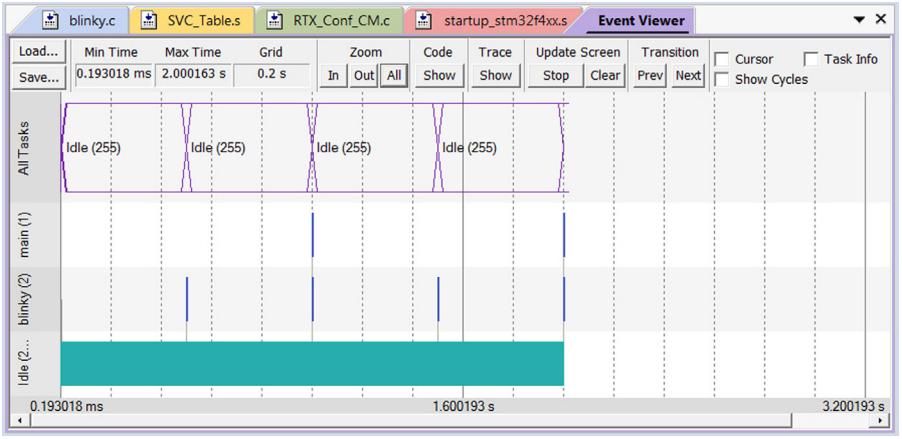

FIGURE 19.15

Event viewer window

In addition, the stack size should be a multiple of 8. You might also need to check the linker report or memory map report to make sure that the stack areas are aligned to double-word boundaries.

19.5.2 Privileged level

If your embedded OS runs threads (or some of them) in unprivileged state, then these threads cannot access SCS areas such as NVIC registers. This can also affect access to the ITM because ITM stimulus ports can be configured to be privileged access

only. Please refer to section 19.3.11 on how to extend SVC services in CMSIS-RTOS RTX.

19.5.3 Miscellaneous

When using CMSIS-RTOS features, remember to create the objects before using them (e.g., using the osXxxxCreate functions). The program code can compile without any issue when some of the create functions are accidentally omitted, but the results can be unpredictable.

When developing the examples ocasionally we have found that the RTX Tasks and System view and Event Viewer in µVision debugger stopped working. Simply unplugging the board and powering it again seems to fix the problem.