UM1722
User manual

Developing Applications on STM32Cube with RTOS

Introduction

The STMCube™ initiative was originated by STMicroelectronics to ease developers life by reducing development efforts, time and cost. STM32Cube covers the STM32 portfolio.

STM32Cube Version 1.x includes:

- The STM32CubeMX, a graphical software configuration tool that allows to generate C initialization code using graphical wizards.
- A comprehensive embedded software platform, delivered per series (namely, STM32CubeF4 for STM32F4 series)
  - The STM32Cube HAL, an STM32 abstraction layer embedded software, ensuring maximized portability across STM32 portfolio
  - A consistent set of middleware components such as RTOS, USB, TCP/IP, Graphics
  - All embedded software utilities coming with a full set of examples.

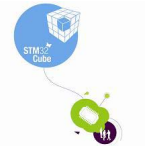
A Real Time Operating System is an operating system optimized for use in embedded/real time applications. Their primary objective is to ensure a timely and deterministic response to events. Using a real time operating system allows applications to be written as a set of independent threads that inter-communicate using message queues and semaphores.

This user manual is intended for developers who use STM32Cube firmware on STM32 microcontrollers. It provides a full description of how to use the STM32Cube firmware components with a real time operating system (RTOS); this user manual comes also with description of a set of examples based on FreeRTOS using the common APIs provided by the CMSIS-OS wrapping layer.

In the STM32Cube firmware FreeRTOS is used as real time operating system through the generic CMSIS-OS wrapping layer provided by ARM. Examples and applications using the FreeRTOS can be direclty ported on any other RTOS without modifying the high level APIs, only the CMSIS-OS wrapper has to be changed in this case.

Please refer to the release notes of the package to know the version of FreeRTOS and CMSIS-RTOS firmware components used with the STM32Cube™.

This document is applicable to all STM32 devices; however for simplicity reason, the STM32F4xx devices and STM32CubeF4 are used as reference platform. To know more about the examples implementation on your STM32 device, please refer to the readme file provided within the associated STM32Cube FW package.



Contents

List of tables

List of figures

1 Free RTOS

1.1 Overview

FreeRTOS is a class of RTOS that is designed to be small enough to run on a microcontroller, although its use is not limited to microcontroller applications.

A microcontroller is a small and resource constrained processor that incorporates, on a single chip, the processor itself, read only memory (ROM or Flash) to hold the program to be executed, and the random access memory (RAM) needed by the programs it executes. Typically the program is executed directly from the read only memory.

Microcontrollers are used in deeply embedded applications (those applications where you never actually see the processors themselves or the software they are running) that normally have a very specific and dedicated job to do. The size constraints, and dedicated end application nature, rarely warrant the use of a full RTOS implementation - or indeed make the use of a full RTOS implementation possible. FreeRTOS therefore provides the core real time scheduling functionality, inter-task communication, timing and synchronization primitives only. This means it is more accurately described as a real time kernel, or real time executive. Additional functionality, such as a command console interface, or networking stacks, can be then be included with add-on components.

FreeRTOS is a scalable real time demonstration builder core designed specifically for small embedded systems. Highlights include

- Free RTOS demonstration builder core-preemptive, cooperative and hybrid configuration options.
- Official support for 27 architectures (counting ARM7 and ARM Cortex M3 as one architecture each).
- FreeRTOS-MPU supports the Cortex M3 Memory Protection Unit (MPU).
- Designed to be small, simple and easy to use. Typically a demonstration builder core binary image will be in the region of 4K to 9K bytes.
- Very portable code structure predominantly written in C.
- Supports both tasks and co-routines.
- Queues, binary semaphores, counting semaphores, recursive semaphores and mutexes for communication and synchronization between tasks, or between tasks and interrupts.
- Mutexes with priority inheritance.
- Supports efficient software timers.
- Powerful execution traces functionality.
- Stack overflows detection options.
- Pre-configured demo applications for selected single board computers allowing out of the box operation and fast learning curve.
- Free forum support, or optional commercial support and licensing.
- No software restriction on the number of tasks that can be created.
- No software restriction on the number of priorities that can be used.
- No restrictions imposed on priority assignment - more than one task can be assigned the same priority.
- Free development tools for many supported architectures.

- Free embedded software source code.
- Royalty free.
- Cross development from a standard Windows host.

The heap2 scheme of FreeRTOS is used for the memory allocation management, this scheme uses a best fit algorithm allows previously allocated blocks to be freed. It does not however combine adjacent free blocks into a single large block. The total amount of available RAM is set by the definition *configTOTAL_HEAP_SIZE* - which is defined in *FreeRTOSConfig.h*.

1.2 License

The FreeRTOS source code is licensed by a modified GNU General Public License. The modification takes the form of an exception. The full text of the GNU General Public License is shown here:

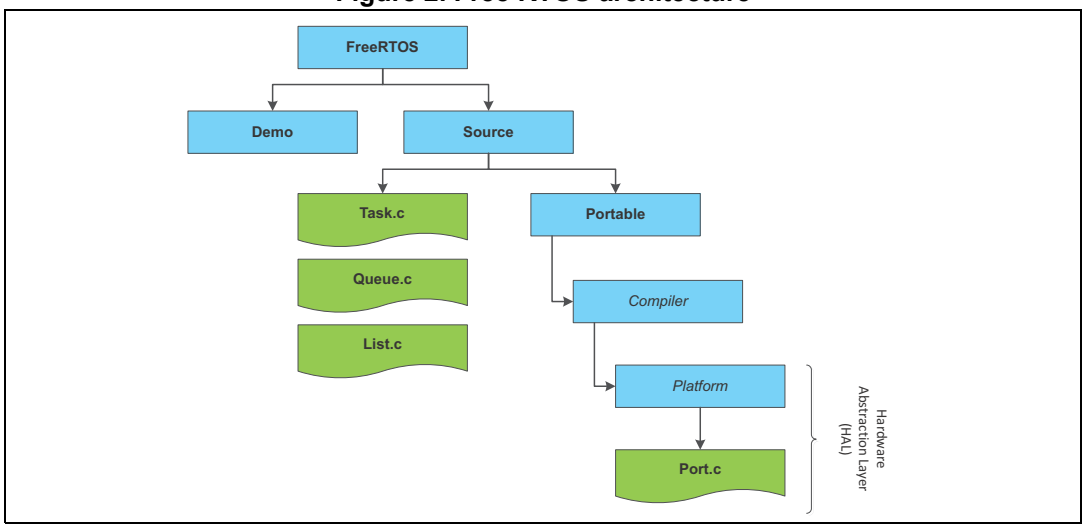
Figure 1. FreeRTOS license

```
The FreeRTOS.org source code is licensed by the modified GNU General Public
License (GPL) text provided below.  The FreeRTOS download also includes
demo application source code, some of which is provided by third parties
AND IS LICENSED SEPARATELY FROM FREERTOS.ORG.

For the avoidance of any doubt refer to the comment included at the top
of each source and header file for license and copyright information.

This is a list of files for which Real Time Engineers Ltd are not the
copyright owner and are NOT COVERED BY THE GPL.


1) Various header files provided by silicon manufacturers and tool vendors
   that define processor specific memory addresses and utility macros.
   Permission has been granted by the various copyright holders for these
   files to be included in the FreeRTOS download.  Users must ensure license
   conditions are adhered to for any use other than compilation of the
   FreeRTOS demo applications.

2) The uIP TCP/IP stack the copyright of which is held by Adam Dunkels.
   Users must ensure the open source license conditions stated at the top
   of each uIP source file is understood and adhered to.

3) The lwIP TCP/IP stack the copyright of which is held by the Swedish
   Institute of Computer Science.  Users must ensure the open source license
   conditions stated at the top  of each lwIP source file is understood and
   adhered to.

4) Various peripheral driver source files and binaries provided by silicon
   manufacturers and tool vendors.  Permission has been granted by the
   various copyright holders for these files to be included in the FreeRTOS
   download.  Users must ensure license conditions are adhered to for any
   use other than compilation of the FreeRTOS demo applications.

5) The files contained within FreeRTOS\Demo\WizNET_DEMO_TERN_186\tern_code,
   which are slightly modified versions of code provided by and copyright to
   Tern Inc.

Errors and omissions should be reported to Richard Barry, contact details for
whom can be obtained from http://www.FreeRTOS.org.


The GPL license text follows.

A special exception to the GPL is included to allow you to distribute a
combined work that includes FreeRTOS without being obliged to provide
the source code for any proprietary components.  See the licensing section
of http://www.FreeRTOS.org for full details.  The exception text is also
included at the bottom of this file.
```

1.3 Free RTOS source organization

The FreeRTOS download includes source code for every processor port, and every demonstration application. Placing all the ports in a single download greatly simplifies distribution, but the number of files may seem daunting. The directory structure is however very simple, and the FreeRTOS real time kernel is contained in just 4 files (additional files are required if software timer or co-routine functionality is required).

Figure 2. Free RTOS architecture



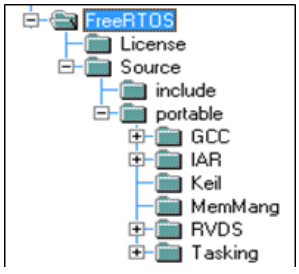
The core RTOS code is contained in three files, called tasks.c, queue.c and list.c., in the FreeRTOS/Source directory. The same directory contains two optional files called timers.c and croutine.c which implement software timer and co-routine functionality. Each supported processor architecture requires a small amount of architecture specific RTOS code. This is the RTOS portable layer, located in the FreeRTOS/Source/Portable/[compiler]/[architecture] sub directories, where [compiler] and [architecture] are the compiler used to create the port, and the architecture on which the port runs, respectively.

The sample heap allocation schemes are also located in the portable layer. The various sample heap_x.c files are located in the FreeRTOS/Source/portable/MemMang directory.

1.4 Porting FreeRTOS on STM32

FreeRTOS supports the following ST processor families: STM32 (Cortex-M0, Cortex-M3 and Cortex-M4F), STR7 (ARM7) and STR9 (ARM9), and can be used with the following tools: IAR, Atollic TrueStudio, GCC, Keil, Rowley CrossWorks.

Figure 3. Free RTOS port



1.5 FreeRTOS API

Table 1. Free RTOS API

APIs Categories	API
Task Creation	– xTaskCreate – vTaskDelete
Task Control	– vTaskDelay – vTaskDelayUntil – uxTaskPriorityGet – vTaskPrioritySet – vTaskSuspend – vTaskResume – xTaskResumeFromISR – vTaskSetApplicationTag – xTaskCallApplicationTaskHook
Task Utilities	– xTaskGetCurrentTaskHandle – xTaskGetSchedulerState – uxTaskGetNumberOfTasks – vTaskList – vTaskStartTrace – ulTaskEndTrace – vTaskGetRunTimeStats
Kernel Control	– vTaskStartScheduler – vTaskEndScheduler – vTaskSuspendAll – xTaskResumeAll
Queue Management	– xQueueCreate – xQueueSend – xQueueReceive – xQueuePeek – xQueueSendFromISR – xQueueSendToBackFromISR – xQueueSendToFrontFromISR – xQueueReceiveFromISR – vQueueAddToRegistry – vQueueUnregisterQueue
Semaphores	– vSemaphoreCreateBinary – vSemaphoreCreateCounting – xSemaphoreCreateMutex – xSemaphoreTake – xSemaphoreGive – xSemaphoreGiveFromISR

1.6 FreeRTOS memory management

Four sample RAM allocation schemes are included in the FreeRTOS source code download (V2.5.0 onwards). These are used by the various demo applications as appropriate. The following sub-sections describe the available schemes, when they should be used, and highlight the demo applications that demonstrate their use.

Each scheme is contained in a separate source file (heap_1.c, heap_2.c, heap_3.c and heap_4.c respectively) which can be located in the Source/Portable/MemMang directory. Other schemes can be added if required.

Scheme 1 - heap_1.c

This is the simplest scheme of all. It does not permit memory to be freed once it has been allocated, but despite this is suitable for a surprisingly large number of applications.

The algorithm simply subdivides a single array into smaller blocks as requests for RAM are made. The total size of the array is set by the definition configTOTAL_HEAP_SIZE - which is defined in FreeRTOSConfig.h. This scheme:

can be used if your application never deletes a task or queue (no calls to vTaskDelete () or vQueueDelete () are ever made).

- is always deterministic (always takes the same amount of time to return a block).
- is used by the PIC, AVR and 8051 demo applications - as these do not dynamically create or delete tasks after vTaskStartScheduler() has been called.

heap_1.c is suitable for a lot of small real time systems provided that all tasks and queues are created before the kernel is started.

Scheme 2 - heap_2.c

This scheme uses a best fit algorithm and, unlike scheme 1, allows previously allocated blocks to be freed. It does not however combine adjacent free blocks into a single large block.

Again the total amount of available RAM is set by the definition configTOTAL_HEAP_SIZE - which is defined in FreeRTOSConfig.h.

This scheme:

- can be used even when the application repeatedly calls vTaskCreate ()/vTaskDelete () or vQueueCreate ()/vQueueDelete () (causing multiple calls to pvPortMalloc() and vPortFree()).
- should not be used if the memory being allocated and freed is of a random size - this would only be the case if tasks being deleted each had a different stack depth, or queues being deleted were of different lengths.
- could possibly result in memory fragmentation problems should your application create blocks of queues and tasks in an unpredictable order. This would be unlikely for nearly all applications but
- should be kept in mind.
- is not deterministic - but is also not particularly inefficient.

heap_2.c is suitable for most small real time systems that have to dynamically create tasks.

Scheme 3 - heap_3.c

This is just a wrapper for the standard malloc() and free() functions. It makes them thread safe. This scheme:

- Requires the linker to setup a heap, and the compiler library to provide malloc() and free() implementations.
- Is not deterministic.
- Will probably considerably increase the kernel code size.
- Is used by the PC (x86 single board computer) demo application.

Scheme 4 - heap_4.c

This scheme uses a first fit algorithm and, unlike scheme 2, does combine adjacent free memory blocks into a single large block (it does include a coalescence algorithm).

The total amount of available heap space is set by configTOTAL_HEAP_SIZE - which is defined in FreeRTOSConfig.h.

The xPortGetFreeHeapSize() API function returns the total amount of heap space that remains unallocated (allowing the configTOTAL_HEAP_SIZE setting to be optimised), but does not provide information on how the unallocated memory is fragmented into smaller blocks.

This implementation:

- Can be used even when the application repeatedly deletes tasks, queues, semaphores, mutexes, etc..
- Is much less likely than the heap_2 implementation to result in a heap space that is badly fragmented into multiple small blocks - even when the memory being allocated and freed is of random size.
- Is not deterministic - but is much more efficient that most standard C library malloc implementations.

heap_4.c is particularly useful for applications that want to use the portable layer memory allocation schemes directly in the application code (rather than just indirectly by calling API functions that themselves call pvPortMalloc() and vPortFree()).

1.7 FreeRTOS low power

It is common to reduce the power consumed by the microcontroller on which FreeRTOS is running by using the Idle task hook to place the microcontroller into a low power state. The power saving that can be achieved by this simple method is limited by the necessity to periodically exit and then re-enter the low power state to process tick interrupts. Further, if the frequency of the tick interrupt is too high, the energy and time consumed entering and then exiting a low power state for every tick will outweigh any potential power saving gains for all but the lightest power saving modes.

The FreeRTOS tickless idle mode stops the periodic tick interrupt during idle periods (periods when there are no application threads that are able to execute), then makes a correcting adjustment to the RTOS tick count value when the tick interrupt is restarted.

Stopping the tick interrupt allows the microcontroller to remain in a power saving state until either an interrupt occurs, or it is time for the RTOS kernel to transition a thread into the Ready state.

1.8 FreeRTOS configuration

A number of configurable parameters exist that allow the FreeRTOS kernel to be tailored to your particular application. These items are located in a file called FreeRTOSConfig.h. Each demo application included in the FreeRTOS source code download has its own FreeRTOSConfig.h file. Here is a typical example

Figure 4. FreeRTOS configuration

```
/* Ensure stdint is only used by the compiler, and not the assembler. */
#if defined(__ICCARM__) || defined(__CC_ARM) || defined(__GNUC__)
 #include <stdint.h>
 extern uint32_t SystemCoreClock;
#endif

#define configUSE_PREEMPTION                    1
#define configUSE_IDLE_HOOK                     0
#define configUSE_TICK_HOOK                     0
#define configCPU_CLOCK_HZ                      ( SystemCoreClock )
#define configTICK_RATE_HZ                      ( ( portTickType ) 1000 )
#define configMAX_PRIORITIES                    ( ( unsigned portBASE_TYPE ) 7 )
#define configMINIMAL_STACK_SIZE                ( ( unsigned short ) 128 )
#define configTOTAL_HEAP_SIZE                   ( ( size_t ) ( 15 * 1024 ) )
#define configMAX_TASK_NAME_LEN                 ( 16 )
#define configUSE_TRACE_FACILITY                1
#define configUSE_16_BIT_TICKS                  0
#define configIDLE_SHOULD_YIELD                 1
#define configUSE_MUTEXES                       1
#define configQUEUE_REGISTRY_SIZE               8
#define configCHECK_FOR_STACK_OVERFLOW          0
#define configUSE_RECURSIVE_MUTEXES             1
#define configUSE_MALLOC_FAILED_HOOK            0
#define configUSE_APPLICATION_TASK_TAG          0
#define configUSE_COUNTING_SEMAPHORES           1
/* Cortex-M specific definitions. */
#ifdef __NVIC_PRIO_BITS
       /* __BVIC_PRIO_BITS will be specified when CMSIS is being used. */
       #define configPRIO_BITS                  __NVIC_PRIO_BITS
#else
       #define configPRIO_BITS                  4        /* 15 priority levels */
#endif

/* The lowest interrupt priority that can be used in a call to a "set priority"
function. */
#define configLIBRARY_LOWEST_INTERRUPT_PRIORITY      0xf

/* The highest interrupt priority that can be used by any interrupt service
routine that makes calls to interrupt safe FreeRTOS API functions */
#define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY     5

/* Interrupt priorities used by the kernel port layer itself.  These are generic
to all Cortex-M ports, and do not rely on any particular library functions. */
#define configKERNEL_INTERRUPT_PRIORITY             (
configLIBRARY_LOWEST_INTERRUPT_PRIORITY << (8 - configPRIO_BITS) )
#define configMAX_SYSCALL_INTERRUPT_PRIORITY  (
configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY << (8 - configPRIO_BITS) )

/* Definitions that map the FreeRTOS port interrupt handlers to their CMSIS
standard names. */
#define vPortSVCHandler SVC_Handler
#define xPortPendSVHandler PendSV_Handler

/* IMPORTANT: This define MUST be commented when used with STM32Cube firmware,
              to prevent overwriting SysTick_Handler defined within STM32Cube HAL
*/
/* #define xPortSysTickHandler SysTick_Handler */
```

Note: SVC_Handler and PendSV_Handler must be removed from stm32f4xx_it.c/.h files when working with FreeRTOS to avoid a duplicate definition

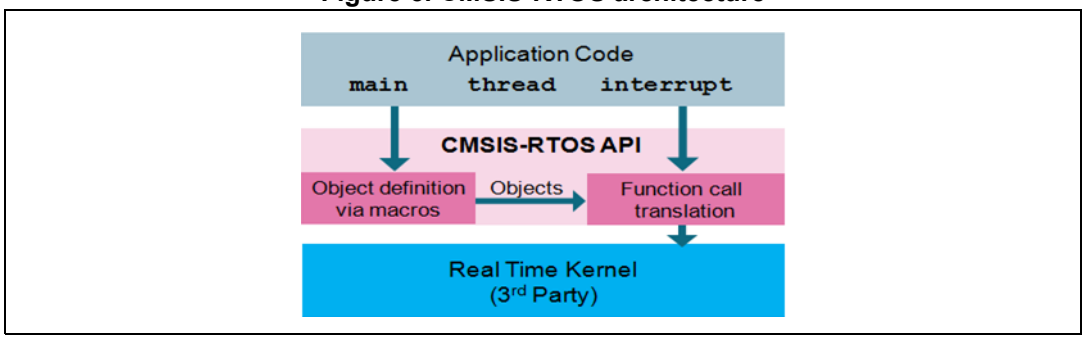
2 CMSIS-RTOS module

2.1 Overview

The CMSIS-RTOS is a common API for Real-Time operating systems. It provides a standardized programming interface that is portable to many RTOS and enables therefore software templates, middleware, libraries, and other components that can work across supported the RTOS systems.

This module is represented by cmsis_os.c/h files located under the following repository “Middlewares\Third_Party\FreeRTOS\CMSIS_RTOS”.

Figure 5. CMSIS-RTOS architecture



A typical CMSIS-RTOS API implementation interfaces to an existing Real-Time Kernel. The CMSIS-RTOS API provides the following attributes and functionality:

- Function names, identifiers, and parameters are descriptive and easy to understand. The functions are powerful and flexible which reduces the number of functions exposed to the user.
- Thread Management allow defining, creating, and controlling threads.
- Interrupt Service Routines (ISR) can call many CMSIS-RTOS functions. When a CMSIS-RTOS function cannot be called from ISR context it rejects the invocation.
- Three different thread event types support communication between multiple threads and/or ISR:
  - Signals: are flags that may be used to signal specific conditions to a thread. Signals can be modified in an ISR or set from other threads.
  - Message: is a 32-bit value that can be sent to a thread or an ISR. Messages are buffered in a queue. The message type and queue size
  - are defined in a descriptor.
  - Mail: is a fixed-size memory block that can be sent to a thread or an ISR. Mails are buffered in a queue and memory allocation is provided. The mail type and queue size
  - are defined in a descriptor.
- Mutex Management and Semaphore Management are incorporated.
- CPU time can be schedule
- d with the following functionality:
  - A timeout parameter is incorporated in many CMSIS-RTOS functions to avoid system lockup. When a timeout is specified the system waits until a resource is available or event occurs. While waiting, other threads are scheduled.

- The osDelay function puts a thread into the state WAITING for a specified period of time.
- The generic osWait function waits for events that are assigned to a thread.
- The osThreadYield provides co-operative thread switching and passes execution to another thread of the same priority.

The CMSIS-RTOS API is designed to optionally incorporate multi-processor systems and/or access protection via the Cortex-M Memory Protection Unit (MPU).

In some RTOS implementation threads may execute on different processors and Mail and Message queues can therefore reside in shared memory resources.

The CMSIS-RTOS API encourages the software industry to evolve existing RTOS implementations. Kernel objects are defined and accessed using macros. This allows differentiation. RTOS implementations can be different and optimized in various aspects towards the Cortex-M processors. Optional features may be for example:

- Generic Wait function; i.e. with support of time intervals.
- Support of the Cortex-M Memory Protection Unit (MPU).
- Zero-copy mail queue.
- Support of multi-processor systems.
- Support of a DMA controller.
- Deterministic context switching.
- Round-robin context switching.
- Deadlock avoidance, for example with priority inversion.
- Zero interrupt latency by using the Cortex-M3/M4 instructions LDEX and STEX.

2.2 CMSIS-RTOS API

The following list provides a brief overview of all CMSIS-RTOS API:

Table 2. CMSIS-RTOS API

Module	API	Description
Kernel Information and Control	osKernelInitialize	Initialize the RTOS kernel
	osKernelStart	Start the RTOS kernel
	osKernelRunning	Query if the RTOS kernel is running
	osKernelSys Tick (*)	Get RTOS kernel system timer counter
	osKernelSys TickFrequency (*)	RTOS kernel system timer frequency in Hz
	osKernelSys TickMicroSec (*)	Convert microseconds value to RTOS kernel system timer value

Table 2. CMSIS-RTOS API (continued)

Module	API	Description
Thread Management: Define, create and control thread functions	osThreadCreate	Start execution of a thread function.
	osThreadTerminate	Stop execution of a thread function.
	osThreadYield	Pass execution to next ready thread function.
	osThreadGetId	Get the thread identifier to reference this thread.
	osThreadSetPriority	Change the execution priority of a thread function.
	osThreadGetPriority	Obtain the current execution priority of a thread function.
Generic Wait Functions: Wait for a time period or unspecified events.	osDelay	Wait for a specified time.
	osWait (*)	Wait for any event of the type Signal, Message, or Mail.
Timer Management (*): Create and control timer and timer callback functions.	osTimerCreate	Define attributes of the timer callback function
	osTimerStart	Start or restart the timer with a time value.
Signal Management: Control or wait for signal flags.	osSignalSet	Set signal flags of a thread.
	osSignalClear	Reset signal flags of a thread.
	osSignalClear	Suspend execution until specific signal flags are set.
Mutex Management (*): Synchronize thread execution with a Mutex.	osMutexCreate	Define and initialize a mutex
	osMutexWait	Obtain a mutex or Wait until it becomes available.
	osMutexRelease	Release a mutex
	osMutexDelete	Delete a mutex
Semaphore Management (*): Control access to shared resources.	osSemaphoreCreate	Define and initialize a semaphore.
	osSemaphoreWait	Obtain a semaphore token or Wait until it becomes available.
	osSemaphoreRelease	Release a semaphore token.
	osSemaphoreDelete	Delete a semaphore.
Memory Pool Management (*): Define and manage fixed-size memory pools.	osPoolCreate	Define and initialize a fix-size memory pool.
	osPoolAlloc	Allocate a memory block.
	osPoolCAlloc	Allocate a memory block and zero-set this block.
	osPoolFree	Return a memory block to the memory pool.

Table 2. CMSIS-RTOS API (continued)

Module	API	Description
Message Queue Management (*): Control, send, receive, or wait for messages.	osMessageCreate	Define and initialize a message queue.
	osMessageCreate	Put a message into a message queue.
	osMessageCreate	Get a message or suspend thread execution until message arrives
Mail Queue Management (*): Control, send, receive, or wait for mail.	osMailCreate	Define and initialize a mail queue with fix-size memory blocks
	osMailAlloc	Allocate a memory block
	osMailCAlloc	Allocate a memory block and zero-set this block
	osMailPut	Put a memory block into a mail queue
	osMailGet	Get a mail or suspend thread until mail arrives.
	osMailFree	Return a memory block to the mail queue.

Modules or APIs marked with (*) are optional.

3 FreeRTOS applications

The STM32CubeF4 FreeRTOS package comes with several applications that use the stack API sets.

The applications are divided into two categories

Table 3. Free RTOS application categories

Categories	Applications
Getting started (basic)	Thread creation example
	Semaphore between threads example
	Semaphore from ISR example
	Mutexes example
	Queues example
Features	Timer example
	Low power example

3.1 Thread Creation example

A real time application that uses an RTOS can be structured as a set of independent threads. Each thread executes within its own context with no coincidental dependency on other threads within the system or the RTOS scheduler itself. Only one thread within the application can be executing at any point in time and the RTOS scheduler is responsible for deciding which thread this should be.

The aim of this example is to explain how to create threads using CMSIS-RTOS based on FreeRTOS API.

The example implements two threads running with the same priority, which execute in a periodic cycle. Below details about each thread execution.

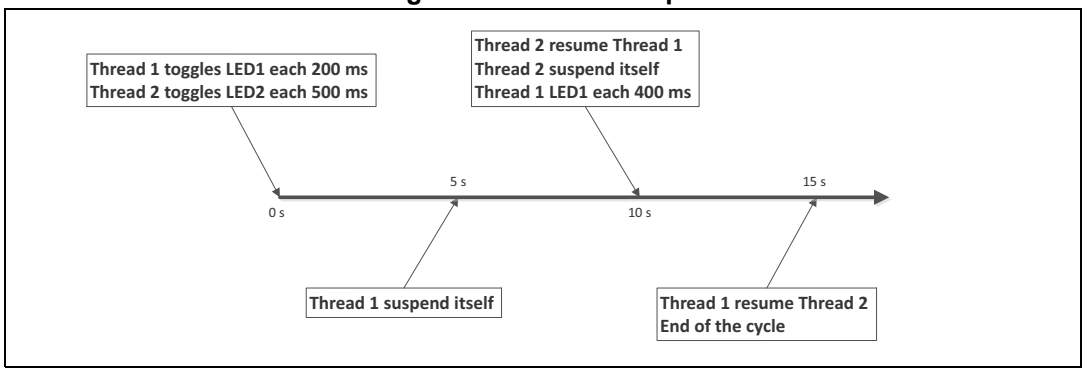
Thread 1: this thread toggles the LED1 each 200 ms for 5 seconds and then it suspends itself, after 5 seconds the thread 2 resume the execution of thread 1 which toggles the LED1 each 400 ms for the next 5 seconds.

Thread creation description:

```
/* Thread 1 definition */
osThreadDef(LED1, LED_Thread1, osPriorityNormal, 0,
configMINIMAL_STACK_SIZE);


/* Start thread 1 */
LEDThread1Handle = osThreadCreate (osThread(LED1), NULL);
```

Thread 2: this thread toggles the LED2 each 500 ms for 10 seconds then it suspend itself. Thread 1 will resume the execution of thread 2 after 5 seconds.

Figure 6. Thread example



Using the example:

- Build and program the application code into the STM32 Flash memory
- Run the example and check that LEDs are toggling as described in *Figure 6*.

3.2 Semaphores examples

Semaphores are used for both mutual exclusion and synchronization purposes.

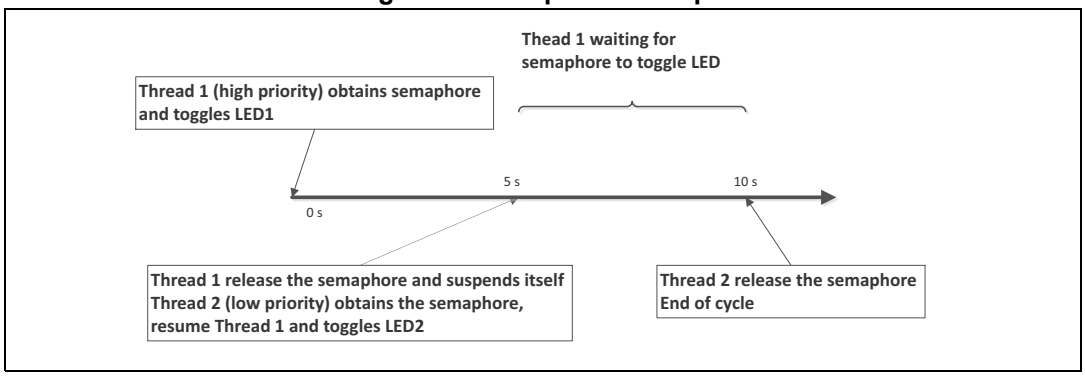
3.2.1 Semaphore between threads

The aim of this example is to explain how to use semaphores through CMSIS-RTOS based on FreeRTOS APIs.

This example implements two threads with different priorities that share a semaphore to toggle LEDs, following more details about the execution of the example.

1. The thread 1 which has the higher priority obtains the semaphore and toggle the LED1 for 5 seconds
2. The thread 1 releases the semaphore and suspends itself.
3. The low priority thread can execute now, it obtains the semaphore and resume execution of the thread 2.
4. As it has the higher priority the thread 1 will try to obtain the semaphore, but it blocks because the semaphore is already taken by the low priority thread,
5. Thread 2 will toggle the LED2 for 5 seconds, before releasing the semaphore and begin a new cycle.

Figure 7. Semaphore example



Semaphore creation description:

```
/* Define the semaphore */
osSemaphoreDef(SEM);

/* Create the binary semaphore */
osSemaphoreId osSemaphore = osSemaphoreCreate(osSemaphore(SEM), 1);
```

Using the example:

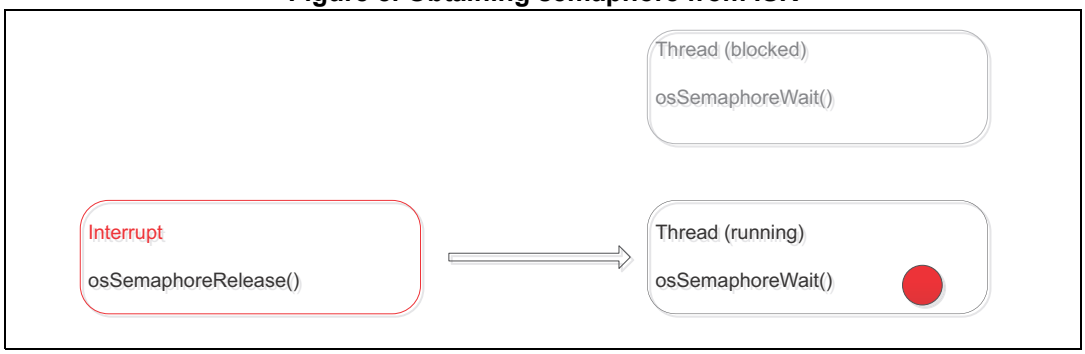
1. Build and program the application code into the STM32 Flash memory
2. Run the example and check that LEDs are toggling as described in *Figure 7*.

3.2.2 Semaphore from ISR

This example demonstrates how to use semaphores from interrupts.

It consists of a basic thread waiting undefinedly for a semaphore to toggle a LED. The semaphore is released when the STM32 generates an interrupt after pushing the KEY button of the evaluation board by the user

Figure 8. Obtaining semaphore from ISR



Using the example

1. Build and program the application code into the STM32 Flash memory
2. Run the example and check that LED1 is toggling when pushing **KEY** button of the evaluation board.

3.3 Mutexes example

Mutexes are binary semaphores that include a priority inheritance mechanism. Whereas binary semaphores are the better choice for implementing synchronization (between tasks or between tasks and an interrupt), mutexes are the better choice for implementing simple mutual exclusion.

This example creates three threads with different priorities, which access to the same mutex. Following

1. the High priority thread executes first and grabs the mutex and sleeps for a short period to let the lower priority threads execute.
2. the Medium priority thread attempts to access the mutex by performing a blocking 'wait'. This thread blocks when the mutex is already taken by the high priority thread. It does not unblock until the high priority thread has released the mutex, and it does not actually run until the high priority thread has suspended itself.
3. the Low priority thread spins round a tight loop attempting to obtain the mutex with a non-blocking call. As the lowest priority thread it will not successfully obtain the mutex until both high and medium priority threads are suspended.
4. the High priority thread gives the mutex back before suspending itself.
5. the Medium priority thread obtain the mutex, all it does is give the mutex back prior to also suspending itself. At this point both the high and medium priority threads are suspended.
6. the Low priority thread obtain the mutex, it first resumes both suspended threads prior to giving the mutex back, resulting in the low priority thread temporarily inheriting the highest thread priority.

Mutex creation description:

```
/* Define the mutex */
osMutexDef(osMutex);

/* Create the mutex */
osMutexId osMutex = osMutexCreate(osMutex(osMutex));
```

Using the example:

1. Build and program the application code into the STM32 Flash memory
2. When running in debug mode, add the following variables to the debugger live watch: HighPriorityThreadCycles, MediumPriorityThreadCycles and LowPriorityThreadCycles; these three variables must remain equal. LED1, LED2 and LED4 should toggle indefinitely and LED3 will turn on in case of error

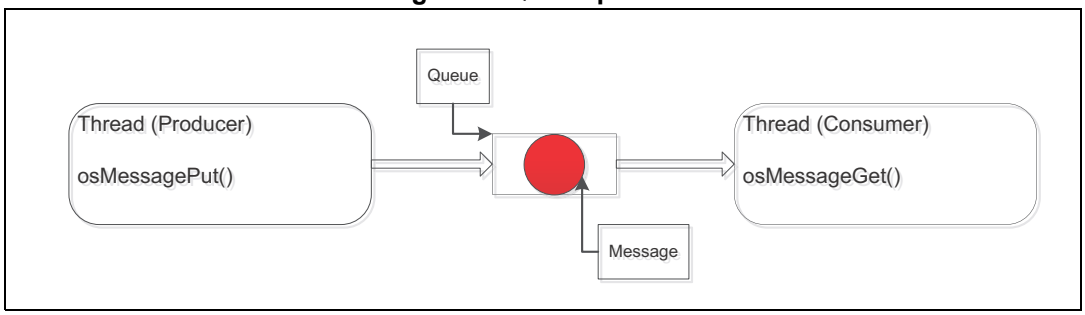
3.4 Queues example

Queues are the primary form of intertask communications. They can be used to send messages between tasks, and between interrupts and tasks. In most cases they are used as thread safe FIFO (First In First Out) buffers with new data being sent to the back of the queue, although data can also be sent to the front.

This example creates two threads that send and receive an incrementing number to/from a queue. One thread acts as a producer and the other as the consumer.

The consumer is a higher priority than the producer and is set to block on queue reads. The queue only has space for one item, as soon as the producer posts a message on the queue the consumer will unblock, preempt the producer, and remove the item.

Figure 9. Queue process



Queue creation description:

```
/* Define a queue with "QUEUE_SIZE" items of 2 bytes */
osMessageQDef(osqueue, QUEUE_SIZE, uint16_t);

/* Create the queue */
osMessageQId osQueue = osMessageCreate (osMessageQ(osqueue), NULL);
```

Using the example:

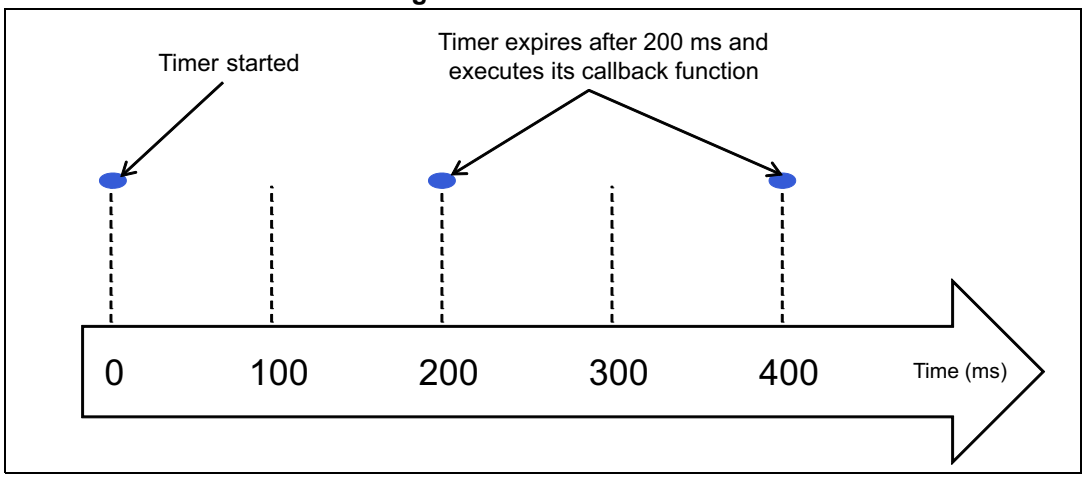
1. Build and program the application code into the STM32 Flash memory
2. Run the example and check that LED1 toggles for each correct message received, else LED3 will toggle.

3.5 Timer example

A timer allows a function to be executed at a set time in the future. The function executed by the timer is called the timer's callback function. The time between a timer being started, and its callback function being executed, is called the timer's period. Put simply, the timer's callback function is executed when the timer's period expires.

This example demonstrates how to use timers of CMSIS RTOS API based on FreeRTOS API., creating a periodic timer that calls a callback function every 200 ms to toggle the LED1 of the evaluation board.

Figure 10. Periodic timer



Periodic timer creation description:

```
/* Define a timer with "osTimerCallback" as callback process */
osTimerDef(LEDTimer, osTimerCallback);

/* Create the timer */
osTimerId osTimer = osTimerCreate (osTimer(LEDTimer), osTimerPeriodic,
NULL);
```

Using the example:

1. Build and program the application code into the STM32 Flash memory
2. Run the example and check that LED1 toggles every 200 ms (Timer expiration)

Note: To use FreeRTOS software timers please add "timers.c" to your project workspace.

3.6 Low power example

This example demonstrates how to run FreeRTOS in low power mode using STM32 devices (for more information about FreeRTOS low power mode refer to *Section 1.7*).

Built-in tickless idle functionality (low power) is enabled by defining configUSE_TICKLESS_IDLE as 1 in FreeRTOSConfig.h

In this example two threads and a queue are created with the following functionality:

- The first thread "RxThread" blocks on a queue to wait for data, toggles an LED each time data is received (turning it on and then off again) before returning to block on the queue once more.
- The second thread "TxThread" repeatedly enters the blocked state for 500ms. On exiting the blocked state the "TxThread" sends a message through the queue to the "RxThread" (causing the "RxThread" to exit the blocked state and toggle the LED).

When the two threads are blocked, the kernel stops the tick interrupt and place the STM32 into low power (sleep) mode to reduce the power consumption.

Table 4 present power consumption measured on STM32F4 devices in the context of the example described above.

Table 4. Comparison of power consumption

Hardware platform	Runtime mode	Sleep mode
STM324xG-EVAL	62.4 mA	14.2 mA
STM324x9I-EVAL	80.5 mA	20.8 mA

4 Conclusions

This User Manual explains how to integrate the FreeRTOS middleware components within the STM32Cube HAL drivers.

A set of examples have been described to help users developing applications with CMSIS-RTOS API based on FreeRTOS operating system.

5 FAQ

How to port FreeRTOS to different Cortex-M cores?

To port FreeRTOS to the right Cortex-M product you have to import the “port.c” from the correct folder. For example if the microcontroller has Cortex-M0 core with IAR tool, you have to get the port.c file from “FreeRTOS\Source\portable\IAR\ARM_CM0” repository.

How much ROM/RAM does FreeRTOS use?

This depends on your compiler, architecture, and RTOS kernel configuration. Generally the RTOS kernel itself required about 5 to 10 KBytes of ROM space.

RAM usage increase if the number of created threads or queues increases.

How to set the CPU clock?

The CPU clock is defined by configCPU_CLOCK_HZ in FreeRTOSConfig.h, within STM32CubeF4 firmware it’s provided by SystemCoreClock which represent the HCLK clock (AHB bus), this value is set when configuring the RCC clock by calling SystemClock_Config() function.

How to set interrupt priorities?

Any interrupt service routine that uses an RTOS API function must have its priority manually set to a value that is numerically equal to or greater than the *configMAX_SYSCALL_INTERRUPT_PRIORITY* setting in *FreeRTOSConfig.h* file.

This ensures the interrupt's logical priority is equal to or less than the *configMAX_SYSCALL_INTERRUPT_PRIORITY setting.*

How to use a clock other than SysTick to generate the tick interrupt?

User can optionally provide its own tick interrupt source by generating an interrupt from a timer other than SysTick:

- Provide an implementation of vPortSetupTimerInterrupt() that generates an interrupt at the frequency specified by the configTICK_RATE_HZ FreeRTOSConfig.h constant.
- Install xPortSysTickHandler() as the handler for the timer interrupt, and ensure xPortSysTickHandler() is not mapped to SysTick_Handler() in FreeRTOSConfig.h, or renamed as SysTick_Handler() in port.c.

How to enable the tickless idle mode?

The FreeRTOS tickles mode (low power), allows reducing the power consumption of the MCU by entering sleep mode and stopping the periodic tick interrupt. This functionality is enabled by defining configUSE_TICKLESS_IDLE as 1 in FreeRTOSConfig.h

The tickless idle mode can be enabled when a timer other than SysTick is used to generate the tick interrupt. User has to add the following actions to those described in the previous question:

- Set configUSE_TICKLESS_IDLE to 2 in FreeRTOSConfig.h.
- Define portSUPPRESS_TICKS_AND_SLEEP() as described in the documentation page on the FreeRTOS website.

6 Revision history

Table 5. Document revision history

Date	Revision	Changes
18-Feb-2014	1	Initial release.
23-Jun-2014	2	On cover page updated: – document title – reference at STM32CubeF4 into STM32Cube

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

ST PRODUCTS ARE NOT DESIGNED OR AUTHORIZED FOR USE IN: (A) SAFETY CRITICAL APPLICATIONS SUCH AS LIFE SUPPORTING, ACTIVE IMPLANTED DEVICES OR SYSTEMS WITH PRODUCT FUNCTIONAL SAFETY REQUIREMENTS; (B) AERONAUTIC APPLICATIONS; (C) AUTOMOTIVE APPLICATIONS OR ENVIRONMENTS, AND/OR (D) AEROSPACE APPLICATIONS OR ENVIRONMENTS. WHERE ST PRODUCTS ARE NOT DESIGNED FOR SUCH USE, THE PURCHASER SHALL USE PRODUCTS AT PURCHASER’S SOLE RISK, EVEN IF ST HAS BEEN INFORMED IN WRITING OF SUCH USAGE, UNLESS A PRODUCT IS EXPRESSLY DESIGNATED BY ST AS BEING INTENDED FOR “AUTOMOTIVE, AUTOMOTIVE SAFETY OR MEDICAL” INDUSTRY DOMAINS ACCORDING TO ST PRODUCT DESIGN SPECIFICATIONS. PRODUCTS FORMALLY ESCC, QML OR JAN QUALIFIED ARE DEEMED SUITABLE FOR USE IN AEROSPACE BY THE CORRESPONDING GOVERNMENTAL AGENCY.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.
Information in this document supersedes and replaces all information previously supplied.
The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com